92

OpenForum

The Pan-European Open Systems Event

Proceedings of the Technical Conference

Utrecht
The Netherlands
25 - 27 Nov
1992

Distributed Computing

Practice and Experience

Proceedings of
the Autumn 1992
OpenForum
Technical Conference

November 23–27, 1992
Utrecht, Holland

ISBN 1 873611 05 6 (EurOpen)
ISBN 0 936593 24 5 (UniForum)

These proceedings will also be available on the conference CDROM.
Further copies of the proceedings may be obtained from:

EurOpen Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

UniForum
2901 Tasman Drive
Suite 201
Santa Clara
CA 95054
USA

These proceedings were typeset in Times Roman, Palatino and Courier on a PostScript printer driven by a sleepy dwarf. PostScript was generated using **refer**, **tt**, **grap**, **pic**, **psfig**, **tbl**, **sed**, **eqn**, **troff**, **pbm**, **xfig**, **xv**, **pm** and **psdit**.

Whilst every care has been taken to ensure the accuracy of the contents of this work, no responsibility for loss occasioned to any person acting or refraining from action as a result of any statement in it can be accepted by the author(s) or publisher.

Acknowledgements

Welcome to Utrecht for the OpenForum 1992 technical conference.

OpenForum is the first pan-European event sponsored by the EurOpen and UniForum associations. OpenForum combines a technical conference, a business and strategy conference, and tutorial seminars.

The theme of the technical conference is "Distributed Systems, Practice and Experience". Consequently, the programme committee looked for practical results when the programme was compiled. I want to thank the committee, consisting of Lori S. Grob, Heinz Lycklama, Robbert van Renesse, Santosh Shrivastava, and Volker Tschammer, for the time and energy they have put into making the technical conference a potential success.

We had the luxury of selecting one out of every fourth submission. We would like to thank all authors who submitted papers to this conference. We are also happy to have Andrew S. Tanenbaum to give the keynote speech. Thanks are also due to the invited speakers, Ozalp Babaoglu, David Powell, and Brian Bershad. Without these contributions we could not have such a quality programme.

Thanks are due to the accompanying organizational and secretarial work, especially to Helen Gibbons and her team at Owles Hall. I would also like to give particular thanks to Pierre Scheuer.

Many thanks to the typesetting crew based at Imperial College; Stuart McRobert and Jan-Simon Pendry. Without their tremendous efforts, there would have been no proceedings.

In closing, thanks must go to all of you, the participants. Without your contributions and your presence, there would not be a conference.

I hope you enjoy OpenForum,

Dag Johansen
Programme Chairman
OpenForum Technical Conference

These proceedings were specially produced for OpenForum at the Department of Computing, Imperial College, London, using resources generously provided by the Computing Support Group. Philip Male and Danny Turner of Computer Newspaper Services Ltd kindly produced the master bromides.

– sm, jsp.

List of Sponsors for OpenForum'92

88Open	*BULL*	*Computer Associates*
Cognos	*Chorus Systèmes*	*Tolerance*
Archipel	*CinCom*	*Data General*
Digital Equipment	*Dun & Bradstreet*	*Hewlett-Packard*
Hicomp	*IBM*	*Informix*
Ingres	*Largotim*	*NCC*
Olivetti	*Oracle*	*OSF*
Progress	*SAP*	*Sequent*
Software AG	*Uniface*	

Table of Contents

Keynote Address

Micro-Kernels

Invited Talk

Distributed Systems Management

Parallelisation

Distributed Systems Development

Performance/Scheduling

Distributed File Systems

Invited Talk

Communications

UNIX Conferences in Europe 1977–1992

UKUUG/NLUUG meetings

1977 May	Glasgow University
1977 September	University of Salford
1978 January	Heriot Watt University, Edinburgh
1978 September	Essex University
1978 November	Dutch Meeting at Vrije University, Amsterdam
1979 March	University of Kent, Canterbury
1979 October	University of Newcastle
1980 March 24th	Vrije University, Amsterdam
1980 March 31st	Heriot Watt University, Edinburgh
1980 September	University College, London

EUUG/EurOpen Meetings

1981 April	CWI, Amsterdam, The Netherlands
1981 September	Nottingham University, UK
1982 April	CNAM, Paris, France
1982 September	University of Leeds, UK
1983 April	Wissenschaft Zentrum, Bonn, Germany
1983 September	Trinity College, Dublin, Eire
1984 April	University of Nijmegen, The Netherlands
1984 September	University of Cambridge, UK
1985 April	Palais des Congres, Paris, France
1985 September	Bella Center, Copenhagen, Denmark
1986 April	Centro Affari/Centro Congressi, Florence, Italy
1986 September	UMIST, Manchester, UK
1987 May	Helsinki/Stockholm, Finland/Sweden
1987 September	Trinity College, Dublin, Ireland
1988 April	Queen Elizabeth II Conference Centre, London, UK
1988 October	Hotel Estoril-Sol, Cascais, Portugal
1989 April	Palais des Congres, Brussels, Belgium
1989 September	Wirtschaftsuniversität, Vienna, Austria
1990 April	Sheraton Hotel, Munich, West Germany
1990 October	Nice Acropolis, Nice, France
1991 May	Kulturhuset, Tromsø, Norway
1991 September	Budapest, Hungary
1992 April	Jersey, Channel Islands

OpenForum Meetings

1992 Novemeber	Utrecht, The Netherlands

Author Index

Distributed Operating Systems Anno 1992 What Have We Learned So Far?

Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam, The Netherlands
ast@cs.vu.nl

Abstract

Research on distributed systems has been going on for over a decade. Perhaps it is time to sit back and take stock. What have we learned in this time? In this paper the author presents one viewpoint of what is important in the field. The topics covered include system structure, communication, and distributed shared memory. The paper concludes by discussing some open issues for future research.

1. Introduction

When I signed up to give this keynote address, I saw it as an opportunity to sit back and reflect on what we have learned about distributed systems so far. After some thought, I came up with a list of what I thought were the main concepts most researchers in the field agreed with. To test out these ideas, I posted them to the USENET news group `comp.os.research`, a sleepy little backwater, populated by gentle souls interested in fairly esoteric matters. I asked for comments and additions to my modest list or alternative lists. I thought a few people might agree with me, and another few might have some minor changes. I was wrong.

My posting ignited a firestorm. Hundreds of responses were posted, most of them ranging in tone from livid to outraged. I was simultaneously accused of being both a hopeless dreamer and an old stick-in-the-mud. Much spleen was vented.

When the dust had settled, I carefully read all the responses and noticed something curious. While many people were most unhappy with something or other that I had written, few had alternatives. This led to my second posting, in which I specifically challenged the previous posters (whose names I had carefully collected) to each produce a list of the five most important things we had learned so far about distributed systems. I decided to be a sport, and included my own list in the posting. I thought I would now get nice fat lists at which I could take a few pot shots. I was wrong again.

Another firestorm ensued, with everyone going after my list again, but only one or two people proposing alternative lists. Several hypotheses come to mind to explain this sequence of events:

1. It is more fun to jump on someone else's ideas than to think up your own.
2. Researchers in distributed systems do not read `comp.os.research`.
3. As a profession, we are a bunch of prima donnas who listen to no one.
4. We really have not learned anything at all in the past 20 years.

I am willing to go along with some combination of 1-3, but I do not agree with 4. I strongly believe we *have* learned some things about distributed operating systems. In this paper I will tell what they are. It may well be that I am the only person in the field who believes these things (but I very much doubt it), but if this paper does nothing else but stimulate thought and discussion about the subject, I think it will have been worth the effort. To emphasize the possibly personal nature of these thoughts, I will write the paper in the first person, a style that is normally frowned upon in Science.

What is a Distributed System?

Before answering the question of what I have learned about distributed operating systems, I have to first explain what I mean by the term. When the term "distributed system" first became popular, at least one vendor began advertising that it already had a distributed system for sale, consisting of a large mainframe to which several hundred dumb ASCII terminals could be connected. This is not quite what I had in mind.

I think that a distributed system has two essential characteristics:

- The system has a number of independent, autonomous, communicating CPUs.
- The system looks to the users like a single computer.

Each of these requires some explanation. The first point means that the system consists of separate computers connected by a network. The old mainframe is still not a distributed system, even if the ASCII terminals are replaced by X-terminals each containing a 50 MIPs CPU, 64M of RAM, a large hard disk to hold all the fonts, and a fiber optic connection to the mainframe. An X-terminal is not an independent computer; it is still operating as a terminal.

A shared-memory multiprocessor, while interesting and important, is also not a distributed system. The CPUs are not independent (e.g., a failure in one of them can corrupt the others). More important, some of the hardest problems that occur in distributed systems, such as lack of global state information, agreement about the exact time, and process synchronization, are easy to solve in multiprocessors. This fact makes their operating systems simpler.

Another common computing model consists of a collection of personal computers or workstations, which all have access to one or more file servers. This configuration meets the first criteria, but not necessarily the second. If each user "owns" one workstation, and can only use other (idle) workstations with some special effort, the whole system does not look like a single computer. When running an interactive editor, the distinction is minimal. The real test comes when a process forks off a sequence of compute-bound children. In a true distributed system, the operating system, not the user, would place each new process on the "best" machine, taking into consideration CPU load, memory availability, location of files needed, network bandwidth, communication patterns, and other factors. Thus a network of personal workstations that use a common file system (e.g., NFS or Andrew) is not a distributed system.

In contrast, one possible way to build a distributed system would be to assemble a collection of single board computers located in a rack in the machine room. Each board would contain a CPU, private memory, a network interface, and its own copy of the operating system. A user at an X-terminal can type a command, which is then executed in this processor pool. Simple jobs may only require one CPU, but other jobs, such as a chess program, may require hundreds of CPUs working in parallel. The location of processors, files, and everything else is done automatically, by the operating system. To the users, the computing power is a single shared resource, like an old-fashioned timesharing system, only constructed with modern technology.

A slightly far-fetched, but possibly instructive, analogy to a distributed system is the worldwide telephone system. The telephone system consists of many thousands of switches (essentially large computers), no one of which is the boss of all the others. Whether a call is routed over fiber optic links, microwave links, coaxial cables, or some combination of them is up to the system, not the user. Similarly, whether analog or digital technology is used is also hidden, as are gateways between different telephone companies and different countries. Calling an 800 number (free number) is a complicated process, involving looking up the "real" number in a distributed data base, but this, too, is not visible as a separate part of the system. To the telephone user, the whole thing looks like a single gigantic switch to which every telephone in the world is connected. This illusion, sometimes called the *single system image*, along with the use of independent computers to implement it, is what sets distributed systems apart from their nondistributed cousins.

It should be clear from the above discussion, that being distributed has at least as much to do with the software (i.e., the operating system) as it has to do with the hardware configuration. The same hardware can either form a distributed system or not, depending on the software.

Few distributed computer systems are currently available commercially, although they will start becoming more numerous during the 1990s. Various prototypes are currently under construction in research laboratories. It is from these experimental systems that most of our knowledge on the subject has come from. It is perhaps worth pointing out explicitly that distributed systems are not easy to design and program – if they were, we would have had a lot more of them in commercial use already. The fact that they are difficult to program suggests to me that we should use techniques that simplify their construction wherever possible. This point will come up over and over in the rest of the paper.

2. System Structure

A good place to begin our discussion is the structure of the operating system, since it is here that the biggest changes have occurred. In this, and subsequent sections, I will try to summarize what I have learned in a short observation i⌒ boldface type, followed by my reasons for believing it.

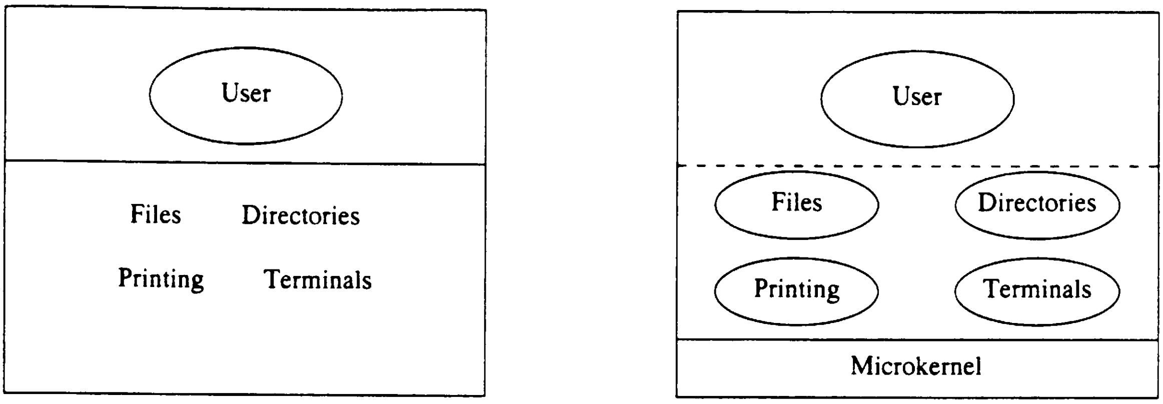


Figure 1: *(a) System with a monolithic kernel. (b) System with a microkernel. Ellipses are processes.*

Observation 1: Distributed operating systems should be based on microkernels

A microkernel is a relatively small operating system kernel with a limited functionality. It runs on the bare hardware. Unlike a conventional operating system, which is intended to directly support application programs, a microkernel is intended to make it possible for system programmers to use it as a base for building various operating systems on top of it. The services provided are those needed by operating systems builders, not those needed by ordinary users. A comparison between a microkernel-based system and a conventional (i.e., monolithic) system is shown in Figure 1.

Microkernels are basically a recent phenomenon, although a case can be made that Brinch Hansen's RC 4000 system had some of the ideas 20 years ago [Han73a]. Well-known examples of microkernels are Amoeba [Tan90a], Chorus [Roz88a], Mach [You87a], and V [Che88a].

For years, operating system designers have made a distinction between mechanism and policy. The *mechanism* is the code that actually does the work. The *policy* is what the user wants done. Microkernels sharpen this distinction by putting much of the mechanism in the microkernel, but leaving the policy out. For example, a microkernel might implement priority-based process scheduling, but allow the owner of a group of processes to set the relative priorities of the processes himself (or herself).

Microkernels typically perform the following functions:

- Interprocess communication
- Low-level process management
- Low-level memory management
- Input/Output

How much they do in each area and the exact functionality they provide varies from system to system, but it is clearly less than in conventional operating systems. Each of these points will be described briefly below.

Since processes in a distributed system can run on disjoint computers, at the lowest level, interprocess communication must be based on message passing. The microkernel must provide some primitives to send and receive messages. Many options are possible, including synchronous vs. asynchronous, reliable vs. unreliable, blocking vs. non-blocking, buffered vs. nonbuffered, and point-to-point vs. group communication. Each combination has its own properties.

Distributed systems need processes as an abstraction tool, just as centralized ones do. Managing these processes is clearly a job for the microkernel. Due to the existence of considerable parallelism in distributed systems, having multiple threads within a single process is often desirable. Although some thread packages can run entirely in user space, without the kernel even being aware of them, for performance and other reasons, it is often useful for the kernel to provide at least some support for threads.

Memory management is clearly a microkernel function, at least in part. The MMU has to be set up, and page faults have to be handled. Some of the work can be done outside the kernel, as in Mach.

Input/output is also done in the microkernel, but typically as an artifact of current architectures. If there is a simple, portable, protected way to allow, say, a disk driver, access to the I/O instructions and I/O ports it needs while running in user mode, there is no reason to put the driver in the microkernel. If the architecture makes this impossible, then the driver has to go in the microkernel.

There is a clear analogy between microkernels and RISC machines. RISC machines have fewer instructions than CISC machines. The guiding principle of RISC design is: if an instruction is not essential, leave it out. To a considerable extent, the same holds for microkernels: if the same functionality can be provided outside the kernel, put it outside the kernel. In short, microkernels can be thought of as RISC operating systems.

The same general advantages that RISC machines have over CISC machines also hold for microkernels: simplicity, modularity, and flexibility. First, they are simpler to design because they do less. They are also easier to implement and debug because there are fewer lines of code there. Since the correct functioning of the entire system is critically dependent on the kernel working correctly, it stands to reason that making the kernel small and simple will also make it more reliable.

Second, microkernels lead to modular systems. Removing the file system from the kernel does not make it go away, but having the file system run as a user process does make it easier to test and debug the file system, since a file system crash does not bring the whole system down, as is the case with a monolithic kernel.

Closely related to modularity is the third advantage, flexibility. The same microkernel can be used as the base for multiple operating systems. The Mach kernel can run UNIX and MS-DOS simultaneously. Microkernels also allow users with special needs (e.g. data base designers) to design and implement their own file systems. With monolithic systems, you are pretty much stuck with whatever file system the operating system provides, which is sometimes a serious problem [Sto81a].

The primary potential disadvantage of a microkernel-based system is performance. On a single processor, consider the difference between a monolithic kernel containing the file system, and a microkernel-based system in which the file system is a user process. In the former, a READ system call traps to the kernel, does the work, and returns. In the latter, it requires sending a message to the file system and getting a reply. Doing so may be slower than just trapping to the kernel, depending on what optimizations are used.

In a distributed system, the situation is somewhat different. The file system is on another machine anyway, and the relative difference in

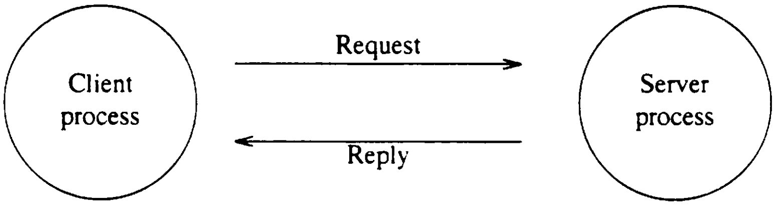


Figure 2: *The client-server paradigm*

cost between sending a message to a file system in a remote kernel vs. sending one to a file system running as a remote user process is small. I do not believe that this small difference in performance even comes close to outweighing the advantages of simplicity, modularity, and flexibility.

Put in other terms, I think the key challenge facing us is how to make the software work, not how to make the system go a little bit faster. If a small amount of performance has to be sacrificed to produce software that is easier to write, get correct, and maintain, I think this is a worthwhile tradeoff.

Observation 2: The client-server paradigm is a good one

Using a microkernel is only part of the story. There remains the issue of how to structure the rest of the software – the part that is outside the microkernel. One paradigm that is widely used, and with great success, is the *client-server paradigm*. In this model, system services are provided by *server processes*, each process typically offering one specific service. Examples are file servers, directory servers, print servers, time servers, mail servers, data base servers, and so on.

Application programs run as *client processes*, normally on different machines from the servers. To obtain service, a client sends a message to a server, which then carries out the work and sends back a reply. During the course of an application, a client may interact with a variety of different servers to get the job done, but each interaction is structured as a request from the client to a server, followed by a reply from the server back to the client, as shown in Figure 2.

The client-server model fits in well with the idea of a microkernel. By having the servers run outside the kernel, they can be highly modular, with each server providing one service well (in the spirit of UNIX). This design also makes it possible to have multiple file servers (e.g., UNIX and MS-DOS) running simultaneously and offering different services to different clients.

The client-server model represents a paradigm shift from systems like UNIX in an important way. In traditional operating systems, everything is either a process or a file. Almost all the system calls relate to manipulating processes and files.

The client-server model, in contrast, is based on abstract data types (objects). A server can define and make available for use any kind of objects it wants to, and provide whatever operations are needed on these objects. It encapsulates the objects and allows only the permitted operations to be performed on the objects. This paradigm allows information hiding and makes it easier to construct correct and easy-to-understand programs.

While it is true one can simulate objects in a system like UNIX by putting them in files and creating daemon processes to manage them, it is

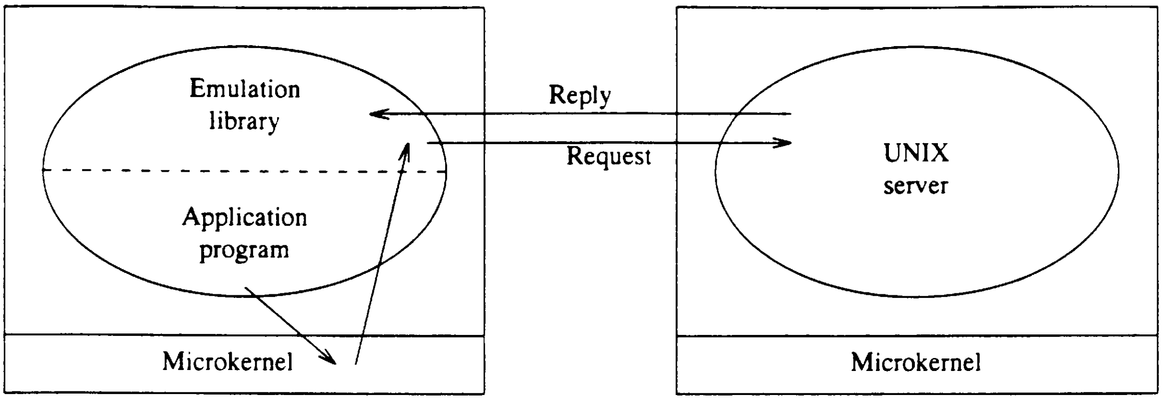


Figure 3: *Achieving binary compatibility with UNIX*

an unnatural way of programming in UNIX, and it is in direct contrast to the mental model that most programmers have that "everything is a file."

Observation 3: UNIX can be successfully run as an application program

If one accepts the idea that the operating system should be based on a microkernel, upon which server processes run, then the logical conclusion is that UNIX (or MS-DOS or other operating systems) should be run as applications. Existing distributed systems have tried two different approaches, both of which seem satisfactory.

In the short run, binary compatibility is often needed to run existing software whose source code is not available. Mach and Chorus have taken this route. One way to provide binary compatibility is to map a (shared) emulation library into the top part of the address space of all UNIX processes. When a process makes a system call, the microkernel catches the call and reflects it back to a handler inside the emulation library. The library then calls a UNIX server to do the work, as shown in Figure 3.

The other approach is to provide a library of UNIX system calls, (OPEN, READ, etc.) and recompile existing programs to use the new library. The library routines can make use of whatever services are needed to get the job done and do not have the overhead of trapping to the kernel. In the long run, it is the most efficient approach since it avoids the emulation trap overhead. This approach is taken in Amoeba.

Many optimizations can be applied here, such as trying to minimize the number of calls to the external UNIX server [Gol90a]. While there is clearly some performance loss compared to running a monolithic UNIX on a single processor, once again I believe that the advantages of having distributed systems such as having more CPU power available, modularity, flexibility, fault tolerance, and more, outweighs the slight loss in performance and makes this approach acceptable.

3. Communication

Since a distributed system consists of many processes communicating with one another, a key issue in the design of any distributed operating

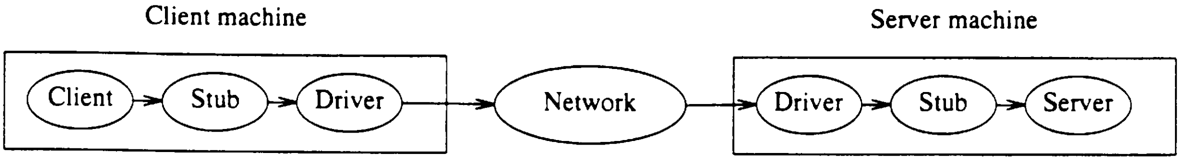


Figure 4: *Request part of a remote procedure call. The reply follows the reverse path.*

system is how communication is managed. In this area I have three observations.

Observation 4: Remote procedure call is a good communication model

The client-server paradigm provides a framework for having processes interact, but leaves open the question of what the communication primitives should be. In a now-classic paper, Birrell and Nelson showed a way to hide the communication altogether, so the only abstraction needed by the client and server was the long-familiar procedure call [Bir84a]. This technique, known as *Remote Procedure Call* (RPC), has become widely used as the basis for communication in distributed operating systems. It is illustrated in Figure 4.

Briefly, for each service that a server offers, it provides a corresponding procedure, called a *stub*, in the library so that clients can use it. For example, a file server might provide stubs for *read* and *write*, each with appropriate parameters. To read data from a file, the client calls *read*. The *read* stub then collects its parameters and puts them into a message, along with a code indicating which operation is desired. It then passes the message to the network driver, which sends it to the server. Once there, the server's network driver passes the message to the server stub, which unpacks the parameters and calls the server as a procedure. When the server has finished its work, it returns to its stub, which builds a reply message and sends it back to the client.

The beauty of this scheme is that neither the client procedure nor the server procedure have to know that they are engaging in remote communication. The client calls a local procedure (its stub) using the usual calling sequence to pass parameters. Similarly, the server is called by a local procedure, and thus gets its parameters on the stack in the usual way. It notices nothing strange.

With RPC, all the message passing is hidden away in the stubs, which are usually generated by a special compiler from a functional description of the server's interface. This mechanism is an enormous advance over having the client and server directly sending and receiving messages, an unfamiliar and error-prone way of programming. By forcing all communication to be synchronous (callers are blocked until their calls finish), many kinds of subtle programming errors are eliminated. Using RPC means that programming the client-server model in a distributed system is not all that different from traditional sequential programming.

RPC also has several disadvantages. For the sophisticated programmer with a special application requiring unusual communication patterns, RPC can be restricting. It is sometimes argued that given SEND and RECEIVE primitives, it is possible for those programmers who want RPC to build it, and those who do not to ignore it. This is like saying that a programming language should just supply the IF and GOTO statements, since FOR and WHILE loops can be constructed from them. This argument is specious. Given that the major problem facing us is how to make all this complicated software work, it is essential that the system provide high-level primitives, not low-level ones in the hope the programmers will not abuse them. Applications that absolutely cannot live with synchronous communication should use multiple threads combined with RPC to achieve parallelism in a clean way.

RPC also has the disadvantage that it is hard to use it along with global variables and unconstrained pointer use. Whether the loss of global variables is bad is arguable. Pointers are a more serious issue, but experience shows that it is usually manageable.

Observation 5: Globally ordered, reliable broadcasting is useful

A distributed system can be constructed and used in various ways. If a large number of CPUs are available for use, either idle workstations or a centralized collection of processors in the machine room, it may be possible to harness multiple machines to work together in parallel to speed up a single application. Alternatively, the system may be designed this way from the beginning.

For certain applications, communication is generally not from a client to a server, so RPC is not appropriate. More often, communication is from one process to many processes. While it is possible to simulate this one-to-many (i.e., broadcast or multicast) communication by repeated RPCs, doing so is inefficient. Furthermore, if two or more processes are engaged in one-to-many communication at the same time, the messages may be interleaved, leading to race conditions and errors.

As a simple example, consider a data base system that replicates its data on multiple machines so that each one can handle queries in parallel with the others. Reads are done locally, but when a record is changed, it must be updated on all machines simultaneously to avoid inconsistencies. Other examples involve simpler shared data structures, where multiple machines need to read and write the same variables. Having a basic primitive to broadcast a message to all machines reliably and indivisibly is a valuable tool.

Various distributed systems have supported broadcasting in diverse forms, starting with V and ISIS [Che88a, Bir91a]. Based on considerable experience with Amoeba, I have seen that having reliable, indivisible, globally ordered broadcasting as a basic primitive makes parallel programming (and also fault-tolerant programming) of distributed systems much easier. Two key properties are needed:

- Either a message is received by all interested processes or by none
- All processes receive all messages in the same order

Using this broadcasting, it is possible to update a variable in all processes simultaneously, without having to worry about the consequences of lost messages, interleaved messages, and other problems. This mechanism can be used to construct even easier-to-use abstractions, such as shared data-objects [Bal91a]. Not having this kind of broadcasting makes programming these applications much more difficult.

The difference between reliable, globally ordered broadcasting and not having it is illustrated in Figure 5. In this example, machines *A* and *B* simultaneously want to broadcast messages. In one case, first *A* goes then *B* (or vice versa), as shown in Figure 5(a) and (b). In the other case, Figure 5(c), some processes may get the messages in the order (*A*, *B*), while others get them in the order (*B*, *A*). Having the system guar-

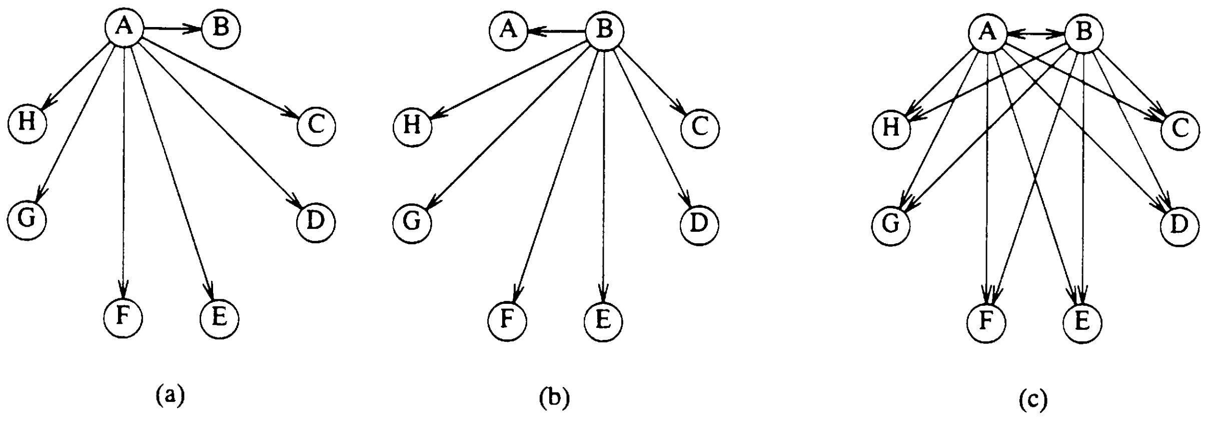


Figure 5: *(a)-(b) Ordered broadcast, first A then B. (c) Interleaved broadcast*

antee that all processes get all broadcasts in exactly the same order, with no lost messages and no interleaving of concurrent broadcasts, makes programming much easier than having weaker semantics.

Observation 6: Communication transparency is important

When one process talks to another process (or to a group of processes, using broadcasting), it should not have to worry about the relative location of the processes. Communication schemes that have one semantics when the communicating parties are on the same machine and different semantics when they are on different machines make programming harder. Programmers have enough to worry about without location being an additional issue (analogous to MS-DOS programmers having to treat memory addresses below 640K, between 640K and 1M, between 1M and 1M+64K, and above 1M as different categories).

The issue of transparency arises in a variety of contexts. For example, having to make distinctions between two machines on the same LAN, two machines on interconnected LANs, and two machines in different countries. For example, if files are named using something like `/machine/filename` or `machine!filename` the system is not transparent. In such a system, it is impossible for the file system to transparently move files from one server to another (e.g., to balance the load) because the location is effectively visible to the users.

4. Distributed Shared Memory

Multiple CPU systems come in two forms: multiprocessors (which have shared memory), and multicomputers (which do not). Building large multiprocessors is difficult due to the need to keep memory coherent over potentially thousands of processors. On the other hand, these systems are straightforward to program. Processes can directly share variables and can synchronize using semaphores, monitors, and other well-known techniques.

Multicomputers, in contrast, are easy to build since each CPU-memory pair is independent of all the other ones. They are harder to program, however, especially for parallel applications. In recent years, a hybrid form has been developed that simulates shared memory on multicomputers. This technique is called *distributed shared memory* and leads to the next observation.

Observation 7: Distributed shared memory makes parallel programming easier

Distributed shared memory comes in two varieties: page based and object based. Page-based distributed shared memory was pioneered by Li and Hudak [Li89a]. It operates as a paging system across machine boundaries. All machines share a common virtual address space, with the pages themselves spread over the machines as need be. When a page fault occurs, the needed page is fetched from the machine that is currently holding it. Read-only pages can be replicated, but read-write pages must be unmapped from the current host, as they may not be present in two machines at the same time.

With object-based distributed shared memory the unit of sharing is defined by software objects [Bal92a, Car89a, Jul88a]. Operations are defined on objects, and when an operation is executed on an object, the software that manages the object goes and gets the object if it has to.

In both cases, the multicomputer programmer is presented with the illusion of having a form of shared memory. Processes on different machines can use the shared memory for communication and synchronization, which is usually much more convenient than message passing. In Amoeba, object-based shared memory is implemented by replicating objects on all machines, doing reads locally, and doing writes using reliable, globally ordered broadcasting [Tan92a]. Other implementation techniques are also possible, of course.

5. Open Issues

Many other design issues are still open. In this section I will briefly mention some points that are potentially subjects for inclusion in the second edition of this paper, assuming we can figure out which ideas are good and which are not.

5.1. Machine Ownership

The issue of machine ownership will occur acutely when the economics make it possible to provide, say, 32 times as many CPUs as there are users. In one model, each user gets a personal 32-node multiprocessor, exclusively for his or her own use. It is likely that nearly all this computing power will be idle all the time, which by itself is not so bad, but once in a while a user will want to start a heavily compute-bound job (e.g. heuristic search in A.I., VLSI placement and routing, or analysis of a digitized photograph). In the pure "personal multiprocessor" model, no one may use anyone else's machine, so all the idle CPUs are wasted.

If users are permitted to find and use other people's idle CPUs, then the communication model can no longer be based on shared memory, since some of the CPUs will be local and others remote. Programming this mixed-mode model will be complicated. In addition, if a CPU machine has been declared "idle" for lack of activity for n minutes and then the owner tries to use it, either the owner is out of luck or the foreign process must be migrated somewhere quickly. If n is small, there will be a lot of migration; if it is large, there will be few CPUs available.

As an alternative model, it might be better to say that most (or all) CPUs are public, with no owner. For example, they could all be put in the machine room in a big rack, and dynamically allocated among users as needed. In this "pool processor" model, all the users get is a simple workstation for running the window system and simple interactive processes, like editors. From queueing theory it is well known that the performance of a system with a large shared resource is better than that of a system in which the resource has been divided statically into *n* fixed, dedicated chunks. Which model is more (cost) effective remains to be seen.

5.2. File Caching

In a distributed system, there are various places one can do caching. These include the server's main memory, the client's main memory, and the client's local disk (if any). There is little controversy about the value of server caching. The problem arises when the caching is done on the client side. When the workstation model is used, it is at least clear where the caching has to be done – in the client machine. The complication arises when two clients have the same file cached and both want to modify it. Either a central administration is needed to keep track of who has which file open for reading or writing, or unsolicited messages are needed to ask clients to delete files from their cache, or something else. There does not appear to be any clean, scalable solution known yet.

In the pool processor model, the issue is even worse. There is no guarantee that just because user *u* happened to be assigned processor *p* last time, that he will get it again next time. That depends on many factors. A serious question about whether client caching on *p* is worth doing at all arises here.

5.3. Thread Management

There seems to be general agreement having multiple threads of control in a process is frequently useful, both in centralized and distributed systems. The question that arises is how much should the kernel know about the threads. At one extreme, the kernel manages the threads completely, just like processes. At the other, it knows nothing about them. Both of these have problems. In the former, thread switching requires a kernel trap, with considerable overhead. In the latter, when one thread blocks on I/O or a page fault, all the threads are blocked, since the kernel does not even know threads exist. Some intermediate forms have been proposed but the last word has not yet been said [And91a].

5.4. Atomic Transactions

Atomic transactions are a powerful technique used in data base systems to maintain consistency in the face of concurrency. They are potentially applicable to many areas of distributed computing as well. The problem is that atomic transactions are extremely heavyweight and expensive. It remains to be seen if they are worth the price.

6. Conclusion

In summary, the main thing I have learned is that designing and implementing distributed operating systems is not as easy as it looks. It is essential to avoid complexity where possible, and resist the temptation to add more features whenever they rear their ugly heads. Putting in 4 ways to do *x*, 9 options for doing *y*, and 13 parameters for *z* is definitely not the way to go. The trick is to have simple conceptual models and not to try to be all things to all people. We should take our inspiration from UNIX Version 7, not OS/360. If every system designer had a great big sign with the old motto:

KISS – Keep It Simple, Stupid

we would go forward quickly.

Acknowledgements

I would like to thank Henri Bal, Frans Kaashoek, and Mark Wood for their helpful suggestions.

References

[And91a] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level," *Management of Parallelism – Proc. Thirteenth Symposium on Operating System Principles*, Pacific Grove, CA, pp. 95-109 (14-16 Oct. 1991).

[Bal91a] H. E. Bal and A. S. Tanenbaum, "Distributed Programming with Shared Data," *Computer Languages* **16**(2), pp. 129-146 (1991).

[Bal92a] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, pp. 190-205 (March 1992).

[Bir91a] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Comp. Syst.* **9**(3), pp. 272-314 (Aug. 1991).

[Bir84a] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems* **2**, pp. 39-59 (Feb. 1984).

[Car89a] N. Carriero and D. Gelernter, "Linda in Context," *Commun. ACM* **32**(4), pp. 444-458 (April 1989).

[Che88a] D. R. Cheriton, "The V Distributed System," *Commun. of the ACM* **31**, pp. 314-333 (March 1988).

[Gol90a] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an application program," *USENIX Summer Conf.*, Anaheim, CA, pp. 87-95 (June 1990).

[Han73a] P. Brinch Hansen, *Operating System Principles,* Prentice Hall, Englewood Cliffs, NJ (1973).

[Jul88a] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Trans. on Computer Systems* **6**, pp. 109-133 (Feb. 1988).

[Li89a] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Comp. Syst.* **7**(4), pp. 321-359 (Nov. 1989).

[Roz88a] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "CHORUS Distributed Operating Systems," *Computing Systems* **1** (Fall 1988).

[Sto81a] M. Stonebraker, “Operating System Support for Database Management,” *Commun. of the ACM* **24**, pp. 412-418 (July 1981).

[Tan90a] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum, “Experiences with the Amoeba Distributed Operating System,” *Commun. of the ACM* **33**, pp. 46-63 (Dec. 1990).

[Tan92a] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal, “Parallel Programming Using Shared Objects and Broadcasting,” *Computer* **25** (Aug. 1992).

[You87a] M. Young, A. Tevenian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, “Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System,” *Proc. Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, pp. 63-67 (Nov. 1987).

Towards a Scalable Kernel Architecture

Jörg Cordsen
Wolfgang Schröder–Preikschat

German National Research Center
for Computer Science
GMD FIRST
Berlin, Germany
{ jc | wosch }@first.gmd.de

Abstract

The paper starts with an examination of the notion *scalability*. Afterwards it discusses scalability issues in state-of-the-art kernel architectures, i.e., microkernels. It motivates the program family concept and object-orientation as the key to success in the design of an open microkernel architecture. To meet the special needs for massively parallel distributed memory machines, a kernel family is proposed by presenting the PEACE approach as case study.

> *"There is no tabula rasa.*
> *We are like skippers*
> *who have to rebuild their ship on the open sea*
> *without ever being able to dismantle it in a dock*
> *and reconstruct it of best parts."*
>
> (Otto Neurath)

1. Introduction

In the light of short-lived cycles of development it is of no surprise that nearly every hard- or software-system is decorated with the attribute of being *scalable*. Regarding kernel architectures, we hear about scalability just like multi-threading or virtual memory support. However, in difference to the latter concepts there is only a vague understanding of what scalability really means within a kernel architecture.

A lot of work has been done on scalable computing system architectures. In this context, scalability refers to an unlimited extensibility of computing systems, e.g., scaling up a 64-node system to a 512-node system without running into architectural bottlenecks. Usually, scalable computing systems are not based on shared memory, but on distributed memory, and often rely on simple processors with a limited general computation capability. In the area of parallel algorithms the notion of scalability is known as well. Despite different metrics, scala-

This work was supported by the Ministry of Research and Technology (BMFT) of the Federal Government, grant no. ITR 9002 2.

bility of a parallel algorithm on a parallel machine is a measure of its capability to utilize a larger number of available processors.

But how are we supposed to understand the notion of scalability concerning kernel architectures? Mainly, it is the task of a kernel to support the mapping of (parallel) algorithms on (parallel) computing systems. As mentioned, scalability issues are considered in the applications as well as in the underlying hardware architecture, but they are neglected by state-of-the-art kernel architectures. Up to now, kernel architectures rely on an *autonomous* design. Dependent on hardware properties a closed set of functionalities is provided, e.g., multi-tasking or virtual memory support. The kernel takes the same shape for all kinds of applications, thus, ignoring the concrete demands on an application.†

The autonomous design of a kernel enforces that the scalability is limited to a scale with exactly one mark. Accordingly, a scalable kernel must provide the whole set of functionality at all time. This spoils the scalability of a parallel algorithm, since the presence of unnecessary code-parts results in a reduced effectiveness. Such a drawback is unacceptable for a high-performance kernel architecture and suggests that autonomy and scalability are incompatible topics within a kernel architecture.

The paper starts with an examination of state-of-the-art operating systems, i.e., microkernel architectures. Afterwards it discusses the design of the parallel operating system PEACE [Ber92a]. It illustrates basic PEACE concepts and explains by what means support for massively parallel, distributed systems is given. Object-oriented mechanisms as well as strategies for dynamic system reconfiguration in PEACE are presented.

2. Microkernel Architectures

State-of-the-art operating systems are based on *microkernel* architectures. One of the most favorite systems representing this category is Mach [You87a]. A microkernel architecture is the attempt to decompose an operating system structure with the overall design rule to keep hold on those functions, whose processing on top of the kernel would be critical. The bulk of operating system services is accordingly executed in non-privileged user mode. Only a small set of services is subject to privileged supervisor mode execution. This organization supports a fault tolerant and application-oriented system structure. It hence seems to be the appropriate basis for all fields of application. This is true for distributed systems, but does not hold entirely for massively parallel distributed memory systems. Even the microkernel is too complex and, thus, too overhead-prone if the very hard performance requirements of massively parallel systems are taken into account. These requirements are to support a system-wide message startup time in the order of magnitude of 10 microseconds (using a 40 MIPS processor) [Mie89a].

Mach is a good example to clarify what the climax of microkernel complexity can be. The Mach 3.0 kernel is a software system of about 100,000 lines of C code (with comments excluded). Alone this large amount of source code is in contradiction to the common understanding of the notion **micro**kernel. As a comparison, the ancestor of the most

† Some very first and timid attempts has been done to break up the inflexibility. These efforts are related to the notion of *extensibility*.

successful operating system known to date was based on a kernel implementation of about 10,000 lines of C code (with comments included) [Lio77a]. Microkernel-based operating systems have been developed as counterpart to monolithic operating systems such as UNIX. However, this does not imply that the microkernel is no monolith too. The monolithic Mach microkernel is an order of magnitude more complex than the original monolithic UNIX kernel.

From the functional point of view, standard microkernels as used in Mach and Chorus [Roz88a] typically encompass interprocess communication, scheduling, security, process management, (virtual) memory management and exception handling. These are functions to support a multi-tasking mode of operation. They are necessary to process microkernel-based operating systems where services are provided by tasks being executed in user mode. Thus, even if an application does not require this operation mode, it must pay for it. This introduces significant overhead for those types of parallel applications which expect that tasks are mapped in one-to-one correspondence with the nodes of a massively parallel distributed memory system [Sch91a]. Multi-tasking is not free of charge and it is only needed if the one-to-one mapping can't be done for all the tasks.

It is true that, in terms of software engineering arguments, a microkernel must not be of minimal size [Gie90a]. However, are all the functions really mandatory fundamental building blocks? It obviously depends on the application field. Only if applications always demand these functions, either explicitly or implicitly, then a microkernel of this complexity is the right choice. A further software engineering argument is to have available for user applications only those functions which are really demanded. This is the only way to keep kernel complexity as small as possible, to have the chance to understand potential performance bottlenecks and to be really application-oriented [Par79a]. In addition, for security reasons complexity must be sacrificed as far as possible.

3. Approaching the Concept of Program Families

Forthcoming massively parallel systems are distributed memory architectures and will consist of several hundreds to thousands of autonomous processing nodes interconnected by a very high-speed network. A major challenge in operating system design for these parallel architectures is to elaborate a structure that reduces system bootstrap time, avoids bottlenecks in serving system calls, promotes fault tolerance, is dynamic alterable, and application-oriented. At the same time utmost highest communication performance must be provided. The solution to these problems is an approach in which an operating system is understood as a *family of program modules* [Par79a] and not as a monolithic "saurian" of more or less related components.

A parallel operating system has to provide only a *minimal subset of system functions*. Driven by the application, additional system/kernel services are to be considered as *minimal system extensions* [Par79a]. In order to optimally support applications, minimal system extensions then are loaded on demand, at the time initially requested.

This approach especially would mean that a microkernel is built by minimal extensions to a "**nano**kernel" and the minimal extensions are subject for incremental loading. Operating system scalability is generally improved. While the microkernel approach promotes a scalable system organization for distributed systems, the "nanokernel" does so for massively parallel systems too – it promotes a scalable kernel architecture. Hence, a "nanokernel" bridges the gap between massively parallel systems and distributed systems. It makes it feasible that design principles of distributed systems can be applied to massively parallel systems.

3.1. Minimal Basis

In the program family concept a *minimal subset of system functions* provides a common platform of fundamental abstractions. This minimal basis encapsulates solely *mechanisms* from which more enhanced system functions can be derived. It will be built by a consequent postponement of design decisions. Fundamental abstractions to make massively parallel systems work then are *processes* and *communication,* i.e., message passing.

Processes introduce scaling transparency and, thus, make the modeling of parallel applications independent from the actual number of processing nodes. Scaling transparency, however, is not only an issue in the programming of massively parallel systems [Gil91a], but improves also availability in the case of permanent nodes failures. Even if the application is tailored to the actual number of processing nodes, the crash of a single node could mean the premature end of application processing if the system does not support the migration of program activities onto still functioning nodes. For this purpose the system needs an instrument for the modeling of program activities, which is the process. Thus, a process serves as the common abstraction for both the application and the operating system.

Communication based on message passing is a must when processing nodes have direct physical access only to local memory. Access to non-local memory, i.e., to the local memories of other nodes, involves the execution of a network communication protocol. Because processes form the basic abstraction to model (user/system) activities, interprocess communication rather than internode communication is required.

Whether synchronous or asynchronous communication should be supported strongly depends on the process model and on its implementation [Beh88a]. Synchronous interprocess communication is the best choice in order to achieve the maximal utilization of network bandwidth. In contrast to asynchronous communication, intermediate buffering and, hence, additional overhead of message copying is not implied by the communication model; at most, it will be implied by the network hardware interface.

The decision for synchronous interprocess communication implies a potential loss of parallelism. This must be compensated by a process model which allows concurrent programming even on a single node, i.e., which supports a *multi-threaded address space.* Obviously, the implementation of this process model must lead to a process switch time which is significantly smaller than the copying and buffering overhead involved in asynchronous communication. Such a model is mechanized by *lightweight processes* [Hew77a] and being implemented as *featherweight processes* [Gil91a] to meet the performance requirements for parallel operating systems. The minimal basis then strongly promotes a processing model in which concurrency is not a side effect of communication, but is expressed explicitly by means of threads building a *team*. It implements a process execution and communication environment (PEACE) for parallel/distributed applications.

3.2. Minimal Extensions

A minimal basis which supports threading and communication already suffices to execute parallel programs. Moreover, it could be considered as the only operating system support residing on a processing node and being required by the application. In these dedicated applications the minimal basis is already the optimum. Additional functions are not used on the nodes and, hence, would only withhold system resources (such as memory space and processor time) from the given application.

It is the second important feature of the program family concept, that, dependent on the individual application, a stepwise functional enrichment of the minimal basis is performed by means of *minimal system extensions* only. These extensions encapsulate *mechanisms* and/or *strategies*. However, it might be the case that no system extensions are necessary at all. The application itself is always the best extension one can think of – it is the final extension anyway.

By adding minimal extensions, an operating system family is constructed bottom-up, whereby construction is controlled top-down: lower-level components are introduced only when required by higher-level components. This way, system functions for scheduling, security, process management, (virtual) memory management, exception handling, file handling, checkpointing and recovery are introduced. An open, application-oriented and evolutionary system organization is the consequence.

Understanding functional enrichment as an add-to in terms of components is only one aspect. It also includes *component replacement*. During the design phase a commitment on the minimal subset of system functions must be made. This includes the risk of stating wrong design decisions. One of the most important decisions is concerned with the identification of the proper operation mode, i.e., whether single-tasking or multi-tasking is to be supported.

The processing of parallel applications by a massively parallel machine always implies communication, hence the need for communication functions. The application might also call for a single or multi-threaded address space (i.e., task) on a node. Another application demands multi-tasking, which then is a functional enrichment of multi-threading. Should the minimal basis therefore support multi-tasking? If the design decision advocates multi-tasked nodes and tasks are mapped in one-to-one correspondence with the nodes, then a significant degradation of the message startup time will be the result [Sch91a]. Multi-tasking is not free of charge, even if not utilized by the application. A design decision to support solely multi-tasked nodes will handicap single-tasking applications and, thus, will not be conform with the idea of program families.

To overcome this problem, all applications must see the same external (abstract) interface of the minimal basis. What differs is the internal behavior, i.e., the concrete implementation. The external interface is mainly concerned with communication, while the internal behavior mainly dictates the process model and the operation mode of the node. With the minimal basis being an *abstract data type* [Lis74a] a number of implementations of the same interface can coexist. This makes the minimal basis exchangeable at least from the design point of view.

Flexibility is maintained although the minimal subset of system functions must have been fixed early in the software design process.

4. The Role of Object Orientation

Applying the family concept in the software design process leads to a highly modular structure. New *system features* are added to a given subset of system functions. One instrument to implement a program family is to apply the abstract data type mechanism. An instance of an abstract data type is implemented by a module. System functions then are represented by the operations which are defined by the module interface specification. The entire system ends up with a multi-level hierarchy of a multitude of program modules, with a well-defined *uses relation* [Par79a] between the modules to associate them to levels in the hierarchical system.

A problem with the module-oriented approach is the potential for a large number of redundant code and data portions in those cases where different implementations of the same module interface coexist [Cam87a]. That the redundant portions are not encapsulated by an abstract data type on its own, i.e., extracted and implemented by a separate "service module" and then being properly used by the instances is due to at least two facts: *genericity* and *efficiency*. One often examines that the new service module must be capable to deal with objects of different type, whereby the type is defined by those instances which will use the new module: the new module is generic. Having strongly followed the pattern of abstract data types, the additional module boundary often implies an increase in runtime overhead due to additional procedure calls for operation invocation: the new module introduces a potential performance bottleneck.

The feasibility of this kind of abstract data typing depends on the power of the programming language to implement generic module interfaces and on the function inlining capabilities of the compiler. If a parallel operating system is required to guarantee a message startup time in the order of magnitude of 10 microseconds (assuming a 40 MIPS processor), any increase of runtime overhead caused by either of programming paradigm, programming language or compiler is not acceptable.[†]

The much more promising approach in the design and development of operating system families, therefore, is to apply *object orientation* [Weg87a]. In other words, object orientation is the natural choice to build program families [Cor91a]. The buzzword is *inheritance* [Hal87a] to avoid large portions of different module versions to be identical. Functional enrichment defines new family members, which always inherit properties of existing family members. The new family member is built by at least one new specialized class by derivation from one or more base classes (*single/multiple inheritance*). This implies the re-usage of existing implementations on a sharing basis, meaning that code/data redundancy will never appear in a clean object-oriented design.

[†] The first PEACE kernel prototype for a distributed memory parallel computer was implemented in Modula-2. Performance was not acceptable. A transformation into C and non-optimized compilation lead to a negligible performance improvement. Applying the keyword "*register*" at meaningful places and with optimized compilation, a 40 percentage performance increase was obtained [Sch88a]. Register optimization lead to fewer memory traffic, which is significant if the processor executes 3 (4) wait states on each read (write) memory access.

In *class-based object orientation* [Bla89a], the class definition includes the implementation of the methods defined on objects of that class. This makes function inlining straightforward and, hence, reduces the procedure call overhead to an absolute minimum. An example is C++ [Str86a], which also supports *abstract data type based object orientation*. Note, the major problem with identical portions of different module versions primarily is not wasted memory space, which function inlining implies too. Above all, it is a software maintenance problem, which (class-based) object orientation with or without function inlining helps to avoid.

There is another feature of object orientation which is of importance for the implementation of a family of operating systems. This feature is known as *polymorphism*. A base class specifies the operations which are defined on objects of that class. In the course of inheritance, a derived class may specify either the same operations again or a subset only. These redefined operations usually show for a different, more specialized implementation. The external class interface is still described by the same base class, while different implementations of the same interface can coexist by means of inheritance and dynamical binding. Polymorphism strongly supports the design and implementation of replaceable components. Featuring the proper derived class is dynamic and works transparently to the instance applying the base class only.

5. A Parallel Operating System

The PEACE family concept distinguishes between a macroscopical view to identify the overall system architecture and a microscopical view to define the minimal subset of system functions that must be present on each node. The former aspect deals with distribution and the latter aspect deals with performance.

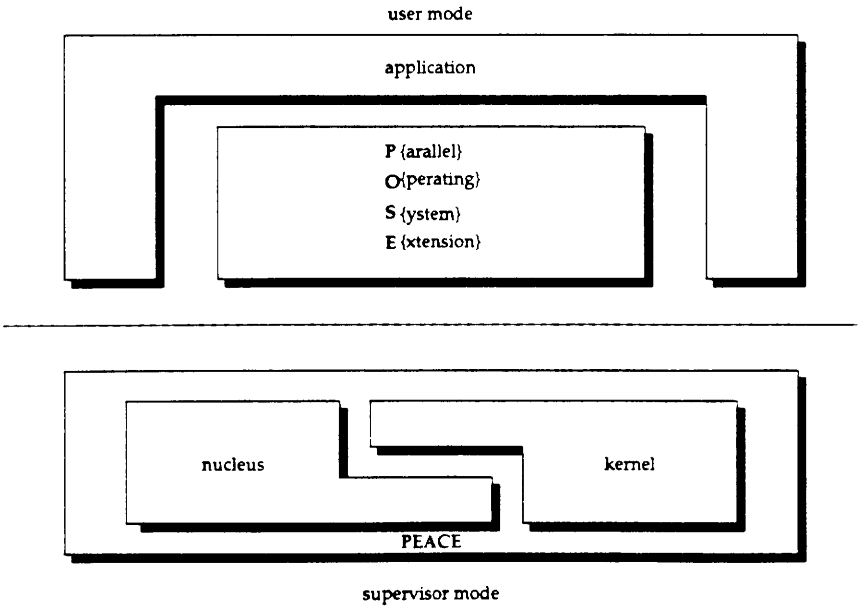


Figure 1: *Building Blocks*

5.1. Macroscopical View

A member of the PEACE parallel operating system family is constituted by three major building blocks: *nucleus, kernel,* and *POSE* (Figure 1). The nucleus implements system-wide interprocess communication and provides a runtime executive for the processing of threads. The PEACE nucleus is part of the kernel domain, with the kernel being a multi-threaded team that encapsulates minimal nucleus extensions. These extensions implement device abstraction, dynamic creation and destruction of process objects and the association of process objects with address spaces. Application-oriented services such as process and memory management, file handling, i/o, are performed by POSE, the parallel operating system extension of PEACE. It is built by a multitude of active objects (i.e., servers) distributed over the nodes of the parallel machine.

The dividing line between user and supervisor mode is a logical boundary only. It depends on the concrete representation of the interactions specified by the functional hierarchy (and of the processor architecture) whether this boundary is physically present. The functional hierarchy of these components (Figure 2) defines the way decentralization works with PEACE. All components are encapsulated by (active/passive) objects. An object invocation scheme must therefore be used to ask for service execution.

Nucleus services are made available to the application via *nearby object invocation* (NOI). The logical design assumes a separation of the nucleus from the application (and POSE), which calls for the potential of address space isolation and of traps to invoke the nucleus. This is the place where *cross domain calls* may happen. The kernel shares with the nucleus the same address space and, hence, performs *local object invocation* (LOI) to request nucleus services. Kernel services are made available via *remote object invocation* (ROI) [Nol92a], an object-bound mechanism similar to the *remote procedure call* paradigm [Nel82a]. Services of POSE are requested via LOI and ROI. Here, LOI is used to interact with the POSE runtime system library and ROI is used to interact with the POSE active objects.

From the design point of view neither the kernel nor POSE need to be present on each node, but the nucleus. In a concrete configuration, the

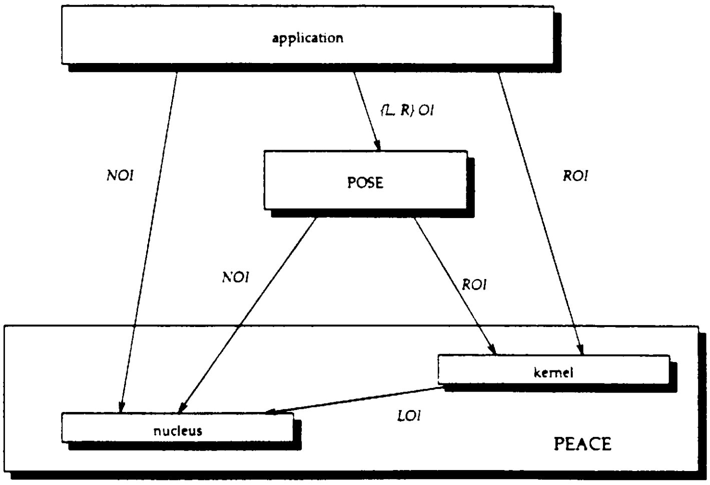


Figure 2: *Functional Hierarchy*

majority of the nodes of a massively parallel machine is equipped with the nucleus only. Some nodes are supported by the kernel and a few nodes are allocated for POSE. All nodes can be used for application processing, but they are not all obliged to be shared between user tasks and system tasks.

It is important to understand that the functional hierarchy of the three building blocks expresses the logical design of PEACE only, and not necessarily the physical representation. The building blocks are designed with respect to the various schemes of object invocation as shown in Figure 2. However, it depends on the actual operating system family member whether these schemes become effective as specified by the design or can be replaced by a more simple alternative. For example, although the functional hierarchy assumes NOI for the interaction between application (POSE) and nucleus, the LOI scheme is used for those members of the nucleus family which place their focus on performance and support single-tasking mode of operation only.

5.2. Microscopical View

A process execution and communication environment forms the minimal subset of system functions required by massively parallel systems. This minimal basis of PEACE is a compromise between *transparency* and *efficiency*. For different applications there are different implementations of the same interface of the minimal basis, hiding all the internals. This transparency is to the convenience of the application programmer.

The minimal basis is defined as a *family of functional dedicated units* with a single external interface – all family members inherit the same

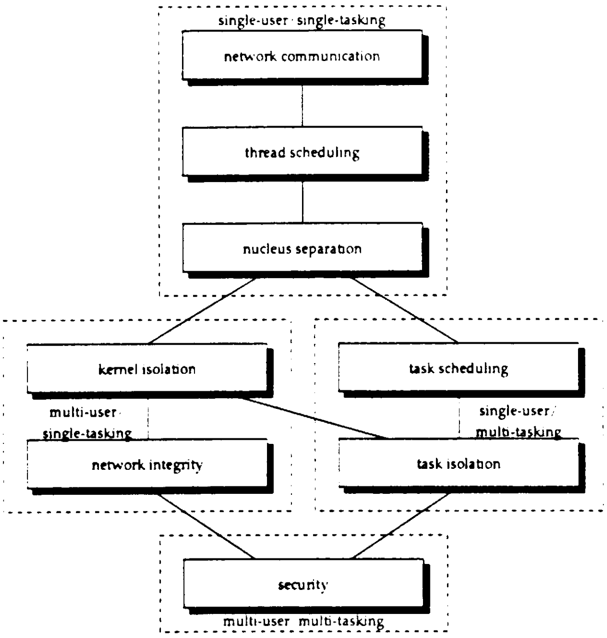


Figure 3: *Nucleus Family Tree*

[†] In reality there are several base classes which represent the external view of the minimal basis. These classes are *ticket* (delivery of system-wide unique communication endpoint identifiers), *notice* (intra-team thread synchronization with empty messages), *parcel* (packet-based synchronous, system-wide inter-process :ommunication), and *region* (segment-based asynchronous, system-wide interteam communication). They stand for horizontal independent functional units of the nucleus.

base class that specifies the unit interface.† This minimal basis is represented by the *PEACE nucleus*, i.e., a nucleus family. The nucleus family implements four different operation modes (Figure 3). Each operation mode is represented by a subfamily, with several implementations of the same nucleus abstract data type. Presently, eight nucleus family members are distinguished.

The entire family tree shows different nucleus versions, with the root (top) being the most simple and the leaf (bottom) being the most complex instance. As complexity increases, performance drops.

The nucleus family defines a pool of functional units of more or less complexity, likewise offering lower or higher performance. Dependent on application requirements and on the actual utilization of the parallel machine, the proper nucleus version comes into play. Whether a nucleus instance is being integrated statically or dynamically is not of primary importance from the design point of view. First the complete family structure must be known and then the decision can be made to implement the family as a dynamically alterable system.

5.2.1. Single-User / Single-Tasking

There are three different nucleus instances supporting single-user/single-tasking mode of operation. The two most efficient instances provide *network communication* and *thread scheduling* on a library basis. Thus, these nucleus instances are part of the address space of the user/system process. This implies that no overhead-prone address space boundaries must be crossed to invoke the nucleus.

PEACE only implements synchronous interprocess communication. Concurrency then is to be modeled explicitly by the application using multiple threads of control. The threading instance (i.e., thread scheduling) is the corresponding mechanization. Because of the nonexistent address space boundary, this nucleus is extremely lightweight and, thus, supports the notion of *featherweight processes*. Featherweight processes are a specialized implementation of lightweight processes. They are the purest form in PEACE to represent units of execution, without consideration of any protection and security measure.

The threading instance combined with the need for kernel code separation makes nucleus calls more heavyweight. Now, traps are used to invoke the nucleus. This implies very small stub routines to marshal and unmarshal nucleus service requests similar to the remote procedure call paradigm. However, instead of passing a message over a narrow channel, a local trap is to be performed. A *featherweight remote procedure call* (i.e., NOI) is executed to activate the nucleus. Solely the gap implied by the trap interface is bridged. Kernel code separation is supported, but not memory protection. As a consequence, the passing of complex data structures between the nucleus and higher-level entities is straightforward and involves no programming of address space protection hardware.

The functional enrichment introduced by *nucleus separation* enables dynamic component replacement by a third party. Higher-level entities are physically uncoupled from nucleus code. Because each nucleus instance is an abstract data type, these entities are also logically uncoupled from nucleus data. The basic mechanism to switch between different nucleus instances on the fly is to exchange trap vector entries.

5.2.2. Multi-User / Single-Tasking

In a distributed memory parallel machine, multi-user mode of operation is feasible even if only a single task is mapped onto each node. The entire multi-node machine can be allocated to different users at the same time. Obviously, this does not require local ("on-board") security measures to protect the tasks from each other, but it requires to protect the network interface from unauthorized access. By direct network access the user task could be able to intrude the network and, thus, tasks of different user applications.

In order to provide a multi-user function, the nucleus must be completely isolated. Memory protection is to be introduced, leading to a new instance: *kernel isolation*. Because the nucleus is part of the kernel domain, applying memory protection to the nucleus also implies the isolation of parts of the kernel address space. Concerning nucleus separation, no additional overhead is introduced. However, the isolated nucleus address space makes the passing of complex data structures heavyweight. It mainly depends on the address space protection hardware how crucial the additional overhead really is. Anyway, the increase of nucleus functionality is encompassed by the potential of communication performance loss.

On each node, *network integrity* must be guaranteed, but not necessarily the integrity of user task address spaces. This leads to the introduction of communication firewalls between different user applications. Each user application builds a unique *communication domain*. The same holds for the set of system processes constituting the operating system. Within the same domain communication is unlimited. In order to invoke system services, application processes must communicate with system processes. Consequently, different communication domains must overlap to let communication succeed. Thus, communication security does not mean complete isolation, solely, but also controlled access.

A capability-based approach is used in PEACE for this purpose. This approach grants object access only if a thread (i.e., subject) is in the possession of that object or one of its proxies. An object must be created before it can be used. It is then the autonomous decision of the object creator to make the object globally accessible. The access domain of an object may be extended by the object creator by exporting a *proxy object* [Nol92a]. Via the proxy global (i.e., network-wide) object access is then feasible.

5.2.3. Single-User / Multi-Tasking

The first step towards multi-tasking support is to introduce *task scheduling*. In PEACE, a task maybe multi-threaded, which implies only lightweight scheduling. In order to schedule tasks, a second scheduling level is implemented. This level knows the ***bundle*** as scheduling unit, which consists of one or more threads. A single threads bundle always is executed by one processor, with non-preemptive scheduling of the threads of the same bundle. Preemptive scheduling is between bundles only, and so is shared-memory multiprocessor scheduling with the different bundles being executed by different processors. A task then may consist of several bundles to take advantage of preemption and of the shared-memory processor architecture. The result is a slightly more expensive scheduler.

At this stage, multi-tasking can be supp^rted even if *task isolation* by means of private address spaces is not provided. A private address space serves for two basic purposes. On the one hand it implements memory protection, isolating programs from each other. On the other hand it defines a logical address space for program execution, enabling code/data relocation at runtime. Being relocatable is also a property of *position independent* code, which then needs to be generated by a compiler. In addition, the use of secure programming languages supports program isolation without the necessity of address space protection hardware. Therefore, the minimal basis to support multi-tasking is task scheduling. Task isolation is the minimal extension of task scheduling. It is used to generally improve system availability and in those cases where neither the programming language nor the compiler supports the nucleus.

5.2.4. Multi-User / Multi-Tasking

The fourth operation mode being supported by the nucleus family is the natural consequence of the two modes discussed before. There is little more of functionality to add. Global multi-user mode of operation is made feasible by enforcing network integrity, whilst local multi-user function is directly supported by task isolation. The nucleus then provides general *security* measures, with completely isolating different (user/system) domains from each other.

6. Adaptive Operating System Architecture

The operating system building block of PEACE is mainly represented by POSE, which implements a family of parallel operating systems. POSE services are application-oriented extensions of the PEACE minimal basis, i.e., of the nucleus and the kernel. These services are provided by teams of lightweight processes and, usually, are executed in non-privileged user mode. Since the representation of the functional hierarchy of PEACE enables an almost arbitrarily decentralization of the building blocks, this does not enforce a microkernel approach and, thus, the need for multi-tasking on a single node.

6.1. Active Objects

Distributed memory architectures at least call for an object-based system design. In POSE, system services are represented by active objects, i.e., teams of lightweight processes implement system functions such as process management or file handling. Consequently, requesting the execution of a system service requires to send a message to some process. A typical client-server relation is established. POSE then consists of a multitude of cooperating teams distributed over the nodes. These teams are called *manager.*

The consequent usage of teams for system service encapsulation has several benefits. It provides a natural basis for building application-oriented operating systems. System services need only be present if they are required, meaning that the corresponding teams are created and loaded on-demand. Especially in the case of massively parallel systems, it is not required that user teams share the same node with system teams. This significantly reduces global system initialization time and makes the parallel system to appear as a *processor bank* whose purpose is to exclusively execute user applications.

Following the team structuring approach, the notion of a system call (service invocation) is slightly different from the traditional viewpoint

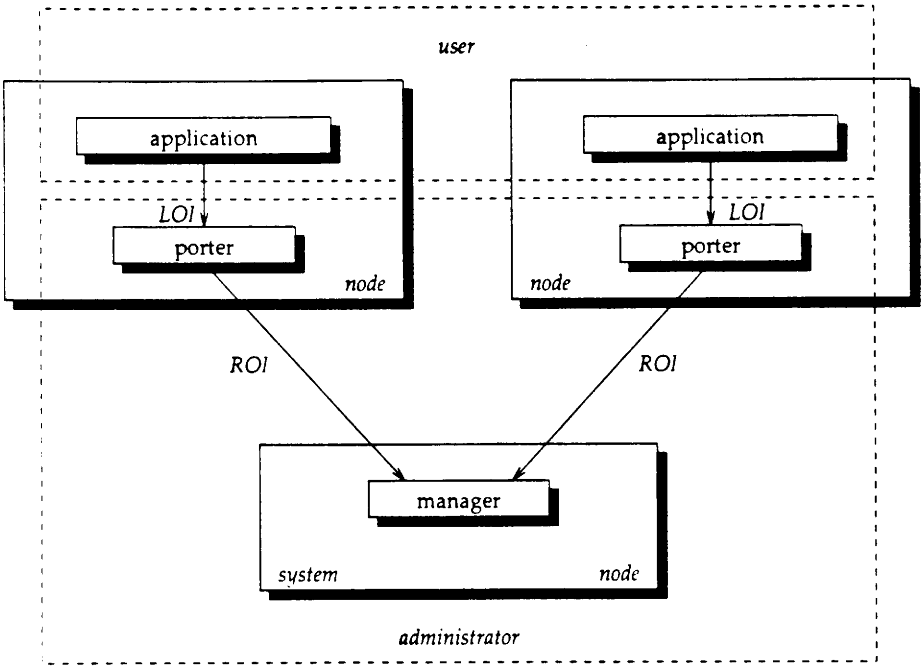


Figure 4: *System Access*

of a trap. A system call must be requested by means of message passing, distinguishing between local and remote operation. In order to hide all these properties from both the service user (client) and the service provider (server), a PEACE system call in general takes the form of *remote object invocation* [Nol92a].

6.2. Functional Replication

There are several reasons for service replication in massively parallel, distributed systems. One aspect is to avoid the presence of bottlenecks when a manager tends to be overloaded by too many service requests. Another case is redundancy for fault tolerant purposes. Furthermore, there might be replicated I/O hardware units such as disks. In all these cases, managers are replicated because of performance, availability, or architectural reasons.

This leads to the concept of distributed managers. The set of managers of the same type (i.e., class) constitutes a PEACE *administrator*. For scalability reasons, processes should not be aware of using replicated services. Rather they interact with an administrator. In this situation, the administrator has to keep track of which manager is to be selected for service execution. For these reasons, the PEACE administrator concept is not only supported by a number of managers, but also by a related *porter* that directs requests to the proper manager and, thus, serves as an administrator interface (Figure 4).

The porter takes the form of a library; it is part of the team address space of the service-requesting process. Dependent on the type of service, the porter may also encapsule private threads. For example, using porter threads enables service-related exception handling on a message-passing basis.

6.3. Third Party Configuration

Above all, a parallel operating system must be designed such that the amount of system software which is to be executed by each node can be reduced to an absolute minimum; otherwise, system bootstrapping becomes a nightmare. For this reason, POSE distinguishes between site-dependent and site-independent managers.

A site-dependent manager typically provides low-level and hardware-related services. For example, the disk manager encapsules device dependent functions and, thus, must reside in a node that has a disk attached. It is site-dependent, whereas the file manager, which uses the disk manager, may reside elsewhere and is considered as site-independent. Another example of a site-dependent manager is the kernel team. If dynamic process management is required on a node, a kernel must be present on that node to construct/destruct process objects. A process manager, however, is site-independent. It may reside on any other node and may also be responsible for the management of several nodes.

The property of being configurable is absolutely necessary to meet the needs for massively parallel systems. Except in the case of site-dependent managers, a *third party* is able to establish PEACE (i.e., POSE) configurations based on the individual needs of parallel/distributed applications. The configuration decision then will be made with respect to either performance, protection, or hardware availability.

6.4. Incremental Loading

The basic idea in PEACE is to perform *on-demand loading* of system services [Sch91b]. That is, system services are only loaded at the time when they are really needed. On-demand loading of services at run-time can be accomplished either explicitly, by using dedicated system calls, or implicitly, during service invocation if the corresponding manager does not yet exist. The latter approach requires close cooperation with the ROI layer.

If service addressing fails, a *server fault* is raised, similar to a page fault in virtual memory systems. Handling a server fault results in the loading of the requested service, i.e., the proper manager team is created and given a program for execution.† Entity (or server) faults are propagated to a system team called *plumber*. Basically, this means that, once having determined that the entity is not yet available, a stub routine requests entity loading by instructing the plumber accordingly (Figure 5). The stub passes the load request to the plumber which then takes charge of all activities related to the loading of the specified entity. Note, the porter takes the form of a system library and belongs to the team of the thread that caused the entity fault. As long as fault handling is in progress, on behalf of the porter the thread is blocked on the plumber, waiting for loading to be completed.

The plumber maps *entity names* onto file names, i.e., associates with entities a file that describes the team image to be loaded. With each entity name several attributes are stored. For example, the file may describe either a plain team image or a complete boot image. In case of site-dependent managers, the node addresses are stored with the

† Any kind of service that can be loaded on demand is in no way distinguished from an application process. Thus, on-demand loading works for both user and system applications. The general term *entity* is used for teams that belong to either of these application classes. In this sense, the server fault actually means an *entity fault*.

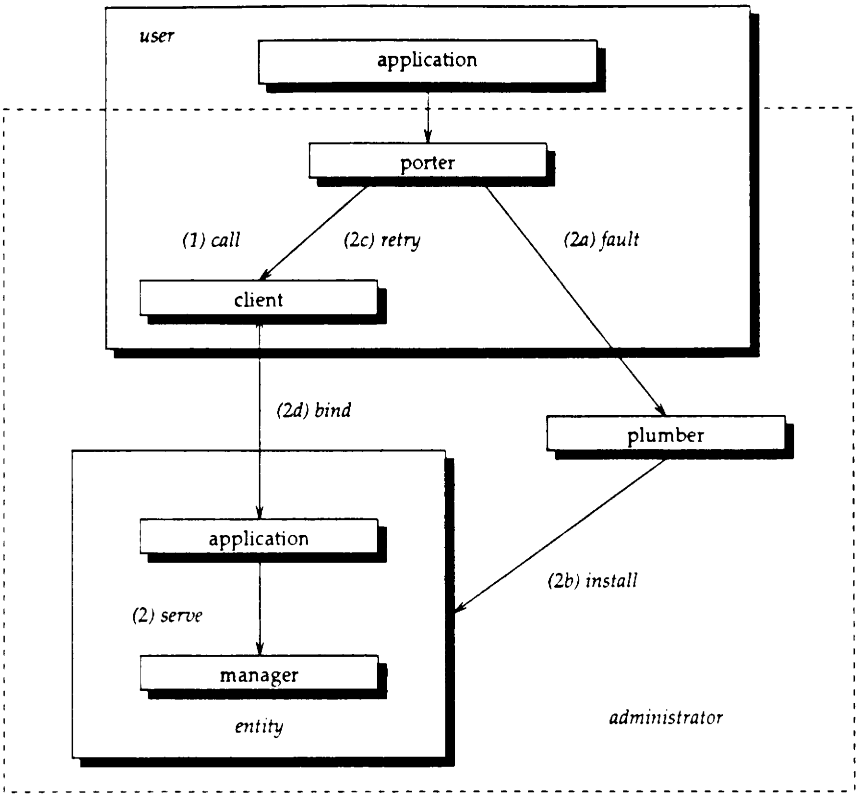


Figure 5: *Entity Fault Handling*

entity name. A distinction between the single-tasking or multi-tasking mode of operation for the entity is also made.

In PEACE, the minimal basis for dynamic restructuring requires no complex memory management functions. A maxim was that even with a single-tasking nucleus instance, which is not based on address space protection and, therefore, encompasses no memory management functions, dynamic restructuring of the node of that nucleus must be feasible yet. This node, e.g., must be given multi-tasking capability by exchanging the kernel and then allocating tasks. If the PEACE kernel comes up, and so the nucleus, it always assumes non-protected address spaces. The capability to protect address spaces is the kernel taught by the *memory manager*, a site-independent system team which is loaded on demand.

7. Related Works

The PEACE approach goes beyond that what is presently intended by state-of-the-art microkernel designs, it defines a microkernel family. In systems such as Mach [You87a] and Chorus [Roz88a], the microkernel is a fairly complex component, used to support the implementation of operating system services and the processing of distributed applications. As in PEACE, a Chorus operating system is considered as a member of a family of functional units, with a unit being represented by a (multi-threaded) system server process, i.e., an active object. PEACE also applies the family concept to structure the kernel and not only an operating system. This results in a (multi-threaded) kernel ir..plementation with a distinguished component, the nucleus, providing a common process execution and communication environment. The

Chorus microkernel (also termed nucleus) is the only choice applications have. In PEACE, the nucleus family presents an assortment of up to eight different members.

Ra[Aub88a] is a minimal kernel for the Clouds distributed operating system [Das88a]. The *Ra* kernel is designed to support the implementation of large scalable object-based systems. *Ra* is a fairly complex minimal kernel too, implementing segment-based virtual memory management and short term scheduling. At best, *Ra* can be compared to the PEACE nucleus instance that provides task isolation, which is one of the most complex nucleus family members of all.

Clouds distinguishes between objects and threads, i.e., it is structured by passive objects. The rationale for this approach is to avoid performance penalties caused by the virtually more complex code of multi-threaded server implementations. That multi-threaded servers are more complex is only true for completely hand-coded implementations, but not for implementations that are supported by a class-based stub generator as in PEACE [Nol92a]. Anyway, reducing server code complexity by downward migration of functions into the minimal kernel as followed with *Ra* is not the ultima ratio. It makes the minimal basis more complex and, thus, more overhead-prone.

The system which comes very close to PEACE is *Choices*[Cam87a]. Many ideas found in *Choices* are present in PEACE, and vice versa. This is because both systems share the same fundamental, classic idea of a family of operating systems [Hab76a]. They extend this idea into object-oriented, distributed/parallel environments. As *Choices,* PEACE is a class-hierarchical system. By means of the nucleus family, PEACE further distinguishes between a number of operation modes a node of a massively parallel system is exposed to. It is exactly this feature which becomes more and more important for forthcoming parallel operating systems.

Dynamic restructuring in PEACE is related to active and passive objects. Introducing active objects is straightforward and based on services to create and destroy teams of lightweight processes. Exchanging passive objects is limited to the nucleus. This is in contrast to Clouds, e.g., where arbitrary passive objects may be dynamically introduced. For this purpose Clouds relies on the segment-based virtual memory management service of the *Ra* kernel. These constraints are not given with PEACE in general. There are some PEACE family members implementing segment-based virtual memory management; there are others not being dependent on the presence of address space protection hardware and supporting dynamic restructuring yet.

8. Conclusion

This paper described rationale and concepts for the design of scalable operating systems for massively parallel systems. The program family concept combines a number of solutions to different application requirements. This concept promotes not only customized operating systems from the application point of view (top-down customization), but also from the hardware architecture point of view (bottom-up customization).

A distinction between operating system family and a nucleus family must be made to meet the performance requirements of forthcoming massively parallel systems. In the former case, the family is built by a number of site-independent functional units representing typical operating system services. In the latter case, a platform for both kernel construction and application processing is provided. A member of the nucleus family must be an abstract data type to allow a number of different implementations to coexist. The nucleus family takes the form of an assembly camp, but not the single nucleus implementation. From this assembly camp the proper solution is selected to optimally support a given application. This way, the PEACE approach provides a scalable, i.e., WYNIWYG-architecture (*What You Need Is What You Get*) for both the kernel and the operating system. A single solution always is a bad compromise if utmost highest communication performance must be guaranteed and a large spectrum of applications must be supported.

Approaching the family concept as exemplified with PEACE makes microkernels appear as extensions to a minimal basis. That is, PEACE provides a framework not only to build upward scalable but also downward scalable kernel architectures, an important property of parallel operating systems. The microkernel as being understood to date is merely a member of the PEACE family. To keep things right in mind: the functionality of state-of-the-art microkernel architectures facilitate scalability but at the same time forms an essential scalability handicap in case of unnecessary functionality is provided. Thus, the PEACE family design bridges the gap between distributed systems and massively parallel systems which are based on distributed memory architectures.

The family is designed, constructed and implemented following the paradigm of object orientation [Cor91a]. Classes implement system features and inheritance (i.e., subclassing) is used to derive new features or specializations of existing ones. First experiences with objective PEACE show that object orientation is superior to non-object oriented approaches. This is true for aspects such as maintainability, extensibility and performance of the resulting operating system. It is indeed a myth that object orientation makes the implementation of very high-performance operating systems impossible. Rather, it is true that object orientation is the only chance to build high-performance systems while maintaining a clean and evolutionary system structure.

The object-oriented paradigm in design and implementation of a distributed/parallel operating systems is widely accepted but, with the exception of a few operating systems, e.g., Choices and PEACE, not applied in correspondence to Wegners definition [Weg87a]. A general problem for commercial systems like Mach or Chorus is how to organize a complete redesign of their operating system. There are plans going into this direction and which shows that the system's investigators are encouraged of the object-oriented paradigm. Unfortunately, as pointed out by *Neurath,* they can't enjoy all the opportunities object-orientation offers because of their market constraints.

References

[Aub88a] J. M. B. Auban, P. W. Hutto, M. Yousef, A. Khalidi, M. Ahamad, W. F. Appelbe, P. Dasgupta, R. J. LeBlanc, and U. Ramachandram, *The Architecture of Ra: A Kernel for CLOUDS,* Georgia Institute of Technology, Technical Report GIT-ICS-88/25 (1988).

[Beh88a] P. M. Behr, W. K. Giloi, and W. Schröder, "Synchronous versus Asynchronous Communication in High Performance Multicomputer Systems," *IFIP Working Conference 5*, Stanford (August 22-26 1988).

[Ber92a] R. Berg, J. Cordsen, J. Heuer, J. Nolte, B. Oestmann, M. Sander, H. Schmidt, F. Schön, and W. Schröder-Preikschat, "The PEACE Family of Distributed Operating Systems," *Technical Report (Arbeitspapiere der GMD 646)*, Berlin, Germany, Gesellschaft für Mathematik und Datenverarbeitung mbH (May 1992).

[Bla89a] G. S. Blair, J. J. Gallagher, and J. Malik, "Genericity vs Inheritance vs Delegation vs Conformance vs ...," *Journal of Object-Oriented Programming* **2**(3), pp. 11-17 (1989).

[Cam87a] R. Campbell, G. Johnston, and V. Russo, "Choices (Class Hierachical Open Interface for Custom Embedded Systems)," *ACM Operating Systems Review* **21**(3), pp. 9-17 (1987).

[Cor91a] J. Cordsen and W. Schröder-Preikschat, "Object-Oriented Operating Systems Design and the Revival of Program Families," *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, Palo Alto, California, pp. 24-28, IEEE (October 17-18 1991).

[Das88a] P. Dasgupta, R. J. LeBlanc, Jr., and W. F. Appelbe, "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work," *Proceedings of the 8th International Conference on Distributed Computer Systems*, San Jose, CA, pp. 2-9, IEEE (June 1988).

[Gie90a] M. Gien, "Micro-Kernel Architecture – Key to Modern Operating System Design," *Technical Report, CS/TR-90-42.1*, Paris, Chorus systemes (1990).

[Gil91a] W. K. Giloi and W. Schröder-Preikschat, "Programming Models for Massively Parallel Systems," *International Symposium on New Information Processing Technologies 1991*, Tokyo, Japan (1991).

[Hab76a] A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," *Communication of the ACM* **19**(5), pp. 266-272 (1976).

[Hal87a] D. C. Halbert and P. D. O'Brien, "Using Types and Inheritance in Object-Oriented Languages," *IEEE Software* **9**, pp. 71-79 (1987).

[Hew77a] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence* **8**, pp. 323-364 (1977).

[Lio77a] J. Lions, *UNIX Operating System Source Code Level Six,* Department of Computer Science, The University of New South Waleles (1977). Second Printing

[Lis74a] B. H. Liskov and S. K. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices* **9**(4) (1974).

[Mie89a] H. Mierendorff, "Bounds of the Startup Time for the GENESIS Node," *Technical Report, ESPRIT Project No. 2447*, GMD F2.G1, Bonn (1989).

[Nel82a] B. J. Nelson, *Remote Procedure Call,* Carnegie-Mellon University (1982). Report CMU-81-119

[Nol92a] J. Nolte, "Language Level Support for Remote Object Invocation," *Technical Report (Arbeitspapiere der GMD 654)*, Berlin, Germany, Gesellschaft für Mathematik und Datenverarbeitung mbH (1992).

[Par79a] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering* **SE-5**(2), pp. 128-138, IEEE (1979).

[Roz88a] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "CHORUS Distributed Operating Systems," *Computing Systems Journal* **1**(4), pp. 305-370, University of California Press and Usenix Association, Also as Technial Report CS/TR-88-7.9, Chorus systemes, Paris (1988).

[Sch91b] H. Schmidt, "Making PEACE a Dynamic Alterable System," *Lecture Notes in Computer Science* **487**, pp. 422-431, Springer-Verlag, Proceedings of the 2nd European Distributed Memory Computing Conference (April 1991).

[Sch91a] W. Schröder-Preikschat, "Overcoming the Startup Time Problem in Distributed Memory Architectures," *Proceedings of the 24th Hawaii International Conference on System Sciences* **1**, pp. 551-559 (January 1991).

[Sch88a] W. Schröder, "A Distributed Process Execution and Communication Environment for High-Performance Application Systems," *Lecture Notes in Computer Science* **309**, pp. 162-188, Springer-Verlag, Proceedings of the International Workschop on "Experiences with Distributed Systems", Kaiserslautern, Germany, September 28-30, 1987 (1988).

[Str86a] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company (1986).

[Weg87a] P. Wegner, "Dimensions of Object-Based Language Design," *Special Issue of SIGPLAN Notices* **22**, pp. 88-97 (Oct. 1987).

[You87a] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *ACM Operating Systems Review* **21**(5), pp. 63-76, Proceedings of the Eleventh ACM Symposium on Operating System Principles, Austin, Texas (1987).

Microkernel Support for Checkpointing

Rong Chen Tony P. Ng

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, USA
rchen@cs.uiuc.edu

Abstract

In microkernel paradigm, a kernel only provides basic computation mechanisms, and conventional operating system environments (policies) are implemented as system servers, e.g., the Mach UNIX server. In this paper, we present the checkpointing mechanisms that we add to the Mach 3.0 microkernel and discuss a fault-tolerant environment that we build on top of the modified kernel.

1. Introduction

A distributed system is called *reliable* or *fault-tolerant* if it has the capability to allow user-level processes to survive partial system failures, including hard and transient failures. A process that can withstand system failures is called a *resilient* process. The choice as to whether a particular process is to be executed as a resilient process is made by the user when that process is started. We refer to the saved state of a process as a *checkpoint* and the act of taking a checkpoint as *checkpointing*.

The basic objective of checkpointing is that when failures occur a job can resume its computation without losing all its previous work. In the 60's, when computers were stand-alone and programs were isolated from each other, checkpointing meant copying intermediate data to secondary storage periodically [Jas69a]. Today a computation is often carried out on a multi-processor computer or in a multi-computer network environment, and checkpointing has to deal with process synchronization and message logging in addition to backing up memory images [Lee90a].

Message logging is a variation of checkpointing. Instead of taking a full fledged checkpoint every so often, a process saves messages either on its stable storage or on a different machine. In case a failure occurs, the process will resume from its last checkpoint and then replay the past messages it had received in order to reconstruct its state consistent with the rest of the system. The duplicate messages produced by the backup process are discarded.

Implementations of software checkpointing algorithms are constrained by the underlying operating system designs. For example, the UNIX process' *pid* is an index of the kernel process table on a specific machine, the same pid might not be available when the process is restarted on a different machine. Therefore, inter-process communications based on pid may not be possible unless the UNIX kernel has to be modified substantially. Similar problems occur for UNIX socket numbers and in-kernel file systems. Due to implementation limitations and complexity, few software fault-tolerant systems have justified their performance on real machines.

During the past few years, the *microkernel* [Use92a] has become a new area in operating system research. In microkernel paradigm, a kernel only provides basic computation mechanisms, and conventional operating system environments (policies) are implemented as system servers, e.g., the Mach UNIX server. Since microkernels do not have any machine dependent information tied to processes, restarting a process from one of its past checkpoints on another machine is both feasible and relatively easy to implement. In this paper, we present the checkpointing mechanisms that we add to the Mach 3.0 microkernel and discuss a fault-tolerant environment that we build on top of the modified kernel. The technique is also applicable to other microkernels, such as Chorus or Amoeba.

In our computation model, a process can be in only one state at a time. The state of a process may change after the process receives a message. State changes are irreversible and a checkpoint is performed whenever a state change occurs. The states of all communicating processes at a given time are called *consistent states*, if they are reachable through normal executions.

We assume all processes are fail-stop, i.e., a process crash results only in the loss of all state information acquired since the last checkpoint. The algorithm that we implement sustains only single machine failures. The kernel checkpointing mechanisms, however, should be general enough to support different checkpointing algorithms.

2. Motivation

We make a few observations about fault-tolerant computation in a distributed environment.

- In a message passing system, some messages are more crucial than others, e.g., missing a read request might not be as critical as losing a write buffer for a file server. That is, checkpointing at every message or logging all messages between checkpoints may not be necessary.
- We only need to checkpoint a sender process whenever it sends a message that could cause a state change at the receiver. We call such a message a *modify* message (see also [Lin90a]).
- By letting senders, rather than receivers, keep most state information, the number of server state changes would be reduced, and so would be the number of checkpoints. An example is the design of stateless file servers [San85a].
- Data caching in a distributed environment improves both efficiency and reliability, as demonstrated by the Coda file system developed at CMU [Sat90a]. The client may flush back the pages again if its server crashes [Joh87a].
- Application-dependent information may further reduce the cost of checkpointing, e.g., an X server can always ask its clients to

redraw the contents of windows [Sch86a], so that it does not have to save clients' bitmaps when checkpointing.

We strongly feel that a software checkpointing system should allow individual applications to take advantage of the above optimizations. It would be difficult, however, to implement some of our observations in traditional monolithic kernels, such as UNIX. But microkernels, such as Mach [Loe92a] and Chorus [Roz90a], which emphasize the client-server computation model, define a small set of operating system abstractions that allow the flexibility of customizing programming environment at the user level.

Similar to UNIX or DOS operating system environments, we can build a fault-tolerant computation environment on top of Mach, such that when a resilient process starts, it automatically communicates with servers that support checkpointing and message logging. All environments may co-exist on the same hardware at the same time. We argue that using message semantics to guarantee data consistency is both efficient and suitable for building a fault-tolerant system at the software level.

3. The Basic Idea

For any resilient process (also known as the *primary* process), we assume that there is a *backup* process, presumably on a different machine on the network. Under the basic checkpointing scheme, a resilient process saves its execution state periodically to a more reliable storage, in this case, to the backup process. If the primary fails, the backup will resume the computation from a previous checkpoint, instead of always restarting from the beginning, thus guaranteeing forward progress.

Both application programs and the operating system may initiate checkpointing. The action may be triggered either by message exchanges or by timer exceptions. The checkpointing interval, to a certain extent, determines the relative costs of the failure-free execution overhead and recovery delays. However, if a process is sending out a message that will modify the state of a receiving process, it is forced to checkpoint first.

The information as to whether a message would change the state of receiving process is defined by using the modified MIG interface specification language [Loe92b]. We call the communication interface generated by MIG, a *contract*. Client programs link with MIG interface functions at compile time. A message being sent through a contract will guarantee not to cause any inconsistency between the sending process and the receiving process despite failures of the sender, the receiver, or both. Therefore, the consistency of global states is guaranteed at all times. The main advantage of using contracts is that checkpointing can be avoided when a message does not modify the receiver's state.

Our proposed system is an otherwise regular operating system (i.e., Mach) except for those processes that have to be resilient to failures. In our computation model, we do not assume repeatable process executions, that is, we allow non-deterministic executions. Messages are not required to arrive in the same order when a program is executed twice. It is, however, the responsibility of the designer of the server (receiver process) to eliminate all non-deterministic program behavior by carefully defining the contract between the server and its clients (sender processes). For example, if a RPC message could potentially be non-deterministic, the sender should checkpoint before the request can be processed by the receiver. With our simple checkpoint protocol, both multiple rollbacks and cascading rollbacks are prevented.

4. An Example

Using our checkpoint system, the cost of checkpointing depends on the way servers are designed. By changing the server (MIG) interface, an application can decrease the frequency of checkpointing and therefore, the cost of checkpointing.

For instance, in a file server supporting sequential file access, an application has to keep track of an index which points to the next location for file access. If this piece of information is kept in the server, the client would only have to issue an access request and the server would be able to determine the location from its own state. According to our checkpoint protocol, since the state of the server (the index) is modified each time by an access request, the client is forced to checkpoint each time it accesses a file. Similarly, the server is forced to checkpoint each time it returns from an access request in order to keep the consistency of the server state (the real index) and the client state (the assumed index). On the other hand, if the index state is kept by the client, the index would have to be sent along with each access request, but no checkpointing would be needed by the client or by the server if the access is a read request.

When a file is opened, a checkpoint is performed in order to guarantee consistency between the server and the client; since reading a file block does not change the state of file server, the message is simply passed through. Whereas, every block written is subject to later reading, so write messages have to be checkpointed immediately. Checkpointing every block written is expensive, yet it guarantees strict UNIX file semantics. However, most distributed file systems (e.g., SUN NFS, AT&T RFS and CMU AFS) cache data on the client machine, avoiding strict UNIX emulation for efficiency reasons.

The concept of *memory-mapped-file* is studied in Mach [Tev87a]. As a file is opened, the whole file is mapped onto the local virtual memory space. Subsequent read and write calls are treated as accesses to the local memory. The memory-mapped-file concept changes the semantics of UNIX sequential file access to random access. The original concern of memory-mapped-file is to improve the efficiency of distributed file servers. If we take a closer look at the implementation of a memory-mapped-file, we will see two copies of the same file, one on the client machine and the other on the server machine.

It is possible to take advantage of the memory-mapped-file concept and use it to improve the reliability of both the server and the client. Specifically, the copy mapped on the client machine can be treated as a backup to the file on the server. If the server fails, the client can transfer its local copy to the server and rebuild the server state. Under the new scenario, a memory-mapped-file need only be checkpointed when the file is opened or closed. When a file is opened for writing or updating, a new version number is assigned to the client copy of the file. When the file is closed, the whole memory-mapped-file will be flushed back to the server. In other words, we define that a file server changes its state when a file is being closed.

Intermediate write operations can be sent back to the server, but they are merely treated as optimizations to the general mechanism. In the case of client failure, all write operations will be undone. This way, no checkpoint is required for write operations. The close operation will flush back the dirty pages that have not been sent to the server or flush all the modified pages to the server depending on whether the primary server has failed after the last open operation. The performance improvement over the sequential file access mechanism is obvious.

5. The Environment

A user invokes a resilient process by typing:

```
buoy prog argl arg2 ...
```

where, buoy is a utility command, similar to the time command in UNIX. It contacts the checkpoint manager residing on each machine, finds one that will spawn a backup server for the user prog command, requests a port send-right from the new backup server, registers the right for the prog task using task_set_checkpt_port(), and finally, executes the prog command with given arguments. The kernel considers a task to be resilient, if and only if the task has a checkpoint port registered in its task structure. Otherwise, the kernel would treat all functions mentioned above, e.g., task_checkpoint(), as null functions.

After a primary process crashes, the backup server will resume the computation from the last checkpoint received for the primary. For reconnecting all port rights, a unique identifier is attached to every port (Chorus furnishes this abstraction but Mach does not). A backup server, before loading the memory image of the recovery process, publishes a port's unique identifier in the name server, if it should hold the receive-right of the port. Otherwise it tries to look up a port's unique identifier in order to re-establish communications. The name server (checkpoint manager) uses mach_port_get_uid() and mach_port_set_uid() to propagate a port's identifier across the network.

In other words, interprocess communication is resumed using existing Mach mechanisms, but messages in communication channels, such as those in port message queues, are lost in the event of a system or application failure. Logged messages, if necessary, are played back from user level message servers that were intercepting and forwarding messages between clients and servers.

When inquired by a resilient process, the name server tries to find a service port that supports resilient processing. For example, if the name to look up is FOO_BAR, the name server searches the port with name RESILIENT_FOO_BAR in its first attempt; if that fails, it will return the port FOO_BAR with a warning. Regular Mach processes will not see any differences. Similarly, when a resilient process checks in a name, the prefix RESILIENT_ would be added transparently.

6. Implementations

The main kernel abstractions of Mach [Loe92a] are *task*, *thread*, *port*, *message*, *virtual memory* and *memory object*. In order to facilitate checkpointing in the kernel, we add a new system call, task_checkpoint(), in our system. The system call suspends the designated task; it notifies those servers that are currently communicating with the task about the event of their client being checkpointed.

The kernel sends the task's kernel state (e.g., registers etc.), dirty pages and port information to the task's backup server. It then waits for acknowledgments from the notified servers. The servers send acknowledgments after they have reacted appropriately to the kernel notification.

We also add following functions into the Mach kernel task interface:

```
mach_notify_task_checkpoint(),
task_get_checkpt_port(),
task_set_checkpt_port(),
mach_port_get_uid(),
mach_port_set_uid(),
vm_checkpt_policy(),
port_sync_policy(), and
cohort_checkpt_resume().
```

System servers use these functions to define the checkpointing arguments and policies for a given task.

There are several possibilities for checkpointing a virtual memory region. The text region needs to be saved only once. On the other hand, the initialized data region is saved in full for the first time and only the dirty pages are saved afterwards. For the stack region, we only need to save the dirty pages. And for a region mapped to an external memory object, we flush back all dirty pages and let the memory object handle its own checkpointing. All the policies of virtual memory regions are defined by servers, mostly by the UNIX server, through the `vm_checkpt_policy()` call. The UNIX server even specifies some regions, such as those for the emulation library code as don't-care, so the kernel would skip those regions when checkpointing a client task.

In the MIG interface generator, we add three key words, *query*, *modify*, and *partial*, to specify the nature of a message. Each message would carry the new type information in its header. When the kernel sees a modify message, it will force the client (sender) to checkpoint, if it has not already done so. In addition, it will invoke the checkpoint handler function in the server (receiver) before returning a reply. The checkpoint handler is `task_checkpoint()`, by default, and the server may change it to only checkpoint the client's data. If a message has the partial flag on, message servers (e.g., the `NetMsgServer`) will log the message until the message receiver checkpoints.

The kernel coordinates all the servers during checkpointing, e.g., UNIX server, net-message server, file servers, etc., in a client's environment with a one-phase commit protocol. Servers use the Mach `mach_port_request_notification()` call to the kernel to indicate its interest in getting checkpoint notifications, and the kernel informs the servers using the `mach_notify_task_checkpt()` call when their client is checkpointed. Upon receiving notifications, servers must save the client's state, if any, to their own backup tasks or on a stable storage.

Similar to the `vm_checkpt_policy()` call for VM, we introduce a corresponding function for port, namely, `port_sync_policy()`. A server uses the new system call to indicate (to the kernel) how it will react to a client's checkpointing. For example, a server may want to receive a special control message every time its client checkpoints, or not at all, or only when the server port is dirty. We say a port is dirty if there are partial or modify messages sent through the port since last checkpoint. The kernel keeps a count on the number of control message being sent out, and waits for acknowledgments from informed servers before committing a checkpoint.

7. Related Work

Checkpointing is often implemented by using dedicated hardware. For example, Targon/32 [Bor89a] designates a spare processor to log all messages, and guarantees that every message is delivered to its receiver, receiver's backup, and sender's backup in an ordered atomic fashion via special devices. Two other well known hardware fault-tolerance systems are Stratus [Har87a] and Tandem [Bar87a]. They provide redundant circuitry such that no single hardware failure can disable an entire system. All three systems support checkpointing at the operating system level.

Many pure software approaches for distributed systems have been described in the literature. For example, the algorithm in [Koo87a] insures that when a process checkpoints (or recovers from a failure), a minimal number of communicating processes are forced to checkpoint (or rollback) at the same time. In object-based systems, message semantics helps to eliminate some checkpoint (and recovery) dependencies [Lin90a], therefore improves the performance of the previous algorithm. To avoid multiple and cascading rollbacks, there are also proposals that emulate Targon/32 system in software [Bab90a] or log messages only on their senders in order to tolerate single site (or n site) failures [Joh87a, Str88a].

Because of the popularity of UNIX, many attempts have been made to implement checkpoint mechanisms on top of it [Tay86a, Smi89a]. Unfortunately, UNIX was designed for stand-alone machines in the early 70's. It distinguishes main memory from the secondary storage; the file system and network protocols are built in the kernel. Certain process information is hidden in the kernel, i.e., the *u-block* for efficiency purposes. Moreover, some UNIX attributes are machine dependent, e.g., pid and socket number, etc. For all the above reasons, it is inherently difficult to adopt UNIX kernel as the base for distributed fault-tolerant systems.

The KeyKOS [Bom92a] system guarantees process and data persistency by conducting periodic system-wide checkpoints. The kernel implements a single-store abstraction, that is, physical memory is mapped onto the secondary storage. The dirty pages in memory are dumped to a predefined disk partition during checkpointing, from where a daemon process would move them to their mapped blocks on disk. Interprocess communication is end-to-end using capabilities and there are no message queues in the kernel. After a power failure, the failed machine falls back to the previous checkpoint. Consistency among coordinating processes is application user's responsibility, while the kernel allows processes to commit pages individually.

BirliX [Sch92a] has mechanisms to checkpoint an individual process (known as *team*). The process state is mapped in its virtual memory, and the virtual memory is mapped to secondary storage servers (called *segments*). The kernel checkpoints a process address space in the volatile memory by marking it copy-on-write before the dirty pages are sent to segments asynchronously. Applications selectively choose whether they want to be checkpointed, and may implement different global checkpoint schemes based on the fact that any one process can be checkpointed by the kernel. Similar to KeyKOS, however, applications must manage the dependency list of processes themselves.

8. Conclusion

Our system is a regular Mach system except for the resilient processes. Using the utilities we have implemented, one can achieve transparent fault-tolerant processing for client applications. The kernel checkpoint functions follow the Mach microkernel philosophy, that is, they only define mechanisms. And different checkpoint policies should be implemented in system servers. We demonstrate our system by designing one specific checkpoint algorithm. However, many different checkpoint algorithms may be implemented using our kernel support. With carefully designed server interfaces, the overhead incurred due to checkpointing can be further reduced through data caching concepts and by using the Mach external memory management mechanism.

Our approach does not require any special hardware. And the proposed fault-tolerant system is implemented on a set of Sun3 workstations. We have basically finished the kernel programming and debugging. A simple testing environment is working, e.g., we can run a program as resilient process and restart from its checkpoints. The development of a prototype file server and several benchmarks are under way. The total checkpointing related code in the kernel is about 2,500 lines of C code. Comparing with around 100,000 lines of Mach kernel code, the implementation overhead is small, yet it creates a new testbed for fault-tolerant computing.

References

[Bab90a] Ozalp Babaoglu, "Fault-tolerant computing based on Mach," *Operating Systems Review* **24**(1), pp. 27-39 (January 1990).

[Bar87a] J. Bartlett, J. Fray, and B. Horst, "Fault tolerance in Tandem computer systems," pp. 55-76 in *The Evolution of Fault-Tolerant Computing*, Springer-Verlag, Wien-New York (1987).

[Bom92a] Alan C. Bomberger et al., "The KeyKOS nanokernel architecture," pp. 95-112 in *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architecture*, Seattle (April 1992).

[Bor89a] Anita Borg and Wolfgang Blau et al., "Fault tolerance under Unix," *ACM Transactions on Computer Systems* **7**(1), pp. 1-24 (February 1989).

[Har87a] E. S. Harrison and E. J. Schmitt, "The structure of System/88: A fault-tolerant computer," *IBM Systems Journal* **26**(3), pp. 293-318 (1987).

[Jas69a] David P. Jasper, "A discussion of checkpoint/restart," *Software Age*, pp. 9-14 (October 1969).

[Joh87a] David B. Johnson and Willy Zwaenepoel, "Sender-based message logging," pp. 14-19 in *Proc. of the 17th International Symposium on Fault-Tolerant Computing*, Pittsburg, PA (July 1987).

[Koo87a] Richard Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering* **SE-13**(1), pp. 23-31 (January 1987).

[Lee90a] P. A. Lee and T. Anderson, in *Fault Tolerance: Principles and Practice*, Springer-Verlag, Wien-New York (1990).

[Lin90a] Luke Lin and Mustaque Ahamad, "Checkpointing and rollback-recovery in distributed object based systems," pp. 97-104 in *Proc. of the 20th International Symposium on Fault-Tolerant Computing*, Newcastle upon Tyne, England (June 1990).

[Loe92b] Keith Loepere, "Mach 3 Server Writer's Guide," in *Open Software Foundation, MK67 edition* (January 1992).

[Loe92a] Keith Loepere, *Mach 3 Kernel Principles,* Open Software Foundation, MK67 edition (January 1992).

[Roz90a] Marc Rozier et al., "Overview of the Chorus distributed operating systems," Technical Report CS/TR-90-25, Chorus Systems (April 1990).

[San85a] Russel Sandberg et al., "Design and implementation of the Sun network file system," pp. 119-130 in *Proc. of the Summer 1985 USENIX Conference*, Portland, Oregon (June 1985).

[Sat90a] Mahadev Satyanarayanan et al., "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers* **39**(4), pp. 447-459 (April 1990).

[Sch86a] Robert W. Scheifler and J. Gettys, "The X window system," *ACM Transactions on Graphics* **5**(2), pp. 79-109 (April 1986).

[Sch92a] P. Schuller et al., "Performance of the BirliX operating system," pp. 147-160 in *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architecture*, Seattle (April 1992).

[Smi89a] J. M. Smith, "Implementing remote `fork()` with checkpoint/restart," *Technical Committee on Operating Systems Newsletter* **3**(1), pp. 15-19 (1989).

[Str88a] Robert E. Strom, David F. Bacon, and Shaula A. Yemini, "Volatile logging in n-fault-tolerant distributed systems," pp. 44-49 in *Proc. of the 18th International Symposium on Fault-Tolerant Computing*, Tokyo, Japan (June 1988).

[Tay86a] David J. Taylor, "Backward error recovery in a Unix environment," pp. 118-123 in *Proc. of the 16th International Conf. on Fault-tolerant Computing Systems*, Vienna, Ausria (July 1986).

[Tev87a] Avadis Tevanian, Jr., "Architecture-independent virtual memory management for parallel and distributed environments: The Mach approach," Technical Report CMU-CS-88-106, School of Computer Science, Carnegie Mellon University (July 1987).

[Use92a] Usenix, *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architecture*, Seattle, Washington (April 1992).

Evaluating MESHIX – A UNIX Compatible Micro-Kernel Operating System

Tom Stiemerling Andy Whitcroft
Tim Wilkinson Nick Williams Peter Osmon

Systems Architecture Research Centre
City University
London England
{ trs | andy | tim | njw | p.osmon }@cs.city.ac.uk

Abstract

A number of current operating systems have been implemented using the micro-kernel approach, which moves functionality from the kernel to user-level servers. MESHIX is a System V UNIX compatible message-passing distributed operating system implemented using this approach. In this paper we evaluate some of the design choices taken in MESHIX, particularly aspects associated with the micro-kernel. To examine the overhead associated with the micro-kernel we analyse local RPC performance and message-delivery, and give timings for the constituent parts of these operations. We conclude that although MESHIX has similar performance to other micro-kernel systems, a significant penalty is paid for the micro-kernel and propose specific optimisations to reduce this penalty.

1. Introduction

MESHIX is a System V UNIX compatible distributed message-passing operating system which is implemented using the micro-kernel approach [Win90a]. Several micro-kernel based operating systems have emerged over recent years following a general movement away from monolithic systems such as BSD UNIX, System V and more recently Sprite [Ous88a], to smaller, simpler and more portable kernels. By layering the operating system in this way, the kernel can be kept fast and efficient with the more complex tasks being provided by servers. Other implementations of this model are Amoeba [Mul90a], Chorus (version 3) [Bri91a] and Mach (version 3) [Acc86a].

The purpose of this paper is to evaluate some of the design choices taken in MESHIX, particularly aspects associated with the micro-kernel. Some parts of MESHIX, such as the filesystem and process migration, are therefore outside the scope of this paper and are not described here.

We first give a brief overview of the current hardware platform for MESHIX, then we describe the organisation of the micro-kernel and some of the functions it performs, a performance analysis of RPC is then presented which highlights some of the problems, and we conclude with our plans for the further development of MESHIX.

2. TOPSY Multicomputer

The current implementation of MESHIX runs on the TOPSY multicomputer, a MC68030 based distributed memory multiprocessor [Win89a]. TOPSY is designed to be a low-cost *extensible* general purpose computer which supports standard UNIX software. A TOPSY machine consists of a number of nodes connected in a mesh topology by a circuit-switched network.

Each TOPSY node contains an MC68030 processor,[†] 8 MBytes of memory and various peripheral controllers such as a SCSI disk controller, a MC68450 DMA controller, a serial interface, and a network interface. A separate Ethernet card can also be connected. The interconnection network is a custom circuit-switched mesh network called MESHNET [Wil91a]. The current network devices have a maximum bandwidth of 12 Mbytes/sec, although the performance of the particular DMA controller and the software overhead constrain the effective throughput to about 0.5 MBytes/sec.

The maximum size TOPSY machine that can be constructed with the current hardware is 256 nodes. We have built a number of 4 node machines and a 16 node machine which are used mainly for systems development. The machines are connected to the departmental Ethernet and generally accessed using the X11 interface.

3. MESHIX Operating System

Operating systems such as MESHIX and Sprite aim primarily to provide a standard UNIX interface giving access to existing UNIX software but with increased performance, which is accomplished using distributed servers running in parallel over multiple networked processors. In Sprite, the physical distribution of the machine is hidden behind a shared filesystem. MESHIX, however, makes the entire distributed system appear as a single, timesharing UNIX. In contrast, systems such as Amoeba, Chorus and Mach are designed in an object oriented manner to provide a general set of facilities upon which different operating system interfaces can be constructed, and provide some support for parallel and distributed programming. These overall design decisions, and the implementation on the TOPSY machine, have dictated to some extent how MESHIX is structured internally.

3.1. Structure of the Kernel

MESHIX is divided into three layers: the micro-kernel, the macro-kernel which includes system server processes, and the user processes. The micro-kernel provides the basic message-passing, interrupt handling, scheduling and memory management functions. The network driver, the clock driver and the swap manager process are part of the micro-kernel. This is shown in Figure 1.

[†] Clock speed of 16MHz and 1 wait state memory.

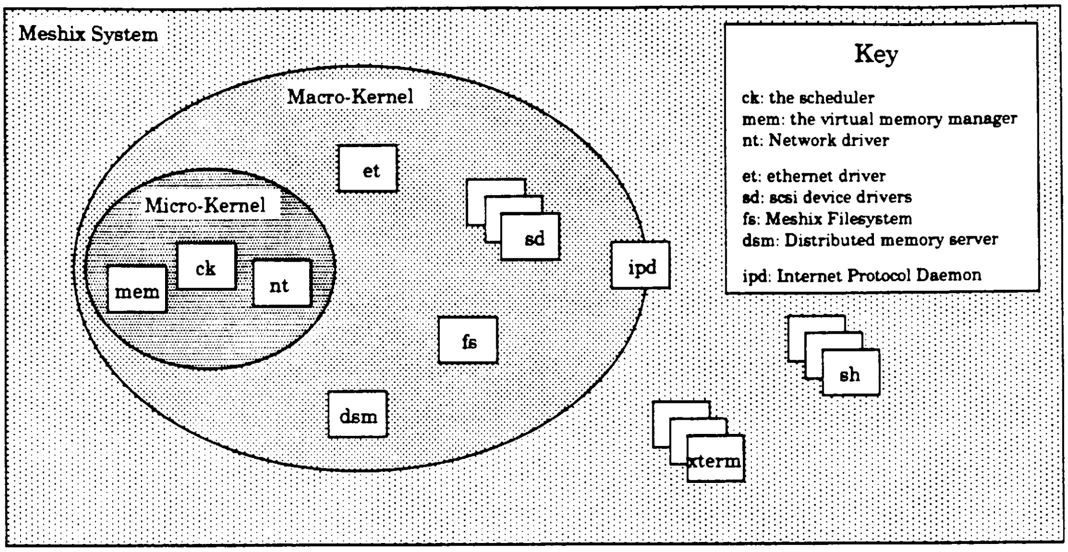


Figure 1: *The structure of the MESHIX operating system. The "ipd" process lies in the user applications area of the operating system, but provides services to applications in a similar way to other macro-kernel services.*

The macro-kernel includes other device drivers and the server processes which provide System V operating system functions using the client-server abstraction. These have special memory and scheduler privileges and run at a higher priority than user processes. Examples are the process and signal manager services, and the device drivers. Some servers like the process, filesystem managers, and device drivers which are attached to specific hardware, are placed only on certain nodes, while other services are replicated on each node.

In the highest layer is the application space, in which user processes are placed. These can ıequest services from the operating system using remote procedure calls which are directed to the appropriate system servers, or in some cases by communicating directly with the micro-kernel.

Unlike Chorus, which supports user-level device drivers, the MESHIX device drivers and most system services are compiled into the kernel and execute in the kernel address space, improving their performance. All communication between servers and drivers however, is accomplished using message-passing.

3.2. Interprocess Communication

Communication between processes on a node and between nodes is by message-passing. A message is a contiguous byte structure of *fixed* size containing addressing information, credentials and a data area. Messages are addressed using unique identifiers within a TOPSY machine defining the network location of a node and a *port* on that node, the node location being given relative to the sender. A port is an integer representing a registered recipient of a message on any node. Ports may have symbolic names attached to them, in which case processes can look up a service by name using the *service map*. This map provides aliasing and location transparency for services. Ports are protected by assigning standard UNIX *user*, *group*, *world* permissions to them.

The `send()` function is provided to send a message asynchronously to any destination in the machine. The complementary function, `receive()`, blocks waiting for a message from another process. For remote procedure calls, the `transact()` function is available to send a request to a specified address and then block waiting for the server's reply.

Because of the small message size the *copy-on-write* optimisation for message transfer as used in Mach is not applicable. To enable copying of messages, MESHIX uses the `vtop()` function to *re-map* the message from the sending process' address space into the kernel address space.

When transferring memory between nodes, messages cannot be the sole agent because of the overhead of fragmenting a large contiguous memory area into many fixed size messages. Instead, a request is made to the micro-kernel to transfer the entire memory block. To do this, the micro-kernel communicates directly with the network driver. The routine `globcopy()` is provided to perform direct memory transfers between process address spaces (which may be on different nodes), and is used to transfer disk blocks and virtual memory pages.

3.3. System Call Execution

In MESHIX the UNIX system calls are compiled into RPC stubs [Bir84a] which marshal the parameters into a message packet and then send the message to the required server process using the `transact()` call. The memory manager process for example is responsible, among others, for the `fork()`, `exec()` and `exit()` system calls.

The location of the server processes is completely transparent to the client, in the case of a remote server the message being first routed to a pre-registered virtual port and then on to the server's true destination by the messaging system. The server then executes the system call on behalf of the client process, and may communicate with other processes during this. Any result is then sent back to the waiting client via a return message.

4. Virtual Memory Management

The MESHIX virtual memory management system originates from a segmented processor architecture (the 68070) but was extended to support a paged model when the system was ported to the 68030. The facilities provided are the same as conventional System V UNIX.

4.1. The Virtual Memory Facilities

The virtual memory model supports the following features:

- Demand paging of binaries,
- Page stealing of both read-only and modified pages,
- Copy-on-write data duplication,
- Per node binary caching, and
- Idle time page scrubbing.

Of these facilities, the first two are commonplace in current UNIX systems, whilst copy-on-write data duplication is used, we believe, more extensively in MESHIX. Copy-on-write allows data to be duplicated into two or more independent processes' address spaces without actually performing a physical copy which may ultimately be wasted time if the duplicated data is not modified by any sharer. However, when data is modified, the process does not change the shared version but takes a copy. Only modified pages are ever copied, thus decreasing the time to duplicate data and reducing the core utilisation.

Generally, copy-on-write techniques are used to speed execution of `fork()` which duplicates an entire process. In MESHIX copy-on-write is also used for `exec()`; every node in the system supports a binary cache which is used to hold copies of the most recently executed programs on that node. When a process executes a new program, this cache is searched to locate it. If found, the program executes directly from the cache. However, in the course of execution, pages of the program will be modified. When this occurs, the copy-on-write mechanism is used to duplicate the data from the cache rather than lose the cached copy.

The binary cache stores its data in otherwise unused core pages. As a system requires these pages, they are removed from the cache's control to prevent the cache holding copies of binaries which are no longer useful or relevant. Additionally, pages may be lost when a copy-on-write from the cache is attempted. If there is insufficient core available and no other processes are sharing the page, rather than perform a copy-on-write operation, the page is removed from the cache and used directly.

The addition of this binary cache greatly improves program startup time by removing the need to demand load from the disk system (which in MESHIX could conceivably be on another node). However, some communication with the file system is still necessary to determine whether the binary in cache is up-to-date. If not, it is flushed from the cache and re-demanded from the file system.

Finally, the **idle** process on each node is assigned the task of page scrubbing; filling free core pages with zeroes. Most requests for zero-filled pages are satisfied with pages from this source.

Operating	Times (μs)	
System	Zero-fill	Copy-on-write
Chorus (one 8K page)	1150	1700
Mach (one 8K page)	1550	2120
MESHIX (four 2K pages)	1360	3740
MESHIX (one 2K page)	340	935

Table 1: *Times taken to allocate and zero-fill a page of memory and the performance of copy-on-write. The quoted figures for Chorus and Mach are taken from [Abr89a]. If we had 8K pages, the zero-fill time for one page would be the same as the time for one 2K page, due to the **idle** page scrubber. The copy-on-write times shown would also be reduced, as only one page fault would occur.*

4.2. Performance

Some figures are presented in Table 1, comparing MESHIX times to similar measurements from other systems.

As can be seen, MESHIX compares favourably, even with our smaller page size. However, one failing of the system is its handling of the binary caches which it does on a per node basis. If a binary cannot be found in the cache it is demanded from disk; if it is in other node's cache the fact cannot be determined. For large parallel programs, initial startup time can be large as each node of a sixteen node machine demands the binary from the same disk (in fact, we added a block cache to the disk device driver to prevent numerous disk accesses for the same page).

5. Inter-Process Synchronisation

Because MESHIX is a message passing kernel, synchronisation between two processes implicitly occurs with the transmission of data from a *sending* process to a *receiving* process. Although this is a simple model for synchronisation, it is too limited. For example, a simple point-to-point message exchange does not handle such events as one-to-many process synchronisation. Therefore, another mechanism is provided in MESHIX: *Sleep/Wakeup.*

5.1. Sleep and Wakeup

The Sleep/Wakeup mechanism is primarily provided for the use of the virtual memory implementation (see §4.1) where a page fault repair might need to release many sharing processes. This situation arises when read-only binaries are shared. Messages cannot be used here for a number of reasons:

1. A number of arbitrary processes can wait for the same synchronisation event, a situation the message system could only handle by sending of multiple messages,
2. Page faults required efficient and speedy handling while only requiring local synchronisation operations. Therefore, the cost of using messages when no data is to be transferred and no remote synchronisation is to be undertaken is prohibitive,
3. The message system relies on facilities provided by the virtual memory system and any attempt to use messages to implement it can lead to unresolvable deadlock conflicts.

Sleep allows a process to block on an arbitrary physical address. The process will not be executed again until another process issues a *Wakeup* on that same address, at which point all processes waiting on the address are released. This mechanism is identical to that used in UNIX System V [Bac86a]. This simple system has a number of drawbacks. Most importantly, it causes processes to block on local physical addresses and so cannot be used to provide synchronisation between remote processes. Furthermore, because there is no sleep control associated with the address being blocked upon, an unforeseen race condition between two processes could allow the wakeup to occur before the sleep. In such circumstances the wakeup would be lost and the sleep would continue indefinitely.

5.2. Comments

The sleep/wakeup mechanism is efficient (see Table 2) and simple although it can only be used between kernel processes on the same node. However in retrospect, the introduction of another mechanism was a mistake. A far better alternative would have been to improve or redesign message delivery, allowing one-to-many messages and improved efficiency of local delivery.

Operation	Time (μs)
Sleep	105
Wakeup	155

Table 2: `Sleep()` *and* `Wakeup()` *timings. Times include a reschedule (100* μs*).*

6. Interrupt Handling and Drivers

In UNIX there are two ways to handle interrupts from devices. The first is to process the event in the interrupt handler itself; known as a bottom half routine. Such handlers must be quick and not attempt to gain resource locks since they cannot block (the interrupts will be disabled so preventing any other event from occurring). Alternatively, the bottom half routine simply initiates an action in an top half routine and terminates. The top half routine is then executed by the kernel as a normal kernel procedure, obtaining locks and blocking as necessary. The bottom half routines provide very fast interrupt handling but are limited by what they are allowed to do. Top half routines may handle much more complex situations but do so by paying a time penalty.

6.1. MESHIX Interrupt Handling

In MESHIX three mechanisms are provided to handle interrupts. These are:

1. Priority handlers,
2. Synchronisation handlers,
3. Message handlers.

Priority handlers correspond to UNIX bottom half routines, providing limited access to resources but with the lowest latency. In MESHIX these are only used to handle kernel profiling. *Synchronisation handlers* correspond to UNIX top half routines, providing a means to release a MESHIX kernel process. These are adequate if an interrupt only informs a process of an important event but does not need to provide any other data. In MESHIX these are used in the SCSI driver. Finally, *Message handlers* have no UNIX equivalent. When an interrupt is generated, a message is sent to the relevant handler. This message is then processed by the handler.

Message handlers are much more flexible than either of the other two mechanisms. Firstly, the handler need not be located on the same node as the interrupt generator. Secondly, the interrupt handler need not be part of the kernel and can exist as a standard user process (MESHIX provides its TCP/IP service in this way). This enables drivers to be improved and modified without recompiling the kernel and, with care, inserted and removed from running machines.

A comparison of the different overheads in using these mechanisms is shown in Table 3.

Mechanism	Time (μs)
Priority handlers	55
Synchronisation handlers	205
Message handlers	345 (506)

Table 3: *Timing overheads of the interrupt mechanisms. The interrupt itself takes 50 μs. For the message handler, also shown is the time taken by the server to receive the message.*

6.2. Comments

The provision of three mechanisms to do essentially the same task is horrible. An improved messaging system would allow the combination of synchronisation handlers and message handlers. MESHIX could then be modified to rely on only that system (we will allow the exception of kernel profiling interrupts).

7. Analysis of Meshix RPC Performance

The micro-kernel implementation of MESHIX results in increased cross-domain communication and context-switching to achieve basic system functions. The MESHIX message system is examined and as an example, we consider MESHIX RPC performance.

In MESHIX, Amoeba and Chorus, local and remote RPC is implemented using message passing (although the latest version of Chorus allows an optimisation similar to lightweight RPC [Ber89a]). In Sprite a local RPC is simply a trap into the kernel. A table of comparative times to perform these operations is shown in Table 4. MESHIX's RPC times compare favourably with the other systems, but the local RPC time is disappointing. Below, we examine local message delivery in detail.

Operating System	Processor	Time (μs)	
		Local	Remote
Amoeba	68020	500	1200
Mach	C-VAX	-	754
MESHIX	68030	900	1300
Sprite	68020	70	1900

Table 4: *Time taken to perform an RPC for various operating systems (quoted results from [Dou91a, Ber89a])*

7.1. Message Delivery

The message delivery system is invoked in two main ways: as a result of `send()`, `receive()` or `transact()` system calls, or by direct invocation of the delivery system by the kernel itself. The system call interface is entered by a trap call with two parameters in registers. The function is decoded and the address parameter verified with the process's message permission mask held in the process table. The delivery system is then called to pass on the message.

The delivery system contains two main parts: `put_message()` and `get_message()`. `Put_message()` is the kernel routine implementing the send system call. It validates the address of the message and maps it into the kernel memory space using `vtop()`. If the destination address is local, then the address is mapped through the service map. If the destination is then remote, the destination address is packed into the message address fields and the destination address is aliased to the network device driver. At this point the destination is local. If the flags in the receiving process indicate the process is blocked in receive then the process table is checked to see if the message is acceptable. If the message can be delivered then it is copied into the receiver memory buffer and the destination process is added to a run queue. If the process is not waiting then a buffer is allocated and the message copied into the buffer. The buffer is then appended to a doubly linked list of waiting messages in the destination process's table entry (`put_message()` takes a further option which allows synchronous message delivery by kernel processes. Currently this mechanism is only used by interrupts.).

`Get_message()` is the kernel routine which implements the receive system call. Addresses are mapped through the service map in the same way as for `put_message()`. The message pointer is validated and mapped into the kernel by `vtop()` as in `put_message()`. A selective receive is implemented by storing the required source process address in the process control block. If there are queued messages then each is searched in turn for a matching source. If a match is found then the message is copied into the user buffer and the kernel buffer is linked out of the message queue and deallocated. The process then continues. If no message can be found then the process is blocked in the receive state and another process scheduled.

7.2. Performance Analysis

Figure 2 shows a time diagram for the processing and hardware activity involved in a typical local RPC. It identifies the most costly activities and their relationships.

Assume a client and server process on the same node. At the start the client process is executing user code (activity *a1*) and the server is initialising (activity *b2*). When the server completes initialisation, it blocks `receive()`ing a message. During execution the client process requests a service. It forms a request message and performs a `transact()` with the server. This causes a message to be sent to the server which then is scheduled. The server performs the required service and `send()`s a reply message to the client which is rescheduled and continues.

Below we describe the observed behaviour of the system, concentrating on the activities identified in Figure 2. The activities are described and the times taken by each identified.

Client Process Execution (*a*)

> Client process executing in its own domain.

Context Save (*s*) and Restore (*r*)

> At various stages in the RPC call the system saves and restores the processor state. Saves and Restores each take approximately 15 μs.

Message Mapping (*M*) and Release (*R*)

> In order to transfer a message from one process address space to another the micro-kernel maps the source and destination mes-

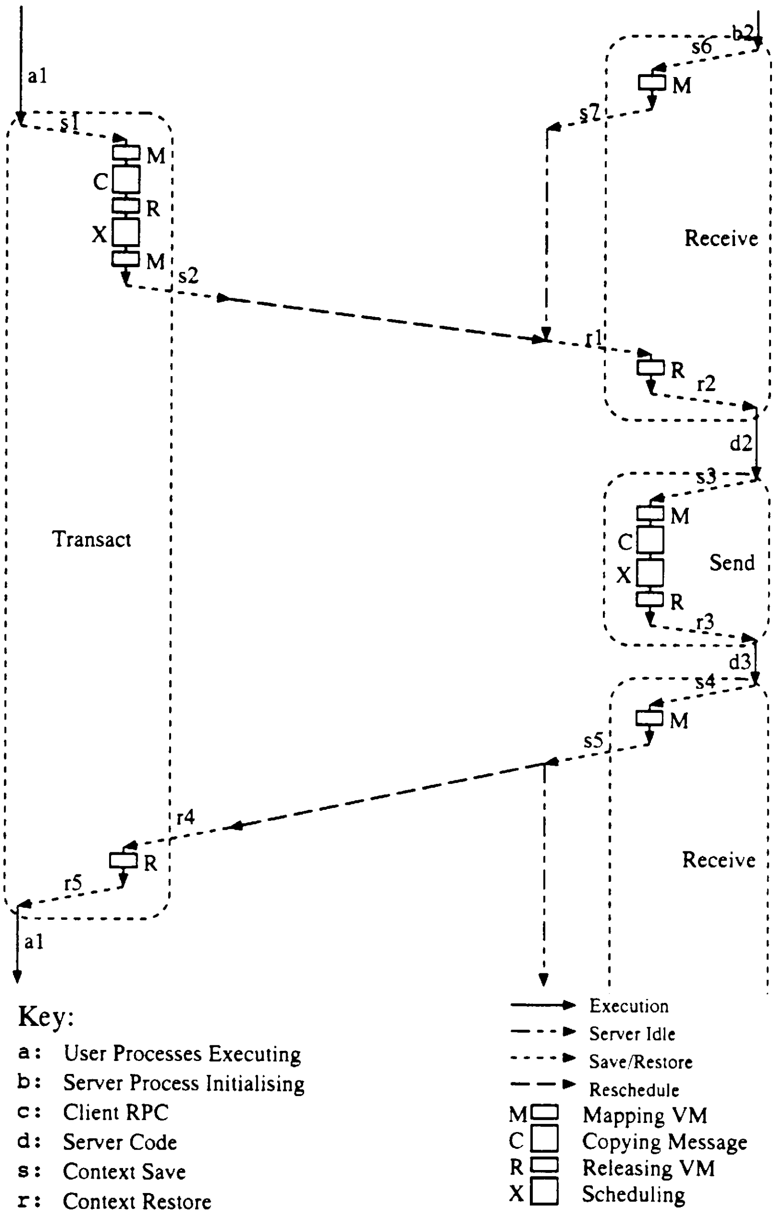


Figure 2: *A time diagram showing the activities of client and server processes involved in a remote procedure call*

sage buffers into the kernel address space. When the mapping is no longer required it is released. Mapping a memory segment takes approximately 26 µs. Releasing a memory segment takes approximately 21 µs. The release necessitates a TLB and cache flush. If the destination process is not currently waiting for a message then the data is copied into a kernel message buffer and this forces an additional message copy.

Message Copying (*C*)

The message is transferred from the source message buffer to the destination message buffer by the processor. This takes approximately 47 µs.

Server Process Execution (*d*)

Activity *d2* indicates the time taken by the server to perform the requested task, at the end of which the reply is made. Activity *d3* indicates the time taken by the server to return to the main server loop and wait for another request.

Processor Rescheduling (*p* and *X*)

Once the client has sent its request message it sleeps, waiting for the reply message. At this stage there is a reschedule causing the server to run. At the end of the server run it sleeps waiting for a new request and a second reschedule causes the client to continue. The reschedule involves finding and selecting a process to run, this takes approximately 100 μs. If the destination of a message send is blocked waiting to receive the message then the process is added to the scheduler queues. This takes approximately 34 μs.

Miscellaneous

In addition to the major activities detailed above there are a number of minor overheads which account for a further 40 μs.

7.3. Measured Performance

From the times identified above it is now possible to determine the total RPC time for MESHIX.

Activity	Time in μs		
	Send	Receive	
Scheduling	34	34	(0)
Context Save	15	15	
Context Restore	15	15	
Mapping	26	26	
Releasing	18	18	
Message Copy	47	0	(47)
Miscellaneous	40	40	
Total	195	148	(161)

Table 5: *Breakdown of where the time goes during send and receive operations*

These figures give 195 μs for a `send()` and 148 μs for a `receive()` (161 μs if request is delivered while the destination process is running). A `transact()` costs 313 μs (one `send()`, one `receive()` less a context save and restore). An RPC is made up of one `send()`, one `receive()` and one `transact()` plus the rescheduling overheads (shown in Table 6). Using these figures, the total null RPC time is 816 μs.

Activity	Time in μs
Scheduling	100
Context Save	30
Context Restore	30
Total	160

Table 6: *Micro-Kernel Reschedule Time*

7.4. Comments

During the progress of a RPC call ten context saves and restores are performed. These, in most cases, are unnecessary as the kernel is a "well behaved" program and saves changed context. Physical servers (those running in the kernel memory map) use the same mechanism provided to users preventing many optimisations. Judicious optimisation would reduce context saves and restores to four with a saving of 90 µs. The actual context switch time is a direct function of the processor speed and cannot be improved.

The message data is mapped from the user process into the kernel in order to copy it to the destination, then the map is released. Next a new map (to the same memory) is made for the reply message. If the map were retained for duration of the RPC then maps and releases could be reduced from eight to six with a saving of 44 µs. In addition to the visible costs of the map and release each map incurs a hidden performance loss caused by forced TLB and cache flushes.

If the destination of the RPC is waiting when the RPC is initiated, then the message data is only copied once to the destination buffer. However, if the destination is busy the message is copied into a kernel message buffer. As the initiator is waiting for a reply message from the server, the message buffer is not changed and therefore need not be copied. This would result in a saving of 47 µs.

Scheduling overheads resulting from the RPC mechanism account for 268 µs. The RPC involves two full context switches; in comparison a conventional UNIX system call would make a single context switch (one save and one restore).

Inspection of the messaging profiles indicate that these improvements and hardware support [Whi92a] reduce the local RPC time below 500 µs.

8. Conclusion

We have described the micro-kernel of the MESHIX operating system, and have analysed the performance of virtual memory, exception handling, synchronisation and message-passing. MESHIX performs similarly to other micro-kernel systems in these areas.

We conclude that there is a significant penalty in structuring MESHIX as a micro-kernel using a client-server abstraction and message-passing, as we have shown in our analysis of RPC performance. This results mainly from the increased number of context-switches and message-passing overhead necessary to perform basic kernel functions. The performance penalty must be compared to the software-engineering advantages of structuring the system in this way.

A number of specific optimisations to the message-passing system were proposed to decrease local message-delivery time. We are also investigating the use of a co-processor to provide message-passing and scheduling support (we estimate that scheduling overhead can be reduced to 10 µs) [Whi92a]. Upgrading the TOPSY hardware to a processor with support for address space identifiers, such as the MIPS R4000, would also reduce the performance penalty.

References

[Abr89a] V. Abrossimov and M. Rozier, "Generic Virtual Memory Management for Operating System Kernels," in *Proceedings 12th Operating Systems Principles* (1989).

[Acc86a] N. Accetta, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "MACH: A new kernel foundation for UNIX development," in *USENIX summer conference* (July 1986).

[Bac86a] M. Bach, *The Design of the UNIX Operating System*, 1986.

[Ber89a] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight remote procedure call," in *ACM Operating Systems Review* (December 1989).

[Bir84a] A. Birrell and B. Nelson, "Implementing remote procedure calls," in *ACM Transactions on Computer Systems* (1984).

[Bri91a] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier, "A new look at microkernel-based UNIX operating systems: lessons in performance and compatibility," in *Proceedings of EurOpen 91 Conference* (May 1991).

[Dou91a] F. Douglis, M. Kaashoek, and A. Tanenbaum, "A Comparison of Two Distributed Systems: Amoeba and Sprite," in *Technical Report*, Dept of Mathematics and Computer Science, Vrije Universiteit (February 1991).

[Mul90a] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: a distributed operating susyem for the 1990's," in *IEEE Computer* (June 1990).

[Ous88a] J. Ousterhout, A. Cherenson, M. Douglis, M. Nelson, and B. Welch, "The Sprite network operating system," in *IEEE Computer* (February 1988).

[Whi92a] A. Whitcroft and P. Osmon, "The CBIC: Architectural support for message passing," in *UK Performance Engineering Workshop*, England (1992).

[Wil91a] Tim Wilkinson, Andy Whitcroft, Phil Winterbottom, and Peter Osmon, *The Meshnet Multi-stage Communications Network*, May 1991.

[Win89a] Phil Winterbottom and Peter Osmon, "Topsy: An Extensible UNIX Multicomputer," in *Proceedings of the UK IT90 Conference* (1989).

[Win90a] Phil Winterbottom and Tim Wilkinson, "Meshix: A Unix like operating system for distributed machines," in *Proceedings of UKUUG Technical conference, 1990* (June 1990).

Basing Micro-Kernel Abstractions on High-Level Language Models

David L. Cohn Arindam Banerji Michael R. Casey
Paul M. Greenawalt Dinesh C. Kulkarni

Distributed Computing Research Laboratory
University of Notre Dame, USA
dlc@cse.nd.edu

Abstract

Micro-kernels and operating systems export a set of abstractions and services. Traditionally, these have been designed to mimic low-level hardware constructs. However, a micro-kernel should be devised with the same concern for computational models that is used in designing high-level languages, balancing ease of use, flexibility, and performance. Higher-level abstractions can provide ample opportunity for efficiency without dictating the nature of the operating system above the micro-kernel. They also allow a variety of implementation options which enables interoperation between new kernels and existing systems. The abstractions in ARCADE, an ARChitecture for A Distributed Environment, mirror the constructs found in many procedural languages. The architecture has been realized as a micro-kernel and as add-ins to widely used operating systems. This paper describes some of the experiences with the ARCADE high-level abstractions as well as their relationship to other work.

1. Introduction

Traditionally, micro-kernels, and sometimes operating systems, have presented abstractions which reflect a virtual machine view of computing. By offering a hardware-based service set, they provide a highly flexible platform for applications. Unfortunately, they also provide one that is not easy to use, and thus application programmers are not always capable of extracting maximum performance. However, a service set based on high-level abstractions, such as those in modern programming languages, can be easier to use. This is particularly valuable in distributed systems where machine-level abstractions are complex. In addition, since the services are optimized by systems programmers, they may be very fast. Also, high-level abstractions can be added to existing operating systems to allow interaction between heterogeneous systems.

1.1. Virtual Machines and High-Level Abstractions

A primary goal of operating systems is to provide controlled access to a computer's resources. These include hardware resources such as memory and processor time, data resources such as files and keystrokes, and abstract resources such as buffers and semaphores. Typically, the operating system exports services designed to allow client code to be written as if it were the sole user of the resources. In order to keep the service layer thin and to minimize overhead, the abstractions seen by the client resemble the actual hardware. Thus, they are generally low-level and often machine specific.

Modern software, however, is written in high-level languages, and its design integrates the constructs provided by such languages – procedure calls, global variables, objects, pointers, etc. However, these abstractions do not always directly map to the underlying system. Related data structures in a program can be mapped onto different virtual pages, thus making items separate that should be grouped. The reverse is also true; unrelated things can be mapped onto the same page, generating contention for it. To prevent such mismatchs, the abstractions presented by the system should resemble those used by high-level languages.

1.2. The Value of High-Level Abstractions

High-level languages provide programmers with a set of tools to facilitate the construction of applications. They optimize the trade-off between *expressiveness*, *performance* and *flexibility*. Their models enable programmers to express applications concisely, and their constructs are easy to understand. Clearly, a problem is often more simply described in C than in assembler. Modern compilers are written with a deep understanding of performance issues on the target machine. It is, therefore, not reasonable for an application programmer to manually optimize low-level code for a modern processor. However, ease of use and performance come at a price. High-level languages provide less direct control than assembly code, and each language tends to focus on a particular class of applications.

Operating systems and micro-kernels provide a set of tools and also must make tradeoffs. The same three measures can be applied to them, and a similar set of arguments can be made.

An operating system which is modeled on high-level abstractions offers expressiveness. The power of its abstractions and services is more accessible if they match a programming model. Consider the RPC programming paradigm, perhaps the most successful distributed cooperation model. It transforms a low-level message passing system into a high-level abstraction. By hiding the details of network communication under the cloak of a classical procedure call paradigm, RPC reduces programming complexity in distributed systems. It generalizes a familiar sequential flow control mechanism by allowing the procedure call target to be remote. The success of this paradigm can be traced to the universal nature of the procedure call.

Performance is a critical measure for operating systems. It might seem that it is always possible for a system based on low-level abstractions to be at least as efficient than one whose abstractions are high-level. The high-level abstractions could simply be constructed from the low-level ones. For example, a programmer could use basic communication primitives to build his or her own RPC system. However, in reality, it does not work that way. The typical application programmer has neither the time nor talent to write system level code. When higher-level services are built as add-ins between the application and the operating system, the services are written on top of an original interface. Therefore, the new services are not privy to the implementation of the base services. For example, an add-in RPC system cannot access the communication buffers or the hardware directly. Just as with compilers, the operating system and micro-kernel writer has a deeper knowledge of the hardware and can tailor an implementation to match it.

Unfortunately, high-level operating system constructs suffer some loss of flexibility, or control. If the constructs match an application's programming model, they work well; if not, there can be problems. Low-level abstractions have the advantage of being able to serve as building blocks for higher-level services, while the reverse is not always true. For example, Mach's thread primitives are low-level, and language independent, but difficult to use. The C Threads run-time library raises the level of the threads abstraction and facilitates their use by C programmers. However, if only the C interface to threads were available, a Lisp programmer would be forced to twist his application to fit the provided model.

Since a micro-kernel is generally used for a broader range of applications than a programming language, flexibility is critical. Therefore, the selection of abstraction set must be done with care. Those high-level ideas with broad applicability are particularly attractive.

1.3. Implementation Options

High-level abstractions offer several interesting implementation options. The same services can be built in three different ways:

- Services provided on top of the kernel
- Add-ins to an existing kernel
- Kernels built directly on the hardware

The C threads package and the RPC implementation are examples of providing high-level abstractions on top of a lower-level ones. An add-in implementation adds abstractions and services either directly to the operating system kernel or as kernel extensions. The direct kernel approach allows the implementation to be optimized for the details of the hardware.

These three options demonstrate two important properties of high-level abstractions. They allow the interface to be separated from the implementation and they permit cooperation between dissimilar systems. The separation means that applications written for one implementation run without modification on another. This has been the driving force behind the wide-spread acceptance of open systems and is an essential element of any future operating system design. The interoperation opportunity is an important indirect benefit of high-level abstractions. It allows a new operating system design to coexist with more traditional approaches.

System software developers can learn a lot from language designers. Abstractions and services should be presented at a level high enough for programmers to use, yet low enough to permit higher-level constructs to be built on top of them. Such abstractions can provide multiple implementation options and enough room for efficient realizations.

1.4. The ARCADE Architecture

The trade-off between ease of use, performance and flexibility for a micro-kernel requires the same concern for computational models that is used in designing high-level languages. There is ample opportunity for efficiency without dictating the nature of the operating system above the micro-kernel. ARCADE, an ARChitecture for A Distributed Environment, presents abstractions which mirror the constructs found in many procedural languages. The architecture was originally realized as a micro-kernel and has subsequently been implemented as add-ins to widely used operating systems. This paper describes some of the experiences with the ARCADE high-level abstractions as well as their relationship to other work.

To optimize the trade-off between ease of use, flexibility and performance, a micro-kernel should be devised with the same concern for computational models that is used in designing high-level languages. This allows ample opportunity for efficiency without dictating the nature of the operating system above the micro-kernel. ARCADE, an ARChitecture for A Distributed Environment, presents abstractions which mirror the constructs found in many procedural languages. The architecture was original realized as a micro-kernel and has subsequently been implemented as part of other well-known, commercial operating systems. This paper describes some of the experiences with the ARCADE high-level abstractions, as well as their relationship to other work.

2. Micro-Kernels

Some of the benefits of high-level abstractions are particularly valuable in attaining the goals of a micro-kernel. A micro-kernel is defined by its top boundary, a hardware independent *architecture*. This specifies the conceptual structure and functional behavior of the layer as seen by the operating system. The architecture is a set of *abstractions*, generalized elements removed from implementation details, and the *services* which act on them. By making these abstractions high-level, we can achieve a good trade-off between performance and function.

One of the principal roles of a micro-kernel is to act as an insulating layer between hardware and system software, thus isolating each from changes in the other. This prevents the operating system from becoming hardware dependent and non-portable. By making no assumptions about the operating system, the micro-kernel can serve as the foundation for a variety of such systems.

Many micro-kernels present abstractions which are based on hardware models. In order to provide the portability desired of a micro-kernel, abstractions should make as few assumptions about the hardware as possible. Although such assumptions may improve performance, they limit portability. The abstractions should be able to exist on a wide variety of platforms.

For example, DSM implementations [Li86a] are often based on hardware features intended for paged virtual memory. Page faults handlers are defined on a shared region and page faults are used to ensure that writes are propagated to remote sites and that reads return fresh data. Such implementations are often difficult to port to systems which do not provide page-based virtual memory.

The computational model presented by the micro-kernel abstractions is available to both operating system and application programmers. Any ties the abstractions have to the hardware also tend to distract the programmer from the nuances of the application at hand. The programmer who wants optimum performance is often forced to deal with hardware issues. However, with a properly designed micro-kernel, the programmer deals with familiar computational concepts. Thus, since programming is normally done in a high-level language, the efficiency of high-level abstractions can be easier to attain.

In the DSM example, performance difficulties often result from *false sharing* and *thrashing*. False sharing occurs when independent sets of processes use shared data items that happen to fall on the same page, introducing contention for the page. Thrashing occurs when two processes on different machines alternately write to a common shared page. The operating system, in an attempt to maintain coherency, shuttles the page back and forth between the two systems to accommodate the alternating writes. Solutions to such problems do exist, but they invariably involve revealing the implementation in the abstraction interface [Ana92a]. Optimizations using programmer supplied help and programmer controlled distribution expect the programmer to directly deal with pages of memory, thus diverting the programmer's attention from the actual application.

Heterogeneity is also a reality in any modern interconnection. Abstractions that reveal hardware dependencies in their service interface do not provide an appropriate substrate for heterogeneity. Support for heterogeneity needs to be an integral component of the abstraction. This greatly increases its merit, applicability and portability. In the DSM example above, page size and data alignment differences can kill the ability of heterogeneous realizations of the abstraction to cooperate. Sun RPC, one of the most popular commercial distributed abstractions, integrates the use of the XDR protocol as a part of its service interface, thus allowing 680x0s to interact with SPARCs and i386s.

High-level abstractions also promote the interoperation provided by micro-kernels. A micro-kernel that provides location independence enables cooperation between two similar remote operating systems. To do this, it typically has an implicit interface to the communication subsystem. In addition, since the interface is hardware-independent, it automatically provides cooperation between dissimilar operating systems built on the same kernel. This interoperation may be local with multiple operating systems running above a single kernel, or remote also. High-level abstractions are implementation independent. They can be implemented upon a wide range of systems using multiple implementation options. In all cases, applications built using these abstractions can cooperate across the variant systems. The programmer has the ability to use the system with which he is most familiar while gaining access to the tools and services of others.

3. The ARCADE Architecture

The ARCADE architecture is designed to provide easy access to parallelism in heterogeneous interconnections. Its abstractions closely match a programmer's view of computation. It models the physical world as a set of resources and provides an event notification mechanism. The resource abstractions are orthogonal; defining a high-level view of data, a global identification tool and a classic execution thread.

Data units are a high-level data abstraction. Unlike simple memory with its raw bytes, they correspond to typed data as seen in procedural languages. Their size and type composition are specified when they are created, and they serve as the basis for a variety of data paradigms. They can be moved to serve as messages and can be shared to act as distributed shared memory. The type information allows automatic translation between heterogeneous machines. Lock services may be used to define synchronization points for flexible control of shared data coherency.

The *data unit link* abstraction is essentially a generalized global reference mechanism. It provides a system-managed, location-independent handle for data units residing throughout a distributed system. Data unit links correspond to pointers as seen in C. Since they span machine boundaries, data unit links can be used to build distributed dynamic data structures. The architecture specifies the data unit link interface, allowing optimization of its realizations. For example, it could be realized as a 64-bit global address in systems which provide such capabilities. When global addresses are not available, the data unit link may still be optimized to an address for local references. As an abstraction, additional properties may also be associated with data unit links. For example, in ARCADE, they include security provisions; they could be extended to include version identification, indexing and a variety of other functions.

The ARCADE *task* abstraction is essentially the same as other kernel-level active abstractions. It consists of a thread of execution that owns other resources. One resource is its *address space* which is composed of data units. Each task is given a globally unique, hierarchically structured name. For convenience, they are also given fixed-length *unique identifiers*, known as *UID*s, to facilitate identification. Every task also owns both a synchronous and asynchronous queue for receiving data units which have been transferred to them. Task properties also include the input/output line synchronization mechanism described below.

Input/output lines are a general event notification mechanism. They can be used to signal events such as the receipt of data units, changes in the state of locks or pulses in other task's output lines. They can also represent external events such as interrupts and timer ticks. A task's input lines control its run/sleep and live/die status through a configurable logic mechanism. Input/output lines can be used to simulate semaphores, monitors and other synchronization mechanisms.

4. ARCADE Services

In keeping with the high-level nature of ARCADE's abstractions, the services which act on them are also high-level. To a large extent, they mirror the kind of operations a programmer would normally use. To see this, we will examine the services associated with data units and data unit links.

A data unit is realized as a block of memory which can be mapped into a task's address space. A task creates a new data unit by invoking the `allocate()` service. Unlike conventional systems, ARCADE requires the caller to specify the *structure* of the data unit, rather than its *size*. Given the structure information, ARCADE maps a block of memory large enough to hold the specified data fields into the requesting task's address space. When a task no longer needs a data unit, it may invoke the release() system call, causing the kernel to unmap the associated memory block. Tasks can also obtain information about data units with qstruc() and qsize() which return the structure and size respectively.

The move() service corresponds to standard message passing. However, the abstract nature of data units yields some significant benefits. First, the data unit concept allows the communication and memory management paradigms to be merged. Tasks do not send messages to each other; instead, they transfer data units between their address spaces. The benefits of this approach, in terms of both performance and ease-of-use, have been demonstrated by systems such as Mach [For88a] and [Arm89a]. Second, the availability of structure information at run time allows data translation to be performed, and perhaps optimized, below the application level.

When requesting the move() service, a task specifies the address of the data unit to be transferred and identifies the destination task. The kernel unmaps the data unit from the sender's address space and, eventually, maps it into a free region in the destination's address space. This operation is similar to a message transfer in a more conventional setting.

For tasks whose most natural form of interaction is direct access to shared data, ARCADE provides the share() service. As with move(), a task calling share() specifies the address of a data unit and identifies the sharing task. However, unlike move(), the data unit is not removed from the caller's address space. By using share(), the sender's data unit mapping is retained, and both tasks are able to directly access the data unit.

In either case, a *notification packet data unit*, NPDU, is placed into either the destination task's *synchronous* or *asynchronous input queue*. The notification packet contains information about the transferred data unit including its origin, size and a data unit link pointing to it. The destination task uses the receive() service to transfer the notification packet to its address space. The NPDU is used to determine if access to the transferred data unit is desired and, subsequently, to access it.

In order to access the data unit itself, the receiver must use access(). This service has a data unit link as an argument and causes the target data unit to be mapped into the caller's address space. Once accessed, the data unit appears as directly accessibly memory. Access() is a general purpose data unit link de-referencing service, and it works with any data unit link. The process of receiving a notification packet and accessing its data unit link to acquire a transferred data unit is just a particularly important case.

The wait_du() service combines the receive() and access() operations into a single service. It allows a task to wait until a data unit arrives, receives the notification packet and accesses the data unit itself.

The data unit link, or DUL, can be used by tasks to build dynamically linked data structures. With the setlink() system call, a task can specify the target of a DUL, i.e. setlink() parallels a pointer assignment operation. When a target data unit is assigned to a DUL, the kernel maintains an internal, globally-unique identifier of the target. This identifier helps the kernel find the data unit when the data unit link is accessed. Data unit links have been found to be an effective way to construct dynamic data structures which can span machine boundaries. The entire structure can be stored in an innovative file system [Smi90a].

To alleviate the performance problems of DSM, we use an application-level locking scheme similar to, but less sophisticated than, that used by Amber [Cha89a]. Specifically, tasks may acquire and release locks on data units to prevent potentially dangerous concurrent access. In addition to ensuring correct program behavior, this technique reduces the impact of thrashing and completely eliminates false sharing.

Locks allow the programmer to regulate concurrent access to shared data units. Such regulation is frequently necessary to ensure the *consistency* of the shared data; in ARCADE, it also ensures *coherency*. ARCADE's transparent, update propagation protocol also includes provisions for a translation phase to accommodate sharing among heterogeneous machines. Locking services similar to those of classical database systems are provided: *read locks* and *write locks*. These locks can be used by applications to provide a variety of coherency schemes.

The `lockdu()` and `unlockdu()` services are used to control the consistency and coherency of a shared data unit. A task calls `lockdu()` specifying a *read lock* to acquire a shared lock on a data unit that is mapped into its address space. A task which successfully acquires this type of lock can safely proceed to read the contents of the locked data unit. No changes will be made to the data unit by other tasks that are participating in the lock-based concurrency control scheme.

Specifying a *write lock* allows a task to obtain an exclusive lock on a data unit. After acquiring such a lock, a task is guaranteed that no other task is holding either type of lock on the data unit. Internally, as with the read lock, the kernel ensures that the local replica of the data unit is up-to-date before returning control to the requesting task. A task with a write lock can safely modify the contents of the locked data unit. Changes made to the data unit are guaranteed to be propagated to all replication sites upon the locks release.

When a task wishes to release either a lock held on a data unit, it invokes the `unlockdu()` system call. Upon release of a write lock, the kernel propagates updates to remote replication sites. The current implementation actually uses an invalidation scheme. Remote sites *pull* the data after the invalidation; updates are not *pushed* by the source of the modifications. The update propagation is handled by the kernel thus shielding programmers from the details of tracking remote replicas.

Every ARCADE tasks controls a set of binary *output lines*. The first line is reserved and remains high throughout the task's life and effectively goes low when the task dies. A second reserved line goes high when the task's input queue is not empty. All other lines can be used to send event-like information to other tasks.

A task also owns a set of *input lines* which are used for synchronization and event notification. It can connect these to other task's outputs, to lines associated with data units, or hardware interrupts. The value of the input lines can be read with `readip()`. More importantly, they are the inputs to a simulated programmable array logic, *PAL*, which can be programmed to generate run/sleep and live/die controls. These controls can suspend the task or cause it to terminate.

This PAL mechanism can be used as a building block for classical synchronization primitives such as semaphores, monitors, etc. However, experience with this mechanism shows that it is difficult to use and ARCADE programmers have generally refrained from using it. The fact that this paradigm was too low-level and did not fit well with any programming language construct, resulted in a rather alien abstraction.

Thus, input/output lines and the PAL mechanism are not included in subsequent implementations of the data unit paradigm.

5. Experiences with ARCADE

To verify that the ARCADE architecture is implementation independent, it has been realized both as stand-alone micro-kernels and as extensions of existing operating systems. The micro-kernels run on the "bare iron" of 386-based workstations and in IBM System/370 virtual machines. These implementations support various operating systems as sets of ARCADE tasks. The ARCADE abstractions have allowed construction of a small development operating system, of a POSIX compliant environment and of a replicated distributed file system.

The orthogonal nature of the ARCADE abstraction set makes it possible to implement a subset, without compromising its functionality. Thus, the key abstractions, data units and data unit links, have been incorporated into three significantly different operating systems: VM/CMS, OS/2 and Mach. In VM/CMS, the abstractions were implemented as kernel extensions. For OS/2 and Mach, data unit services are provided by user-level servers executing above the operating system kernels. Names were added to the normal execution elements of these systems and their regular synchronization mechanisms replaced input/output lines. Data units and data unit links were then available to normal applications.

These diverse implementations of data units have clearly demonstrated their support for transparent interoperation between heterogeneous systems. Applications running in, say, OS/2 can share data units with Mach tasks or VM/CMS processes. Issues of communication and translation are all hidden below the interface. In fact, application source code dealing with data units is portable across all of these environments.

As a micro-kernel, ARCADE/386 has been called on to support a variety of system tasks. The first application of ARCADE was a basic operating system built on top of ARCADE/386 [Tra89a]. This Kernel Operating System (KOS) includes such services as device support, a file system, a command interpreter and a loader. It was built as a set of normal ARCADE tasks using the abstractions and services described above.

The original design of the ARCADE architecture integrated security features into each of the abstractions. These features conform to the most stringent security requirements. Thus, when the micro-kernel is combined with an encrypted communication subsystem, it is possible to build a provably secure operating system. Subsequently, a version of KOS was designed to address security needs [Bel90a]. The Secure ARCADE-Based Operating System (SABOS) follows the stringent security requirements of the "Orange Book" [Def83a].

KOS's device support allows multiple tasks to share physical hardware without interfering with each other. KOS's three main device support tasks are the console manager (*conman*), the file system (*filesys*), and timer (*timer*). The *conman* task maps a single *physical* console, consisting of a screen and keyboard, to multiple *logical* consoles, each with its own screen and keyboard. User tasks, even remote user tasks, may request logical consoles from *conman*. A task on one machine can manipulate a console on a second machine by using standard library routines.

The *filesys* task mediates user task requests for file access, converting them into a series of disk reads and writes. These are forwarded to the proper disk driver task. The disk format is the same FAT-based structure used by DOS and OS/2.

File access requests need not be only from local tasks. A task can operate on a remote file by dealing with the remote filesys task. The KOS file system can be viewed as a simplistic, location-dependent distributed file system. A user task can use the KOS *cfs* (change file system) command to direct its file system requests to the *filesys* task on any machine. A true location-independent distributed file system (DFS) has been built above KOS and its file system. DFS spans multiple machines and shields the user from file location concerns. It uses replicated files to provide fault tolerance and data units to support heterogeneity.

DFS extends the file abstraction by including structure information along with the data. Data units and their metadata are be saved and retrieved as units, rather than as streams of bytes. A file written by a machine of one architecture can be correctly read by a machine of another. An entire binary tree can be saved and retrieved without either the reader or writer knowing its configuration or size.

The basic ARCADE abstractions can also be used as building blocks for higher-level abstractions. A Nested Transaction Subsystem (NTS) has been implemented to add the transaction paradigm to ARCADE [Kul91a]. While the kernel provides *coherency* among data unit replicas, *consistency* constraints that span multiple data units must be handled above the kernel. NTS offers both *serializability* and *recoverability*. Serializability guarantees that interleaved accesses to data units by different transactions are equivalent to some serial order among the transactions. Recoverability allows partial changes to be undone and ensures that committed transactions are never lost. An NTS task uses blocking locks for serializability and DFS files allow recoverability by making data units persistent.

Two projects have evaluated language-level support for ARCADE abstractions. One, ABC, augmented standard C to hide ARCADE-specific details, while a second created support for object-oriented programming in ARCADE. ABC is implemented as a preprocessor which converts a superset of C into a combination of C code and ARCADE service calls [Ban90a]. It allows data units to be viewed essentially as simple C structures. The type information for the data units is automatically generated from the *struct* declaration by the preprocessor. In addition, data unit links appear as C pointers and the preprocessor inserts the required `access()` calls.

The object oriented environment based on ARCADE is designed for "programming in the large" [Bry89a]. Active objects are built on the ARCADE task abstraction. Five types of inter-object communication are seen by the programmer:

- `tell()` is a method invocation that does not expect a reply
- `submit()` is an invocation that blocks waiting for a reply
- `ask()` is a non-blocking invocation which expects a reply
- `reply()` is the response to `ask()` and `submit()`
- `forward()` lets its destination `reply()` to the original request.

A preprocessor translates each of these requests into ARCADE service calls. Thus, the programmer sees object oriented services which are really ARCADE-based.

6. Comparisons with Related Work

ARCADE makes several important contributions to the current discussion of distributed computing. Perhaps the three most significant are:

- The nature of distributed computing abstractions
- Structured heterogeneous distributed shared memory
- Security and resource identification mechanisms.

This section discusses these contributions and their relationship to ongoing work in distributed computing.

The ARCADE service interface is designed to support both system-software and user applications. Other systems, including Mach [Ace86a], Chorus and Amoeba [Mul90a], also hide the hardware but are intended to support only system-software, not applications. For example, Mach was initially designed as a micro-kernel to support 4.3 BSD. Although other operating systems have now been built on Mach, it is a difficult interface for applications. With Chorus, the *nucleus* normally presents its abstractions to a set of *system servers*, collectively called a *subsystem*. The subsystem, in turn, provides operating system services which are seen by application processes.

Micro-kernels have been used to realize portable distributed systems. Mach has been implemented on 80386s, 680x0s and SPARCs, and others, and Chorus is available on multiple platforms. However, while it is possible to mimic standard operating system services on top them [Gol90a], they cannot co-exist with standard operating systems. Their abstractions are close to the hardware and implementations typically need complete control.

There are systems that offer abstractions close to application programming constructs. Amber's passive and active objects allow the programmer to develop flexible object-oriented programs. Concert [Yem89a] extends standard programming languages such as C and PL/1 to implement heterogeneous cooperative peer-processing. Such systems are typically implemented on top of an existing operating system, Amber on Topaz and Concert on OS/2 and VM/370. However, they confine the programmer to a particular programming model, Amber with object-oriented programming and Concert with RPC.

The ARCADE abstractions strike a balance. Data units and data unit links are high enough for multiple cooperating implementations and low enough to support a variety of computational models.

Distributed shared memory has been implemented in hardware [Len90a], as operating system software and through compiler generated code [Bal89a]. The major DSM design issues are granularity of shared data, coherence protocol and support for heterogeneity [Nit91a]. Ivy classically assumes shared data is totally unstructured, using hardware-dependent page-based granularity. Linda's shared data is a *tuple space* [Ahu86a], defining application-dependent tuple-based granularity. Munin [Car91a] structures its shared data on the basis of variables in the source language. Most sharing schemes commit to either a page or a data object as a unit of granularity, but not both. However, depending on the data usage patterns, either approach may be best. Thus, it can be desirable to support both types of granularity.

This is only possible when, as in ARCADE, the abstraction-level unit of coherency is a data object.

Coherence protocols can be classified on the basis of *synchronization points* in a sequence of shared accesses. With Ivy's strict coherence, every read or write is a synchronization point. Munin's release consistency is based on *acquire* and *release* operations which are similar to ARCADE's lockdu and unlockdu. Clouds offers both strict and weak coherency.

Weaker coherency typically increases the concurrency of shared data accesses, but their use depends on the application's ability to tolerate stale data. Therefore, application specific coherence policies, can be more efficient. Applications can use ARCADE's advisory locks to control the coherency semantics. In fact, when it makes sense, applications can choose to ignore some synchronization points and use data that may be incoherent.

Several projects have extended the DSM abstraction a to heterogeneous environments. In Mermaid [Zho90a] for example, memory is shared in pages and a page contains data of only one type. Whenever a page is moved between two architecturally different systems, the data is converted to the appropriate format. Since the unit of coherency is a page, several restrictions apply. The size of each supported data type must be uniform and the translation process is not entirely transparent. Agora [Bis87a], on the other hand, provides a multi-language structured shared data facility which can span heterogeneous architectures. However, shared data is accessible only through a set of *access functions* and it may not contain references to other data objects. While these projects make important contributions, none offers the flexibility and ease of use available with data units and data unit links.

A fairly common resource identification and protection mechanism is a *capability* [Tan86a]. Mach, Amoeba, Chorus and Concert all support capabilities which contain encoded access rights to a resource. Not all capabilities are not context sensitive. A thread which possesses one may have the access rights, regardless of how it was obtained. They can be passed by one thread to another, allowing unconstrained access to the resource. Other identification mechanisms, such as global virtual addresses in Amber, have trouble supporting heterogeneity.

ARCADE associates security information with all of its abstractions, including data unit links. Data unit links are context sensitive, preventing illegal transfers of access rights. Once a task has access to a data unit, the data unit link can be optimized to a pointer.

7. Conclusions

ARCADE demonstrates the value of an implementation-independent architecture based on high-level abstractions. It shows that basing abstractions on a computational model can leave adequate room for performance optimization without dictating the method of computation. Data units and data unit links are particularly valuable for distributed applications and for cooperation between dissimilar systems. While it is not yet clear whether the ultimate micro-kernel will be superior to macro-kernels, ARCADE tells us a great deal about what that ultimate micro-kernel will be.

References

[Ace86a] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," in *Proceedings Summer Usenix* (July 1986).

[Ahu86a] S. Ahuja, "Linda and Friends," *IEEE Computer* **19**(8), pp. 26-34 (August 1986).

[Ana92a] R. Ananthanarayanan, "Application Specific Coherence Control for High Performance Distributed Shared Memory," in *Proceedings Symposium on Experiences with Distributed and Multiprocessor Systems* (1992).

[Arm89a] F. Armand, "Distributing UNIX Brings it Back to its Original Virtues," pp. 153-174 in *Proceedings Workshop on Experiences with Distributed and Multiprocessor Systems* (October 1989).

[Bal89a] H. Bal, "The Shared Data-Object Model as a Paradigm for Programming Distributed Systems," in *Ph.D. Dissertation, Vrije Universiteit* (1989).

[Ban90a] A. Banerji, "A New Programming Model for Distributed Applications," in *Master's Thesis, University of Notre Dame* (July 1990).

[Bel90a] M. Bellon, "A New Approach to Secure Distributed Operating System Design," in *Master's Thesis, University of Notre Dame* (April 1990).

[Bis87a] R. Bisianni, "Heterogeneous Parallel Processing: The Agora Shared Memory," in *Technical Report CMU-CS-87-112, Carnegie Mellon University* (March 1987).

[Bry89a] T. Bryden, "C Language Support for Object-Oriented Programming in ARCADE," in *Master's Thesis, University of Notre Dame* (November 1989).

[Car91a] J. Carter, "Implementation and Performance of Munin," pp. 152-164 in *Proceedings of the 13th ACM Symposium on Operating System Principles* (October 1991).

[Cha89a] J. Chase, "The Amber System: Parallel Programming on a Network of Multiprocessors," pp. 147-158 in *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (December 1989).

[Def83a] Department of Defense, "Department of Defense Trusted Computer System Evaluation Criteria," in *Department of Defence Computer Security Center, CSC-STD-001-83, Library No. s225,711* (August 1983).

[For88a] A. Forin, J. Barrera, M. Young, and R. Rashid, "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach," in *Technical Report CMU-CS-88-165, Carnegie-Mellon University* (August 1988).

[Gol90a] D. Golub, "Unix as an Application Program," in *Proceedings of the Summer USENIX Conference* (June 1990).

[Kul91a] D. Kulkarni, "Nested Transaction Support for Reliable Distributed Applications," in *Master's Thesis, University of Notre Dame* (July 1991).

[Len90a] D. Lenoski, "The Directory Based Cache Coherence Protocol for the DASH Multiprocessor," pp. 148-159 in *Proceedings of the 17th Annual Int. Symposium on Computer Architecture, IEEE* (1990).

[Li86a] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," in *Ph.D. Dissertation, Yale University, YALEU/DCS/RR-492* (September 1986).

[Mul90a] S. Mullender, "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer* **23**(5), pp. 44-53 (May 1990).

[Nit91a] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer* **24**(8), pp. 52-60 (August 1991).

[Smi90a] E. Smith, "A New Distributed File System Architecture for ARCADE," in *Master's Thesis, University of Notre Dame* (June 1990).

[Tan86a] A. Tanenbaum, "Using Sparse Capabilities in a Distributed Operating System," pp. 558-563 in *Proceedings of the 6th Int. Conf. on Distributed Operating Systems, IEEE* (1986).

[Tra89a] K. Tracey, "The Design and Implementation of an ARCADE-based Operating System," in *Master's Thesis, University of Notre Dame* (April 1989).

[Yem89a] S. Yemini, "CONCERT: A High-Level-Language Approach to Heterogeneous Distributed Systems," pp. 162-171 in *Proceedings of the Ninth International Conference on Distributed Computing* (June 1989).

[Zho90a] S. Zhou, "Extending Distributed Shared Memory to Heterogeneous Environments," pp. 30-37 in *Proceedings of the 10th Int. Conf. on Distributed Computing Systems, IEEE* (1990).

Delta-4 Fault-Tolerance and its Validation

David Powell

LAAS-CNRS
Toulouse, France
David.Powell@laas.fr

Abstract

The Delta-4 project developed a distributed fault-tolerant architecture featuring:

(a) A distributed object-oriented application support environment;

(b) Built-in support for user-transparent fault-tolerance;

(c) Use of multicast or group communication protocols; and

(d) Use of standard off-the-shelf processors and standard local area network technology with minimum specialized hardware.

This short paper gives a brief critical overview of the Delta-4 fault-tolerance techniques and their validation. A subject-classified bibliography is given for readers requiring a more in-depth description of Delta-4 concepts and mechanisms and their validation.

1. Introduction

It is a common observation that fault-tolerance and distribution are quite intimately related. First, should a single element of a distributed system fail, users expect at worst a slight degradation of the service that they are offered; distributed systems must thus at least have some built-in fault-tolerance. Such distribution-motivated fault-tolerance is thus aimed more at avoiding a decrease in dependability due to distribution rather than achieving significantly higher levels of dependability than in a non-distributed (fault-intolerant) system. Consequently, the main aim in such "fault-tolerant" distributed systems is to tolerate the common situation in which nodes in the system have become silent or "crashed" (e.g., due to a local power outage). More rarely, attention is also given to situations in which the distributed system may become partitioned. In this case, the objective is to avoid the inconsistencies that may occur should the various partitions be allowed to continue operation as "independent" systems.

This work was supported from March 1986 to January 1992 by the CEC Esprit programme (projects 818 and 2252). The consortium for project 2252 consisted of Bull SA (France), Crédit Agricole (France), Ferranti International (UK), IEI-CNR (Italy), IITB-Fraunhofer (Germany), INESC (Portugal), LAAS-CNRS (France), LGI-IMAG (France), MARI (UK), SRD-AEA Technology (UK), Renault (France), SEMA Group (France) and the University of Newcastle (UK). The opinions presented here are those of the author.

On the other hand, systems that are specifically designed to achieve high dependability by means of fault-tolerance can always be regarded as being distributed at some level or another since it is impossible to achieve fault-tolerance without redundancy. The redundant elements are distributed in the sense that they must constitute independent fault containment regions and must interact in such a way as to achieve a given level of service although some of them may be faulty. Such fault-tolerance-motivated distribution is therefore guided by a specific requirement to improve dependability with respect to a fault-intolerant (non-distributed) system. Consequently, much attention is paid to the definition of the components that are the elements of distribution (redundant fault containment regions), the way in which these components can fail and the means by which they are interconnected. Since the motivation in this case is high dependability (rather than just distribution), it is therefore common to admit that components may fail in a more arbitrary fashion than just by going silent. Furthermore, the interconnection scheme is often purpose-designed to be itself fault-tolerant to avoid partitioning.

In the Delta-4 Esprit project, the aim was to investigate how both objectives – distribution and fault-tolerance – could be pursued simultaneously to define an *open* and highly *dependable* distributed architecture for money-critical (as opposed to life-critical) applications. We wanted to be able to reap the potential modularity and multi-vendor support advantages of open distributed systems and to take advantage of distribution in order to provide useful levels of dependability by means of fault-tolerance. The challenge was to find a way of linking together standard, possibly heterogeneous, off-the-shelf computers with user-transparent fault-tolerance implemented in software to minimize the need for re-design of specialized hardware during technology updates.

2. Failure Mode Assumptions and Hardware Architecture

To achieve high dependability, we did not want to restrict ourselves systematically to the assumption that nodes were "fail-silent" in the sense mentioned in the introduction. For such an assumption to be justifiable to a high degree, it would be necessary to impose that all nodes included in the system would be equipped with extensive self-checking to ensure local error-detection with high coverage and low latency. Such a requirement is hardly compatible with our desire to accommodate standard off-the-shelf node hardware.

The least restrictive assumption about the way that nodes can fail is that they are "fail-uncontrolled", i.e., that they do not possess any local error-detection mechanisms and can thus produce quite arbitrary or even malicious behaviour. In particular, a fail-uncontrolled node may:

(a) Omit or delay sending (some) messages,

(b) Send extra messages,

(c) Send messages with erroneous content, or

(d) Refuse to receive messages.

Unfortunately, if complete nodes can fail in such an arbitrary fashion then the interconnection scheme must be made much more complex than the single (or possibly, duplex) broadcast channel that would be sufficient for fail-silent nodes. For example, a fail-uncontrolled node connected to multiple channels could fail by saturating all channels, thus bringing down the complete system. Furthermore, protocols for ensuring agreement under such a failure mode assumption are notoriously complex and time consuming.

Therefore, the Delta-4 architecture is based on a hybrid approach whereby each node is split into two sub-systems:

- An off-the-shelf computation component, called a *host*, that may be fail-uncontrolled;
- A communication component, called a *network attachment controller* (NAC), that is assumed to be fail-silent.

The fail-silent assumption for the network attachment controllers alleviates the problems stated earlier regarding agreement protocol complexity and the use of broadcast channels. In Delta-4, the NACs of each station are interconnected by a standard LAN (8802.4 or 8802.5). Duplex (or even simplex) channels have been shown to be sufficient for achieving a very low probability of communication system failure in the maintainable environments for which Delta-4 is intended. The communication system is therefore considered as hardcore and no attempt is made at the application level to tolerate physical network partitioning.

The network attachment controller consists of a pair of piggy-backed cards that plugs into the host's back-plane bus and interfaces the node with the physical communication channels. The NAC is very similar to any other standard LAN controller card; the only difference is that it uses built-in hardware self-checking to substantiate the assumption that it is fail-silent. Self-checking is achieved by standard duplication and comparison techniques for the main processing part of the NAC in conjunction with a watchdog timer. A hardware-implemented memory protection scheme is also used to prevent corruption of the NAC memory by some fail-uncontrolled behaviour of the host. Duplication could not be carried out in the low-level interface to the network due to the impossibility of synchronizing the specialized LAN-specific VLSI components; coverage of faults in this part of the NAC therefore relies on the built-in error detection capabilities of these components.

3. Fault-Tolerance Mechanisms

The NACs are the only specialized hardware components in the Delta-4 architecture; the rest of the Delta-4 fault-tolerance and functionality is achieved by system software implemented either on top of the hosts' local operating systems or on the NACs' real-time kernels. The system software consists of three parts:

- A host-resident infrastructure for supporting distributed computation;
- A computation and communication administration system (executing partly on the hosts and partly on the NACs);
- A multipoint communication protocol stack (executing on the NACs).

A particular host-resident infrastructure for supporting open *object-oriented* distributed computation was developed for the Delta 4 architecture: the *Delta 4 Application Support Environment* (Deltase). According to the philosophy of "open" distributed processing, Deltase facilitates the use of heterogeneous languages for implementing the various objects of a distributed application and allows the differences in underlying local operating systems to be ' idden (in practice though, all the implemented Delta-4 prototypes were based on UNIX). Deltase provides the means for generating and supporting interactions between run-time software components called "capsules" (executable representations of objects).

Fault-tolerance in Delta-4 is achieved by means of replicating capsules on separate nodes. Delta-4 applications can be made incrementally fault-tolerant on a service-by-service basis; at application configuration time, the application designer can choose which services he wishes to make fault-tolerant and to which degree. Extensive use is made of group or multipoint communication protocols that enable capsule interactions to be programmed without heed for the degree of capsule replication.

The design of the Delta-4 fault-tolerance mechanisms makes a clear distinction between error processing and fault treatment. Error-processing aims to remove errors from the computational state, if possible before the occurrence of a failure in the service delivered by the system. It involves the coordination of interactions between replicas using error detection and recovery or compensation to mask the fact that one (or more) of the underlying nodes may be faulty. Subsequent fault-treatment is aimed at preventing faults from being activated again. It can be seen as self-repair facility that identifies and passivates faulty nodes and, by creating new replicas, can allow software components to survive further faults (within the limits of available hardware resources).

Delta 4 provides three different – but complementary – techniques for coordinating replicated computation: active, passive and semi-active replication. The *active replication* technique relies on the assumption that replicas supplied with the same input messages are deterministic in the absence of faults. This technique must be used when hosts are fail-uncontrolled; it allows messages produced by replicas to be cross-checked in value and time to detect and compensate errors produced by a minority of replicas. This cross-checking is carried out through an inter-replica protocol on the basis of message signatures rather than complete messages. The active replication technique can be optimized when is is justifiable to assume that hosts are fail-silent (value error detection is no longer necessary). The other two replication techniques can only be used when hosts can be assumed to be fail-silent. The *passive replication* technique relies on a primary/standby approach in which the primary checkpoints its state to the standby replica(s) whenever it sends an output message. At the expense of a decrease in performance, this approach has the advantage of not requiring computation to be deterministic. The *semi-active replication* technique is a hybrid approach that seeks to achieve the low recovery overheads of active replication while relaxing the constraints on computation determinism. This approach is based on a replica group consisting of a leader replica and one or more follower replicas. The follower replicas carry out exactly the same computation as the leader replica except when some non-deterministic decision must be made; when this occurs, they wait until the leader instructs them as to the decision that he took at that point in the computation.

Any error detected locally (by a NAC or a host) results in the node disconnecting itself from the communication network. Any error detected remotely (i.e., when active replication is used) is reported to the administration system that then passivates the incriminated host by instructing its NAC to remove itself from the network. The administration system then attempts to carry out reconfiguration by cloning new replicas on other nodes to replace those that were resident on the faulty node.

This involves choosing the nodes on which new replicas are to be created, setting up template replicas, initializing them by copying and transferring the state of cohort replicas on unaffected nodes, and then re-synchronizing each reconfigured replica group.

Since any detected error results in node passivation, this approach to fault treatment treats all faults as if they were permanent faults. However, the mechanisms could be modified to provide an improved treatment of temporary faults (widely known to be much more common than permanent faults) which affect only a single replica (or even a subset of replicas). In this case, the replica cloning operation could be used to re-initialize the replica(s) on the same node instead of passivating the node and cloning all resident replicas to new nodes.

4. Validation

Validation – both from the verification viewpoint (removal of faults in the specification, design and/or implementation) and the evaluation viewpoint (quantification of the provided dependability and performance) – should be carried out at each step in the process of producing a “dependable” system. At the specification stage, validation consists essentially of verifying that the system specifications are consistent both with each other and with the requirements of the intended application domains. This “informal” verification was carried out in Delta 4 by a peer review process during scientific and technical committee meetings and during project reviews. More tangible validation activities were carried out during the design and implementation phases. Ideally, all components of a system should be extensively validated. However, for the money-critical (as opposed to life-critical) applications for which Delta 4 is intended, it was decided to restrict the validation to the most important (or the most critical) sub-systems.

Design validation is centered on descriptions or models of the future implementation. Its purpose is (a) to verify that these models are consistent with the specifications and (b) to evaluate (predict) some characteristics (e.g., performance, dependability) of the future implementation. Two design validation activities were carried out:

- *Protocol verification* aimed at removing faults in the protocol design was carried out on two protocols in the Delta-4 multipoint communication stack: the basic atomic multicasting protocol and the inter-replica protocol for active replication. Inconsistencies of different nature were detected such as incorrect initializations of local variables, state machine transition conditions that were too weak, etc. Other inconsistencies, such as unspecified receptions, non termination of certain protocol phases and message duplications, were only detectable in some peculiar sequence of events that it would be unlikely to obtain by simulation. Implementations of these protocols were derived from the formal specifications.
- *Dependability evaluation* work was carried out with a view to quantifying the dependability actually achievable by the Delta 4 architecture. The initial work concentrated on the communication system and demonstrated the sufficiency of simplex or duplex channels since undependability was mainly dominated by the lack of coverage of the NAC self-checking mechanisms. Later work considere' the availability and reliability of a simple banking application using various Delta 4 fault-tolerance models.

One important conclusion of this work was a demonstration of the importance of input and output configurations on the overall dependability.

Implementation validation is centered on *testing* actual prototype versions of the architecture instead of on models. Like design validation, its purpose is two fold: (a) to verify that the implementation provides the specified functionality and (b) to evaluate (measure) some characteristics of the actual implementation. Implementation validation included three aspects:

- *Performance testing* was carried out to assess the overheads of software-implemented fault-tolerance. In particular, the overhead due to the cross-checking activity of the inter-replica protocol for active replication was assessed by measuring the round-trip delay of a null-RPC (a client sends a message to a replicated server that replies immediately with the same message). As was to be expected, the signature-comparison technique that is used causes the overhead to decrease rapidly with message size. With very small messages (a few bytes) the overhead is in the order of 150%, which is lower than the 200% minimum that would be expected when comparing complete messages from a triplicated server. With large messages (several tens of thousands of bytes), the overhead drops to about 30%.
- *Software reliability evaluation* was carried out on many of the major software subsystems of the architecture. Static testing tools were used to identify important characteristics of the implemented software so as to focus the testing effort on the components that were likely to be the most unreliable. In addition, failure data was collected during the software development and testing phase in order to predict the rate at which it can be expected that residual design/implementation faults will cause the system to fail when in operational use. Unfortunately, however, the amount of data collected was insufficient for any useful estimations of software reliability to be made.
- *Fault injection* (into the prototype hardware) has been used as a means for validating (a) the self-checking mechanisms of the Delta 4 network attachment controllers (NACs) and (b) the implementation, on these NACs, of the Delta-4 atomic multicasting protocol. Fault injection is a technique for testing fault-tolerant systems *in the presence of the very faults they are meant to tolerate.* In addition to identifying implementation faults (and residual design faults), fault-injection also enables the measurement of the effectiveness of the built-in error detection and fault-tolerance mechanisms by means of coverage, dormancy and latency estimations. As a result of these fault-injection experiments, several improvements were made to the NAC self-checking mechanisms and the atomic multicasting protocol implementation with demonstrable evidence of reliability growth.

5. Conclusions

Several lessons can be learnt from the distributed software-implemented approach to hardware fault-tolerance adopted in Delta-4:

- The performance of the system, especially in the presence of faults, is evidently an important aspect to be taken into account when comparing the approach with hardware-intensive tightly-synchronized approaches. Still, the performance overheads are quite reasonable and are acceptable price to pay when considering the flexibility advantages of software-implemented fault-tolerance.

- Distributed techniques for fault-tolerance like those used in Delta-4 are also capable of tolerating those software design faults that cause replicas to fail independently. However, this advantage should be carefully weighed against the complexity of the additional system software that is necessary for fault-tolerance. Unless one is careful, design and implementation faults in this additional software can have very detrimental effects on system dependability. It was unfortunate that insufficient software failure data could be collected during the project's lifetime for it to be able to assess the trend in system software reliability. Nevertheless, the achievement and assessment of reliability of the additional system software would clearly be facilitated in a pure development framework rather than in a pre-competitive project such as Delta-4.
- Another possible disadvantage of distributed fault-tolerance implemented by software concerns the interface that is offered to application software. Re-utilisation of code written for a non-fault-tolerant system is often difficult and sometimes impossible. Even new code should be written respecting special rules if the choice of replication technique is be left open (e.g., to ensure replica determinism). One possible way round this problem, other than resorting to tightly-synchronized redundant hardware, may be to implement the fault-tolerance software on top of a micro-kernel together with servers providing a "standard" operating system interface. The use of micro-kernel technology could also greatly simplify the checkpointing and cloning operations if the execution contexts of replicas can be clearly identified and managed.

The Delta-4 project nevertheless showed successfully that software-implemented mechanisms can be used to tolerate hardware faults in off-the-shelf distributed computing nodes. A little specialized hardware support can also allow the range of accommodated host failure modes to be considerably wider than in approaches that prefer to ignore hardware details by means of simplifying assumptions that are often unjustified.

Bibliography

For further information,

- General [Pow91a].
- Architecture and Fault-Tolerance [Pow88a, Spe89a, Bar90a, Tul90a, Pow91b, Sea91a, Che92a].
- Security [Bla90a, Des91a].
- System Administration [Bon89a, Bu91a].
- Communication [Ver88a, Ver90a, Ver92a].
- Validation [Arl90a, Arl90b, Bap90a, Kan91a, Kan91b, Avr92a].

References

[Arl90a] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault-Injection for Dependability Validation – A Methodology and Some Applications," *IEEE Transactions on Software Engineering* **16**(2), pp. 166-182 (1990).

[Arl90b] J. Arlat, M. Aguera, Y. Crouzet, J. Fabre, E. Martins, and D. Powell, "Experimental Evaluation of the Fault Tolerance of an Atomic Multicast Protocol," *IEEE Transactions on Reliability* **39**(4), pp. 455-467 (1990).

[Avr92a] D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet, "Fault Injection for the Formal Testing of Fault Tolerance," pp. 345-354 in *Proc. 22nd Int. Conf. on Fault-Tolerant Computing Systems (FTCS-22)*, IEEE, Boston, MA, USA (July 1992).

[Bap90a] M. Baptista, S. Graf, J.-L. Richier, L. Rodrigues, C. Rodriguez, P. Verı́ssimo, and J. Voiron, "Formal Specification and Verification of a Network Independent Atomic Multicast Protocol," in *Proc. 3rd Int. Conf. on Formal Description Techniques (FORTE'90)*, ed. J. Quemada, J. Mañas and E. Vazquez, North-Holland (1990).

[Bar90a] P. A. Barrett, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Verı́ssimo, L. Rodrigues, and N. A. Speirs, "The Delta-4 XPA Extra Performance Architecture," pp. 481-488 in *Proc. 20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, IEEE, Newcastle upon Tyne, UK (June 1990).

[Bla90a] L. Blain and Y. Deswarte, "An Intrusion-Tolerant Security Server for an Open Distributed System," pp. 97-104 in *Proc. European Symp. on Reseach in Computer Security*, AFCET, Toulouse, France (October1990).

[Bon89a] G. Bonn, U. Bügel, F. Kaiser, and T. Usländer, "Management of the Delta-4 Open, Distributed and Dependable Computing System," pp. 573-584 in *Proc. IFIP TC6/WG6.6 Symp. on Integrated Network Management*, ed. B. Meandzija and J. Wescott, Elsevier Science Publishers BV (North Holland), Boston, MA, USA (May 1989).

[Bu91a] U. Bügel and B. Gilmore, "Process Cloning in Delta-4," pp. 179-189 in *ESPRIT Information Processing Systems and Software – Results and Progress of Selected Projects*, CEC-DGXIII, Brussels, Belgium (1991). XIII/372/91.

[Che92a] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron, "Active Replication in Delta-4," pp. 28-37 in *Proc. 22nd Int. Conf. on Fault-Tolerant Computing Systems (FTCS-22)*, IEEE, Boston, MA, USA (July 1992).

[Des91a] Yves Deswarte, Laurent Blain, and Jean-Charles Fabre, "Intrusion Tolerance in Distributed Systems," pp. 110-121 in *Proc. Symp. on Research in Security and Privacy*, IEEE, Oakland, CA, USA (May 1991).

[Kan91a] K. Kanoun and D. Powell, "Dependability Evaluation of Bus and Ring Communication Topologies for the Delta-4 Distributed Fault-Tolerant Architecture," pp. 130-141 in

Proc. 10th Symp. on Reliable Distributed Systems (SRDS-10), IEEE, Pisa, Italy (September-October 1991).

[Kan91b] K. Kanoun, J. Arlat, L. Burrill, Y. Crouzet, S. Graf, E. Martins, A. MacInnes, D. Powell, J.-L. Richier, and J. Voiron, "Delta-4 Architecture Validation," pp. 234-252 in *Proc. Esprit Conference*, CEC-DGXIII, Brussels, Belgium (November 1991).

[Pow88a] D. Powell, G. Bonn, D. Seaton, P. Verı́ssimo, and F. Waeselynck, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems," pp. 246-251 in *Proc. 18th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18)*, IEEE, Tokyo, Japan (June 1988).

[Pow91a] D. Powell, "Delta-4: a Generic Architecture for Dependable Distributed Computing," pp. 484 in *Research Reports ESPRIT*, Springer-Verlag, Berlin, Germany (1991). (Editor).

[Pow91b] D. Powell, M. Chérèque, and D. Drackley, "Fault-Tolerance in Delta-4," pp. 122-125 in *ACM Operating Systems Review* (April 1991).

[Sea91a] D. Seaton, P. Barrett, P. Bond, and P. Verı́ssimo, "The Delta-4 Extra Performance Architecture and Real-Time," pp. 203-215 in *ESPRIT Information Processing Systems and Software – Results and Progress of Selected Projects*, CEC-DGXIII, Brussels, Belgium (1991). X111/372/91.

[Spe89a] N. A. Speirs and P. A. Barrett, "Using Passive Replicates in Delta-4 to provide Dependable Distributed Computing," pp. 184-190 in *Proc. 19th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-19)*, IEEE, Chigago, MI, USA (June 1989).

[Tul90a] A. Tully and S. K. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs," pp. 104-113 in *Proc. 9th Symp. on Reliable Distributed Systems (SRDS-9)*, IEEE, Huntsville, AL, USA (October 1990).

[Ver88a] P. Verı́ssimo, "Redundant Media Mechanisms for Dependable Communication in Token-Bus LANs," pp. 453-462 in *Proc. 13th Local Computer Network Conf.*, IEEE, Minneapolis, USA (October 1988).

[Ver90a] P. Verı́ssimo and J. A. Marques, "Reliable Broadcast for Fault-Tolerance on Local Computer Networks," pp. 54-63 in *Proc. 9th Symp. on Reliable Distributed Systems*, IEEE, Huntsville, AL, USA (October 1990).

[Ver92a] P. Verı́ssimo and L. Rodrigues, "A Posteriori Agreement for Fault-Tolerant Clock Synchronization on Broadcast Networks," in *Proc. 22nd Int. Conf. on Fault-Tolerant Computing Systems (FTCS-22)*, IEEE, Boston, MA, USA (July 1992).

User and Mechanism Views of Distributed Systems Management

Jonathan D. Moffettt

Department of Computer Science
University of York
jdm@minster.york.ac.uk

Morris S. Sloman

Department of Computing
Imperial College, London
mss@doc.ic.ac.uk

Abstract

This paper discusses the concepts developed within the DOMINO project on Domain Management for Open Systems, and how these concepts are implemented. Domains are a means of grouping objects, distinct from the management policies which are specified in terms of domains. Domains and policies are discussed from the viewpoint of both the manager and the underlying mechanisms which implement them. The emphasis of the user view is on conceptual clarity and the emphasis of the mechanism view is on efficient implementation in distributed systems. Both views need to be implemented, although in some cases there may be a direct correspondence between the two views. The paper argues that keeping managers independent from the domain of objects they manage gives flexibility and simplifies both the user and mechanism views of system management.

> *F.E Smith, the famous advocate, once saw two housewives haranguing each other from upstairs windows across a street. "They will never agree." he observed, "They are arguing from different premises."*

1. Introduction

There is not yet general agreement on the concepts which are needed for the automation of Distributed System Management (DSM). This paper presents a set of concepts for specifying management policy as defined and implemented in the Domino project. Discussions with other research groups and within standard organisations has indicated confusion between the user interface view of management concepts and the underlying implementation mechanisms. Both views are needed to allow users to express their management requirements while enabling efficient implementation of the mechanisms.

The Domino project on Domain Management in Open Distributed System is a collaborative project funded by the UK Information Engineering Directorate and involves Imperial College, SEMA Group and British Petroleum. This project is concerned with managing very large distributed processing systems which typically consist of multiple interconnected networks and span the computer systems belonging to a number of different organisations. They cannot be managed from one central point, but management has to be achieved by negotiation between independent managers who wish to cooperate while retaining their autonomy.

We have identified two separate generic activities of managers in systems:

- Performing **management operations** for monitoring or controlling the behaviour of object related to a management function such as configuration or accounting.
- Creating, interpreting and monitoring **policies**. Policies are different from management operations. A management operation is an instantaneous activity, whereas a policy is intended as a persistent means of influencing operations. Large scale systems may have millions of objects so it is impractical to specify policies for individual objects. Policies need to be specified and applied to a set of objects as discussed in more detail below.

It was the need to be able to group objects in order to specify a common policy for them which led to the introduction of **management domains**[†] as a key concept in the Domino project. A domain is an object which represents a set of objects which have been explicitly grouped together to apply a common management policy. A domain is also used as a means of structuring the namespace for objects.

The concept of grouping objects should not be confused with that of encapsulation. Hierarchical composition can be used to construct a composite object from several primitive or other composite objects [Mag89a]. The composite object is viewed as a single object for the purposes of invoking operations and the interface of the composite object hides its internal structure from the user. The component objects are then said to be encapsulated. Encapsulation is an essential concept for coping with the complexity of distributed systems and for building management systems. Although it is sometimes useful to apply a management operation to a set of objects, in many cases the managers have to perform management operations on the individual objects, which encapsulation prevents. Domains provide grouping but not encapsulation. Section 2 of this paper distinguishes between the user and mechanism views of management, and justifies the need for the coexistence these views in a system. The views are related to the domain concept in section 3 and to policies in section 4. Section 5 shows how domains can represent users and manger positions. Section 6 discusses some of the related work on domains, and section 7 relates our approach to Open Distributed Processing (ODP) viewpoints.

[†] Abbreviated to *domain* in this paper.

2. User and Mechanism Views

One aspect of DSM which has taken some time to emerge is the importance of distinguishing between a manager's view of an enterprise which is often reflected in the human interface to the management system and the underlying mechanisms used within the system for implementing this view. The following examples will help to illustrate this.

2.1. Object Based Approach

The emergence of graphical interface technology has made the object based approach to human interfaces very popular. Objects are represented by icons which can be selected by means of a mouse, and the operations applying to an object can be selected from a menu for invocation on the object. Different icons are used to represent object types, and instances are distinguished by means of names. This approach is supported by object oriented environments which provide the means to define object types, create/delete instances of objects and for objects to invoke operations on each other. The manager's view is thus one of obtaining information on the objects they are managing and performing management operations on them to change their behaviour.

In some cases this view may not be able to map onto the underlying mechanisms. The object being managed may be a hardware component which does not support an object invocation mechanism compatible with that of the manager object. It is then necessary to "front" the managed resource with an adapter which provides a representation of the resource which is compatible with that of the manager. The approach of a managed object separate from the resource object is part of the OSI Management model [ISO91a] and has been built into both their user and mechanism view even though it is inappropriate for software objects, which can be managed directly. Another justification given for managed objects being independent from the resources they represent is that the managed object can provide an abstraction of the resource for management purposes and hide the resource's normal functionality. Our approach (Figure 2.1) simply assumes a managed object may have multiple interfaces such as a management interface which defines the management operations it supports, an interface for interaction with a file service, plus other interfaces to support application-dependent functionality.

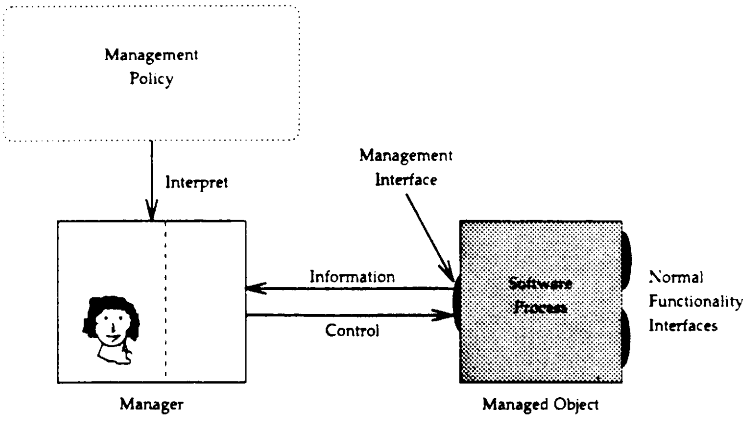


Figure 2.1: *Domino Management Interactions*

There is another example of low level mechanisms which confuses the higher level object interaction view in the OSI Management Model. The concept of a local agent in a computer is a common means of implementing remote invocations; the agent receives an invocation message from a remote manager and then performs the invocation on a local object. Again this has been built into the OSI management model rather than being considered a transparent implementation mechanism. If the OSI model is represented directly in an object based implementation a manager would invoke an operation on a remote manager agent, which would invoke an operation on a local managed object, which would finally invoke an operation on the resource being managed, rather than the manager directly invoking operations on a managed resource object (see Figure 2.2).

2.2. Management Domains

Typical graphical window interfaces to computer systems provide the concept of a directory or folder to contain files or programs. The directory is a naming context so that object names need only be unique within a directory. The human manager needs a similar view for grouping objects in order to apply a common management policy. A domain provides this means of grouping objects for management purposes. From the manager's viewpoint, domains contain named objects, but it is necessary to permit objects to be members of multiple domains to reflect overlapping responsibility or different types of management responsibility for the same object. For example the manager responsible for the security of an object may be different from the manager responsible for maintenance of that object.

The underlying implementation mechanism for domains has no concept of containment of the object itself. Instead the domain holds a reference to an object and provides a mapping from a text name string to an internal object identifier and address (see section 3.3). Users refer to objects by names or using a pointing device, while the system refers to them by object identifiers.

2.3. Access Rules

Access rules specify discretionary access control policy in terms of the set of operations which any of a set of users is authorised to perform on any of a set of target objects. This also is a concept which users find intuitively easy to deal with, but does not implement efficiently. It would involve unreasonably long searches to evaluate access requests if the system had to search through all access rule objects in order to

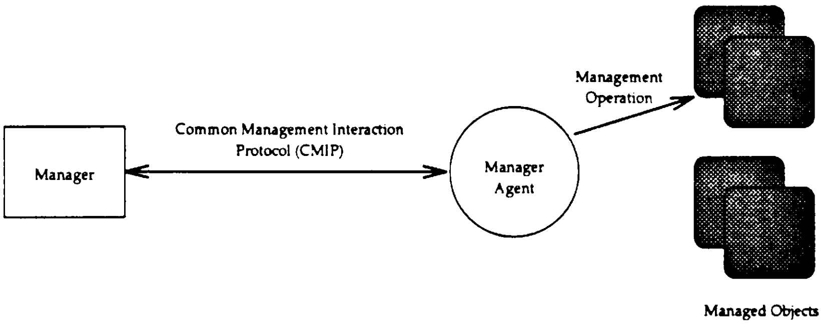


Figure 2.2: *OSI Management Interactions*

decide whether to allow the request. Therefore, for the Domino project, we are implementing the access rules by means of Access Control Lists (ACLs) attached to target object domains. The user sees and makes policy in terms of access rule objects, but the system maps these access rules onto ACLs, which it uses as the mechanism to interpret policy when it receives an access request. See Section 4.

2.4. Users

Although users of a computer system may be human, managers manipulate user objects representing registered users within the computer system. These user objects define the resources allocated and the services accessible to the user. There are substantial advantages to be gained from a mechanism in which user objects are implemented as a special kind of domain, which we call a **user representation domain** (URD). The URD represents the persistent aspects of a user from one logon to the next. When the user logs on, process objects which temporarily represent the user become members of the URD and gain the privileges and authority which belong to that user. When the user logs off, the processes cease to be members and no longer have the users privileges. In this case the user interface view of user objects does not use the concept of a domain at all, but the system uses the mechanism view of a domain.

We believe that it is important to maintain both a distinction and a relationship between the user interface and mechanism views of management concepts. As with any system, the user requirements must drive the mechanism, but it must be subject to the condition that it can be implemented efficiently. The user interface and mechanism views in Domino both have to be implemented; human managers have to be able to work with objects which are implemented in accordance with their view, while the underlying system has to be able to use a mechanism which provides an efficient implementation of the functions seen by the manager.

3. Domains

We have already observed the need to be able to group and structure objects for management operations and policies. In practice, it is an essential aspect of this grouping that it should be hierarchical. There are two reasons for this. The first is that, as illustrated in the examples below, many existing special-purpose management structures use hierarchical structuring, and so any generic method of grouping must be able to reflect this. In addition, hierarchical structuring methods have come into existence because it is, in practice, impossible to manage large numbers of objects in a purely linear way, hence the widespread use of tree structures of objects in computer systems. Exactly the same reasoning applies to management policies; a means of hierarchical grouping of objects must be integrated into any system for defining policies.

Two familiar examples of hierarchical grouping are the description of organisations in a hierarchical manner, and the grouping of files into hierarchical directories. Figures 3.1 and 3.2 show these examples expressed traditionally; Figure 3.1 is an enterprise view of its personnel structure, while Figure 3.2 is a file directory structure. Both are typi-:ally, but not necessarily, associated with hierarchical policy statements, e.g. “members of an organisation have authority over those

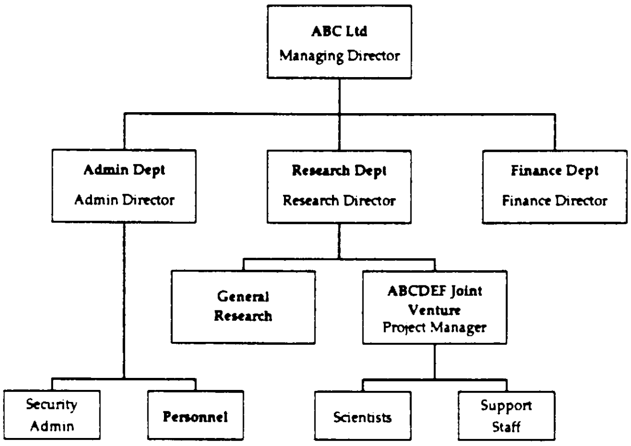


Figure 3.1: *Organisation Tree*

below them in the hierarchy" or "access to a directory gives access to all its subdirectories".

In order to represent these concepts we have defined **domain** objects, with attributes which include a **policy set** and a set of **constraints**. Their purpose is to group objects together for management purposes, while not affecting the objects' normal functionality. Domains are described in [Slo89a].

3.1. The Policy Set and Subdomains

The policy set of a domain consists of an enumerated set of member objects to which the policy associated with the domain applies. From the point of view of a user it would have been quite feasible to define domain membership in terms of a predicate on object attributes, e.g. "the set of all users under 65 years age". However determining the set of members would require the underlying mechanism to examine every object within the system to check whether it met the constraints of the predicate. Also tracking current domain membership as the attributes of distributed objects change would incur very heavy overheads (see our comments on constraints below). This is an example of problems with the mechanism view which forced us to drop a concept from the

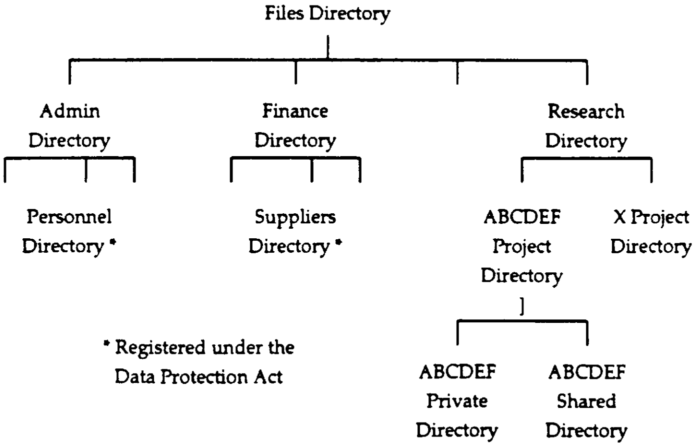


Figure 3.2: *Directory File Structure*

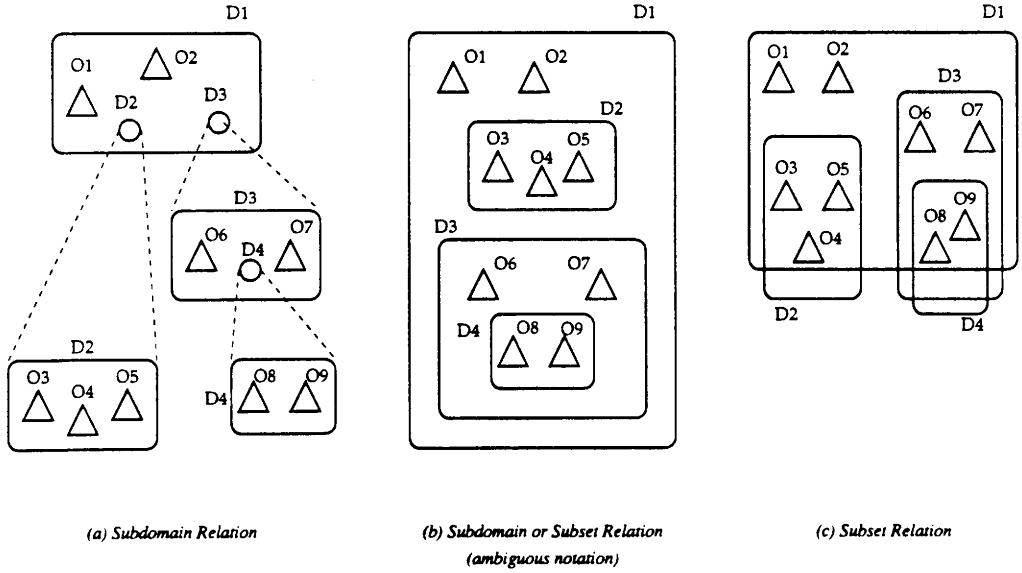
user view. However, another motivation for the decision is that it avoids potential logical paradoxes, such as Russells Paradox: is the class of all classes which are not members of themselves a member of itself? This definition cannot be expressed by us and therefore the paradox cannot arise.

There are some definitions of "domain" which do not make it clear whether the members are defined by enumeration or predicate. For example the ISO Security Frameworks Overview [ISO92a] says: "A security domain is a set of elements under a given security policy administered by a single authority for some specific security relevant activities." We are given no indication of whether its members are defined by the predicate "under a given security policy ..." or whether they are an enumerated set of objects to which a policy is to be applied.

The policy set achieves a simple grouping facility by enumerating the set of objects which are **direct members** of the domain. Membership of domains has no effect on the state of the member objects. However, the power of domains for structuring is provided by the concept of **subdomains** which are domain objects that are members of other domains. In order to understand this, it is useful to explain the subdomain, overlap and subset relationships between domains.

If one domain object is a member of another, the first is referred to as a **subdomain** of the second. In Figure 3.3a, we refer to D2 as a **direct subdomain** of D1, and to D4 as an **indirect subdomain** of D1. An object is a **direct member** of a domain if it is in the domain's policy set. An object is an **indirect member** of a domain Dx if it is a member of a domain Dy which is a subdomain of Dx. Note that indirect members of a domain are not enumerated in the domain's policy set. When we refer to a "member" (unqualified), we mean a direct or indirect member. If an object (domain or other) is a direct member of a domain, then the domain is its **parent**, and if it is a member, then the domain is its **ancestor**.

Two domains **overlap** if there are objects which are members of both domains. A special case of overlapping occurs when the objects in one



(a) Subdomain Relation

(b) Subdomain or Subset Relation (ambiguous notation)

(c) Subset Relation

Figure 3.3: *Subdomain and Subset Relations*

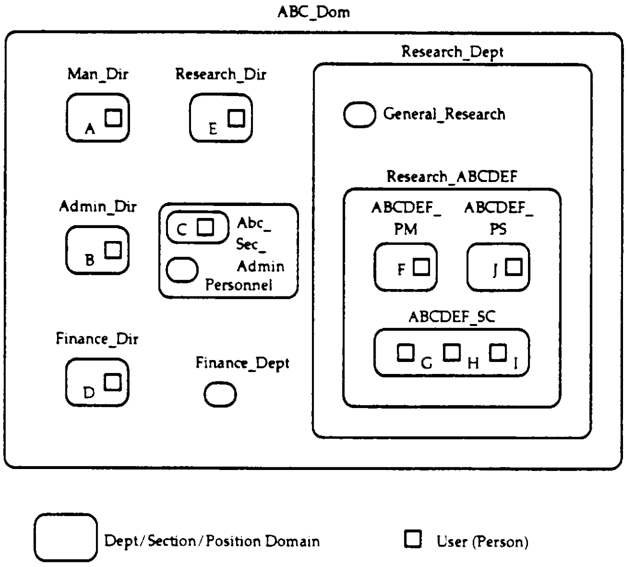
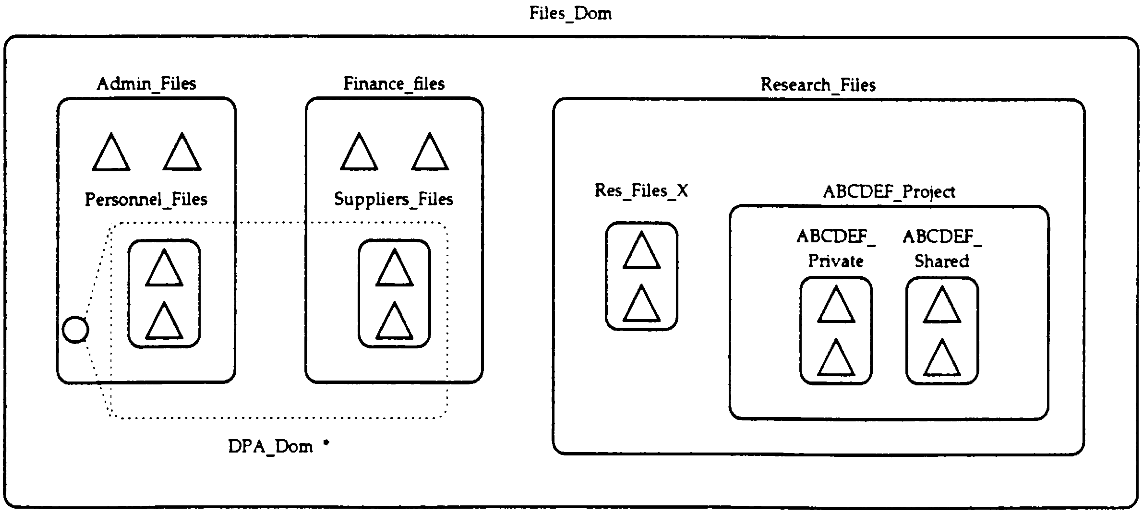


Figure 3.4: *Domain Representation of Organisation*

domain are a **subset** of the objects in another. This is illustrated in Figure 3.3c.

An alternative notation for showing a subdomain relation is shown in Figure 3.3b. It is simpler, but ambiguous, as it can also be seen as a subset relation. We use the "shorthand" notation of Figure 3.3b in this document to denote the subdomain relation, for compactness. Figures 3.4 and 3.5 show the examples of Figures 3.1 and 3.2 expressed in terms of domains in this way.

In a static situation the evaluation of the indirect membership of a domain hierarchy such as shown in Figure 3.3 would yield the same overall set of objects whether the domains were subsets or subdomains. However, the effect of removal or inclusion of an object in D2 will



* Domain of files registered under the Data Protection Act

Figure 3.5: *Domain Representation of Directory File Structure*

have different results for the evaluation of the membership of D1, depending on the sort of relation.

If D2 is a *subset* of D1, as in Figure 3.3c, then the addition of an object O10 to D2's policy set does not affect D1 in any way and there is no resulting relationship between O10 and D1. After the operation, D2 is no longer a subset of D1, but D1 and D2 overlap. If D2 is a *subdomain* of D1, as in Figure 3.3a, then O10 becomes an indirect member of D1 and of all parent domains of D1.

Figures 3.4 and 3.5 show the domain representation of the hierarchies shown in Figures 3.1 and 3.2. In Figure 3.4, Research Dept is a subdomain of ABC Ltd and General Research is a subdomain of Research Dept, so when a user becomes a member of General Research he automatically becomes an indirect member of Research Dept and ABC Ltd. However, if the relationships were subset relationships instead, it would be necessary to perform three separate operations to include the user in the three different domains. For this reason subsets are not a particularly useful relationship for structuring management.

The ISO Security Frameworks Overview defines "subdomain" as a subset rather than a membership relationship, which makes hierarchical structures difficult to express.

3.2. Constraints on Membership Operations

We envisage the constraints attribute of a domain being used to place constraints upon the membership of the domain. Some examples of possible application-specific constraints are:

- There must be at least two members of the domain of Security Administrators.
- Processors in a domain must be able to support the M68000 instruction set.
- Destroying an object is not possible if the object is still a member of another domain.

There are of course two possible kinds of constraint. The first, which we favour, is a constraint upon the operations which affect domain membership, typically the **create** and **destroy** operations for any object type and the **include** and **remove** operations on a domain. These add and subtract objects to and from the policy set. The predicate defined by the constraint has to be evaluated once only, when the operation is performed. Constraints on the number of members, object types or other object attributes can be enforced by constraints of this kind. The second kind of constraint is a general predicate, which the system is required to maintain, about the attributes of members of a domain. We regard this as being potentially very difficult to achieve, because every time any application functional operation attempts to change an object's attribute, the system is required to verify that the domain constraints, of every domain of which the object is a member, are not violated. We do not therefore envisage users being permitted to specify constraints of this kind. We still have a lot of work to do on constraints.

3.3. User and Mechanism Views of Objects and Domains

There is no incompatibility between the user view of objects being contained in domains and the mechanism view of actually holding references. The mechanism view is in fact what is implemented, and the user view is provided by filtering out the invisible mechanical features

User View		Mechanism View
Object Name	<-------->	OID
Parent Relationship	<-------->	Parent Set
Other Attributes	<-------->	Other Attributes

Figure 3.6: *User and Mechanism Views of Objects*

and presenting objects in terms of names, not object identifiers. This is a summary of the differences:

> In the users view, objects are referred to by local names (within a domain) or a domain path name plus local name. In the mechanism view they are referred to by unique identifiers, through addresses. An OID (Object ID), consisting of its address and identifier, is associated with each object.

The decision to take this approach for the mechanism was in order to avoid possible inconsistencies when an object is a member of several domains; there is only one copy of the object, upon which all operations are performed. The approach achieves consistency at the cost of having to perform remote accesses when the object is remote from one of its parent domains. A different approach is being considered by the DOMAINS project [DOM91a], where even in the mechanism view it will be a local shield object which is encapsulated with a manager within a domain. The shield object may cache state information relating to the remote managed object. This requires mechanisms to maintain consistency of the distributed and replicated state information relating to a managed object which is a member of multiple domains. Our experience indicates that the complexity and costs in maintaining consistency far outweighs the benefit of local caching of the comparatively small amount of management information needed for a managed object.

Policies relate to domains, and in order to determine the policies which apply to a particular object, it is necessary to know its ancestors. It is impractical to inspect all domains for this purpose, so in our implementation the domains service maintains a **parent set** for each object, which is a list of its parent domains, enabling derivation of its ancestors. The parent set is the implementation mechanism for the parent/ancestor relationship of the user view.

The relationships between the user and mechanism views of objects and of domains is illustrated in Figures 3.6 and 3.7.

User View		Mechanism View
Object Name	<-------->	OID
Parent/Ancestor Relationship		Parent Set
Policy Set { Name }	<-------->	Policy Set { (OID, Local Name) }
Constraints	<-------->	Constraints

Figure 3.7: *User and Mechanism Views of Domains*

3.4. Domain Expressions

Domains and subdomains are a powerful means of expressing hierarchical membership and set union, but provide no means of expressing other basic set operations such as Set Difference and Set Intersection. Set Difference can express sets of objects such as "all files in Payroll_Files except Payroll_Master". Set Intersection can express sets such as "all files which are in Payroll_Files and also in Personal_Data". We have therefore introduced **domain expressions**, which allow the expression of a set of objects in a policy by means of a formula containing the standard set operations.

It would be possible to use domain manipulation operations to create a new domain with the required membership e.g. create DomA with the same members as (DomB ∩ DomC), and refer to DomA in the policy. However, the policy then applies to membership of DomB and DomC at the time DomA is created and not at the time that its applicability is checked. Static enumeration of objects at some point in the past is not usually what is required in evaluation of policies, and domain expressions in a policy such as an access rule are evaluated at the time the rule is checked.

4. Policies

We have recently presented a generic user view of policies [Mof91a] which identifies their essential characteristics and models them as objects with the following attributes: **subjects** who are motivated or authorised, depending on the **modality** of the policy, to achieve **goals** on target **objects**. While we believe that this may be a good model to develop, in Domino we so far only have one kind of policy object which is modelled, and this is an **access rule**. The user view of an access rule is straightforward. It is an object with four attributes:

- **Subject Domain**, which is a domain expression defining the set of subjects of the policy.
- **Operation Set**, a set of permitted operations.
- **Target Domain**, which is a domain expression defining the set of target objects of the policy.
- **Constraints**, which limit the applicability of the access rule, and which we currently see as using generally available information such as date, time of day and the terminal at which the current user is logged on.

An access rule, shown in Figure 4.1, maps exactly onto policy objects in which the modality is positive authorisation and the goals are operations in the interface of the target objects of the access rule. There is a good reason for users having a view which shows them access rules, rather than generic policies, and this is that they actually think in terms of making rules which authorise access or remove authorisation, rather than in terms of making generic policies. We therefore envisage any generic model of policies which may emerge in future as a framework for ensuring that a consistent user interface is provided rather than a view which the user sees directly.

The actual representation of authority to perform management operations in an organisations hierarchy is not by the domain hierarchy, but by access rules which give users that authority. The contribution of domains to that authority is that, while access rules determine the oper-

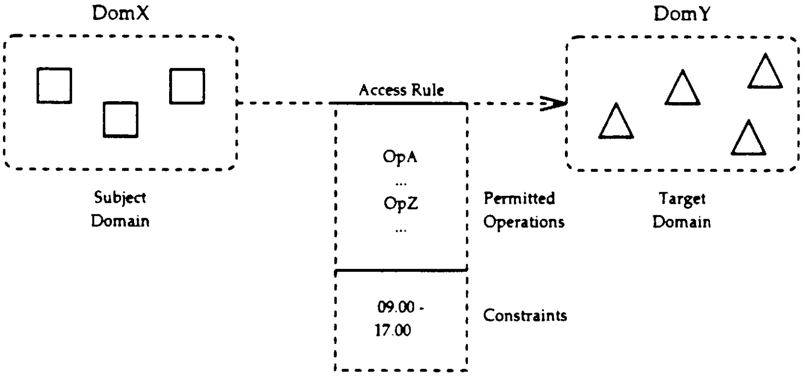


Figure 4.1: *An Access Rule*

ations which can be performed, the target domains of the rules determine the **scope** of that authority.

The meaning of an access rule is straightforward. It states that any user object in the set defined by the User Domain of the rule is authorised to perform any of the operations in the Operation Set on any of the objects in the set defined by the Target Domain of the rule. There is a fixed policy that in the absence of authorisation by an access rule, an operation request (attempted access) is forbidden. In the user view of the system, whenever a user issues an operation request, the system must search every access rule in the system, allowing the request if an access rule matching it is found, but forbidding it if none can be found. This is clearly completely impractical as a mechanism.

We therefore use two implementations of access rules: an **access rule** object corresponds to the user view (Figure 4.2) and is the means by which users specify access control policy. The underlying mechanism which controls access uses Access Control List (ACL) entries for each domain (and its subdomains) which is specified in the Target Domain expression of the access rule (for implementation efficiency the lowest level of granularity for target objects is a domain). The **Access Control Entries (ACEs)** consist of the User Domain expression and the Operation Set of the access rule object. Every operation request carries with it an authenticated list of the domains which are ancestors of the

User View		**Mechanism View**
Object Name	<-------->	OID
User Domain (Name Based DE[†])	<-------->	Combined to form an ACL entry which is an attribute of every domain affected by the Target Domain expression.
Operation Set	<-------->	
Constraints	<-------->	Not yet implemented
Target Domain (Name Based DE)	<-------->	Every domain affected by the Target Domain expression holds an ACL entry.

Figure 4.2: *User and Mechanism Views of Access Rules*

[†] DE – Domain Expression.

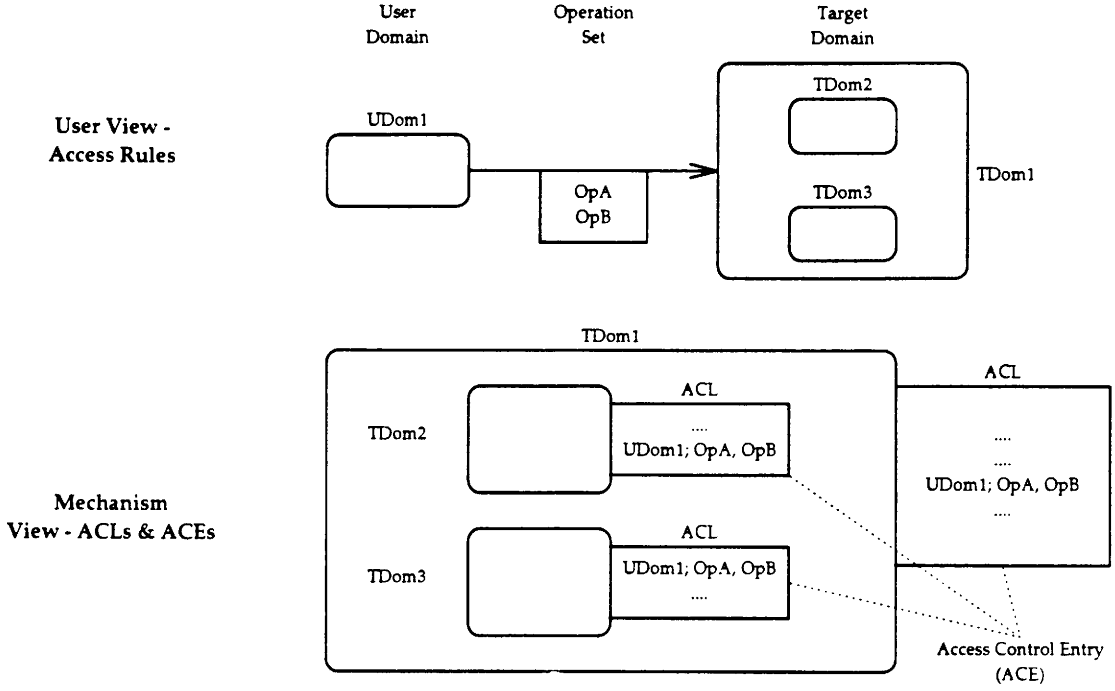


Figure 4.3: *An Access Rule, and its Corresponding Access Control Lists and Entries*

user object which makes the request. The system can therefore evaluate the request by reference solely to the information in the operation request and the parent domains of the target object. Details of this implementation can be found in [Twi92a]. The relationships between the user and mechanism views of access rules is illustrated in Figures 4.2 and 4.3. Note that the access rule applying to TDom1 in Figure 4.3, propagates to subdomains TDom2 and TDom3 and so appears as entries in the ACL of each domain.

There is clearly a problem of ensuring consistency between the views of access rules. Ideally, at the design stage, we should have taken our formal specification of the user view [Mof91b] and either refined it to the mechanism view while ensuring the preservation of properties or created a formal specification of the mechanism view and proved that the properties were preserved. However, in the absence of tools to support these activities we have opted for an informal design of the mechanism view.

There also a problem of ensuring consistency between views during operation, as the two views can become inconsistent through system errors and failures. We are therefore developing consistency checking and restoration tools in order to diagnose and recover from any inconsistencies.

5. Representing Users and Manager Positions

Two default domains shown in Figure 5.1 are needed to represent human users within the system:

i) **A User Representation Domain** (**URD**) is a persistent representation of the human user or manager. When the user logs into the system an user interface object is created within the URD and inherits all access rules specified for the URD.

ii) A **User Personal Domain** (**UPD**) corresponds to a user's home directory and represents the personal resources which the user "owns". In addition the user may have limited access to other service domains representing the shared resources the user can access.

In most cases policies should be expressed not in terms of individual users but in terms of **manager positions** which they occupy so that when the human manager is transferred to another position, the access rules pertaining to the positions do not have to be changed. Since positions are "occupied" and domains have members, the correspondence is so close that, even at the user view, we regard it as satisfactory to see positions as being represented by domains, with constraints on membership operations to ensure that only user objects can be members and that there are the appropriate limits on the number of members. As far as practical, all access rules are specified with respect to a Manager Position Domain (Figure 5.2). Allocating a human manager to a position is accomplished by including his/her URD within the position domain. The manager automatically inherits all rights for that position and may be a member of multiple position domains if performing multiple management roles.

6. Alternative Views of Domains

One feature of several other approaches to the definition of management domains stands out. This is a definition of domain which includes both the manager and the objects which are managed [ISO92a, DOM91a]. It may also include an implicit or explicit definition of the management operations or goals in the domain. It then enables policy

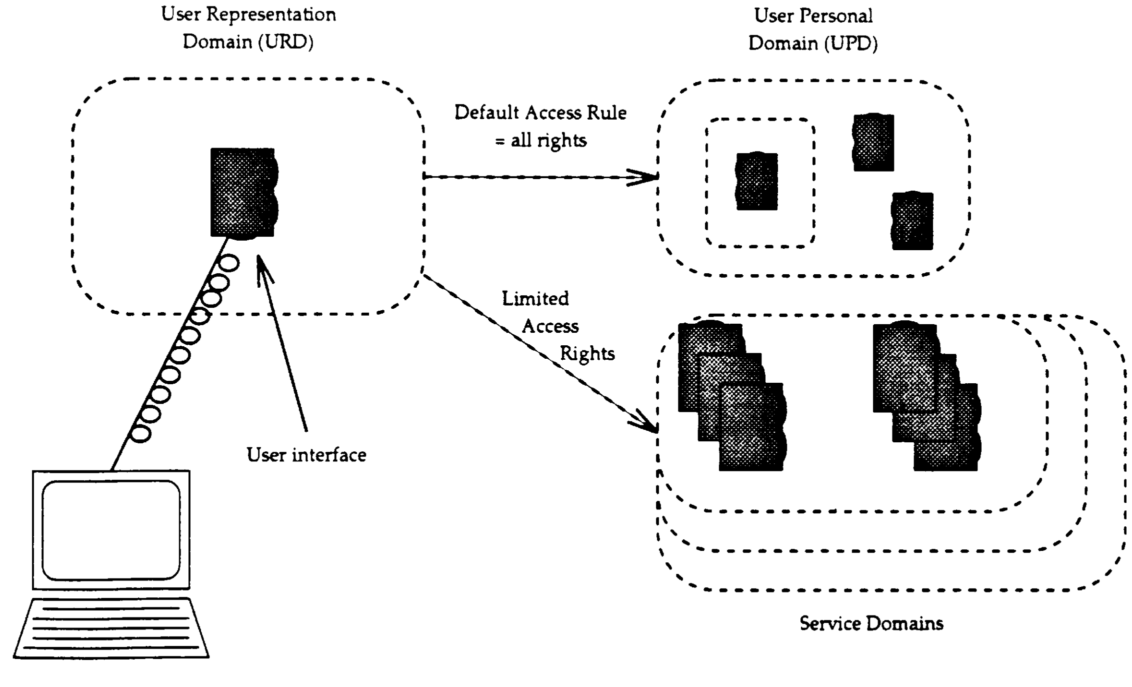

User Representation Domain (URD)
User Personal Domain (UPD)
Default Access Rule = all rights
Limited Access Rights
User interface
Service Domains

Figure 5.1: *Default User Domains*

to be expressed in a single object, without the need for a separate policy object. Also, in the DOMAINS approach the domain encapsulates the manager and objects.

Domains should not be defined to include managers as well as target objects, for the following reasons:

- Managers in one organisation may be given (limited) management rights over objects in other organisations. A domain which includes managers from one organisation with objects from another organisation leads to policy and implementation problems. The manager or the managed object has to become a member of a domain "owned" by another organisation. What are the policy implications of this? Does the access control system allow the operation? What if an external manager is only to be allowed a limited subset of the management operations which are available to local managers?
- Encapsulating managed objects plus managers in a domain makes it very difficult to permit the objects or managers to be in multiple domains. Implementing a "shared" encapsulated object is not easy as the encapsulated object is usually dependent on the encapsulating object for its existence. In some circumstances it may be appropriate to encapsulate a manager with the objects it manages and treat this composite object as a single entity. However the model should permit, but not enforce it.
- A single manager object may be responsible for managing multiple independent sets of objects and so would have to be included in multiple domains.
- There is a need to specify policy in terms of the user's position in an organisation, not the individual user, and so a new concept of position has to be invented.
- There is sometimes a need to treat managers as managed objects in a domain hierarchy which is independent from the domain hierarchy of the objects they manage. How can this be achieved if managers are in the same domains as the managed objects?

We have no doubt that these questions can be solved, but only at the cost of additional complexity. We believe that our simple concept of domain and the use of access rules to indicate the relationship between managers and managed objects makes it much easier to represent the flexible relationships shown in Figure 6.1.

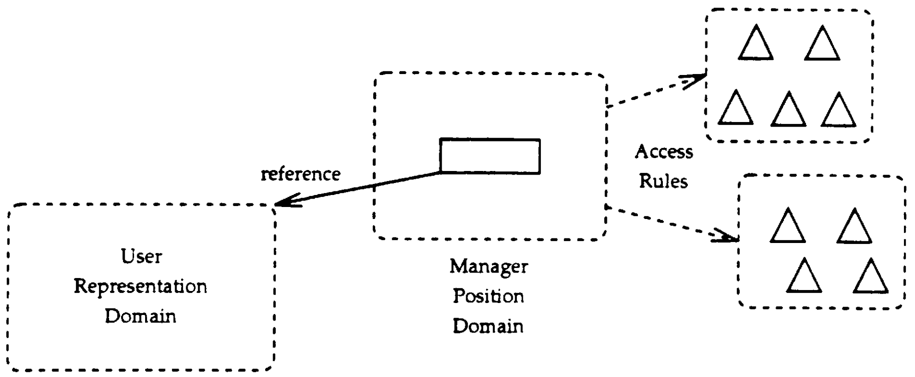


Figure 5.2: *Manager Positions*

The DEC Enterprise Management Architecture [Str91a] uses a domain to define a "sphere of interest of a set of managed objects for a manager" – managers are not part of the domain. Only the managers are aware of the domains and objects do not know which domains they are members of. This model is compatible with the Domino approach as it implements a subset of our concepts. There are no domains of managers and relationships between a manager and domain are implicit in the knowledge of the domain name rather than explicitly shown by the use of access rules. They do not use the domains for access control. Domain membership information is held by the name service.

7. ODP Viewpoints

The ISO Open Distributed Processing (ODP) work [Lin91a] defines the following viewpoints on distributed processing:

- **Enterprise viewpoint** – this describes the overall objectives of a system in terms of roles, actions, goals and policies. Many of the user view concepts we have discussed in this paper e.g. domains, users, manager positions, policy would have to be modelled in the enterprise viewpoint.
- **Information viewpoint** – this provides a framework to describe the information requirements and information flows of a system. It does not have to differentiate between parts that are to be automated, or performed manually.
- **Computation viewpoint** – this provides a framework for modelling the operations necessary to automate information processing. The mechanisms required to support this model are specified in the engineering projection of the system.
- **Engineering viewpoint** – this provides a framework for describing how to mechanise an application.
- **Technology viewpoint** – this provides a framework for describing the technical artifacts from which a distributed system is built.

Our user interface view has points in common with both the information and computation viewpoints. It describes the information which a user is dealing with, but it is also explicitly an automated view, because it is the user interface to a system.

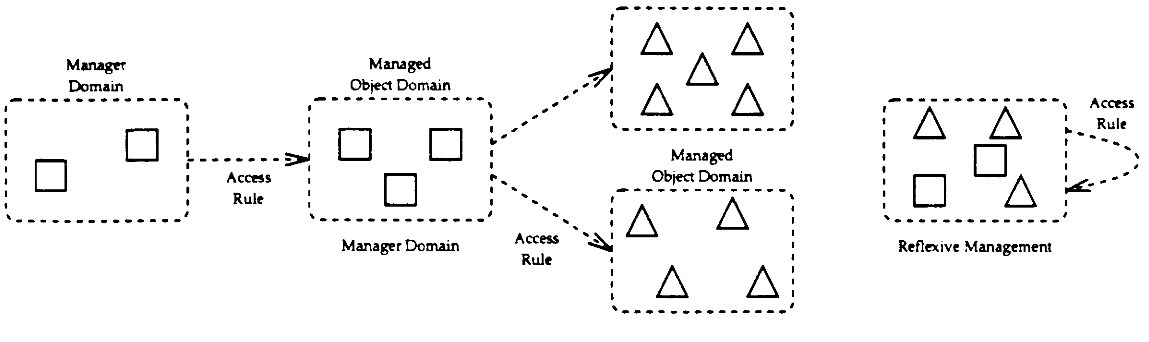


Figure 6.1: *Typical management relationships*

The obvious parallel for our mechanism view is the engineering viewpoint. However, the mechanism view does not provide additional functionality for the user interface view, such as the transparencies which are provided as a service by the engineering viewpoint to the computational viewpoint. All that it does is provide identical functionality with better performance. Extra functionality, such as transparencies, could also be provided by it – we plan to provide replication of domains for fault tolerance – but need not be.

We believe that our identification of a user interface view is an essential aspect of our system modelling which is not recognised explicitly by ODP. A corollary of this view is a consideration of the performance implications which then forces a mechanism view upon us. If this were to be implemented on an ODP platform, it is probable that we would choose to represent both the user interface and the mechanism views with the computational viewpoint. This is the highest level at which the viewpoint is explicitly automated, and would enable the transparency services provided by the engineering viewpoint to be available to the implementations of both our views.

8. Conclusions

There are several tasks to be performed by users carrying out Distributed System Management. One of these is grouping and structuring large numbers of objects. We achieve this by our concept of Domain objects, allowing hierarchical structuring by allowing domains to be members (subdomains) of other domains. Another task is to define the actions which managers are motivated and/or authorised to do, and the objects which are the targets of their management. This is done by the definition of a generic Policy object, of which Access Rules are a specialisation. There are, of course, other tasks which we have not yet even recognised, which will be revealed by methodically analysing the power and limitations of our present concepts.

Work on implementation in the Domino project has shown that it is essential to have separate coexisting user and mechanism views of some of the concepts, both of which are implemented. The emphasis for the user view is conceptual clarity while the emphasis on the mechanism view is efficient implementation. There are substantial differences between the user and mechanism views of Domains, while retaining the same basic structure. On the other hand the structure of the mechanism for Access Rules is completely different from the object which the user sees.

We have developed an approach to Distributed System Management concepts which separates the different concerns of managers into two main types of object which they can work with: Domains and Policies. We have further separated these into User and Mechanism views. This has enabled us to address our work to the correct concern and at the correct level, in order to make sound progress.

Acknowledgements

Acknowledgement is made of the contribution to the above concepts by our colleagues in the DOMINO projects, and in particular Kevin Twidle who has been responsible for much of the implementation of the domain service. We gratefully acknowledge the support of the DTI/SERC under grant GR/F 35197.

References

[DOM91a] DOMAINS, *Basic Concepts, version 2.0, Esprit Project 5165*, Philips Gmbh, Aachen, Germany. Nov 1991.

[ISO91a] ISO 10040, *Systems Management Overview*, ISO/IEC JTC1/SC21 WG4. 1991.

[ISO92a] ISO 10181-1, *Security Frameworks Overview*, ISO/IEC JTC1/SC21 N6693. January 1992.

[Lin91a] P. Linington, "Open Distributed Processing and Open Management," pp. 553-562 in *Integrated Network Management II*, ed. I. Krishnan and W. Zimmer, North Holland (1991).

[Mag89a] J. Magee, J. Kramer, and M. S. Sloman, "Constructing Distributed Systems in Conic," *IEEE Transactions on Software Engineering* **15**(6), pp. 663-675 (June 1989).

[Mof91a] J. D. Moffett and M. S. Sloman, "The Representation of Policies as System Objects," *SIGOIS Bulletin*, Atlanta, USA **12**(2 & 3), pp. 171-184, Proc Conf on Organisational Computer Systems (COCS 91) (Nov 1991).

[Mof91b] J. D. Moffett, *Delegation of Authority for Access – A Formal Model*, Domino paper A2/IC/4.1. revised May 1991.

[Slo89a] M. S. Sloman and J. D. Moffett, "Domain Management for Distributed Systems," *Proc of the IFIP Symposium on Integrated Network Management*, Boston, USA, pp. 505-516, North Holland (May 1989). Domino paper A1/IC/1.

[Str91a] C. Strutt, "Dealing with Scale in an Enterprise Management Director," *Integrated Network Management II*, pp. 577-593, North Holland (1991).

[Twi92a] K. P. Twidle and J. D. Moffett, *Domino Reference Manual*, Domino paper A3/IC/4. June 1992.

The Distributed Computing Environment Naming Architecture

Norbert Leser

DCE Architect
Open Software Foundation
nl@osf.org

Abstract

The OSF Distributed Computing Environment (DCE) constitutes a set of technologies which have been selected through an open acquisition process and integrated into a coherent DCE architecture. This paper discusses the architecture of DCE while focusing on the architecture of the name service as one of the key elements.

The notion of the **cell** is introduced and described. The cell is an essential property of the DCE architecture. While providing a means for modeling a distributed system, cells go beyond being a formalism for describing the system. This paper elaborates on the role and purpose of the cell, the semantics of the cell namespace, the placement in a large worldwide spanning system, the implications of security in a distributed system, and other related architectural properties of DCE. All this will be discussed in conjunction with other architectural approaches of modeling distributed systems.

The research and engineering work on this paper is based on a leading involvement with this project during the last 4 years. It includes studies and evaluation of worldwide available technologies and research projects, architectural, design and developmental work on these technologies and experience with further usage of this system, in particular the integration of the object oriented distributed management technology, using DCE Naming and Security Services.

1. The Key Elements of a Distributed System

The DCE architecture was determined and developed by analyzing currently available technologies and by defining the inherent properties of distributed systems [Les90a, Joh91a]. Essential requirements are the needs for integrating networked systems into a coherent distributed systems environment and for solving system growth from homogeneous parallel processing systems into distributed systems [Bir89a].

The architecture of a distributed system cannot make any assumptions on homogeneity due to the diversity of the distributed environment. One has to deal with **heterogeneity** on all levels, from different machine types, various networks and communication protocols, up to service levels. Multipurpose services may be sufficient for most applications but for various reasons, be it competition or be it the need for especially tailored services, the distributed systems architecture must be capable of adopting diverse technologies.

Another significant area which needs attention in architecting the distributed system is the need for **unlimited growth** of the system. The entire connected distributed system must be sufficiently scalable worldwide and may have to deal with millions of nodes.

1.1. The Core Distributed Services and the Role of Naming

In creating an architecture for the distributed computing environment we were faced with the requirement that the traditional modeling of single systems on one hand and purely network and communications technologies on the other hand are not sufficient to provide a coherent view of the distributed system. Thus, we developed an architecture which provides for an integration of a well defined set of **core services**, creating an infrastructure for programming and using the facilities of the distributed system. These services include technologies for Remote Procedure Call (RPC) based communication, security, a provider for a truly distributed time, and a naming system [OSF91a, OSF91b]. Figure 1 shows the relationship of the DCE technology components.

Naming was always considered to play a central role in meeting the goal of providing transparency and homogeneity to the consumers of the system by hiding the inherent complexity, heterogeneity, and diversity of the distributed environment. The naming architecture must be

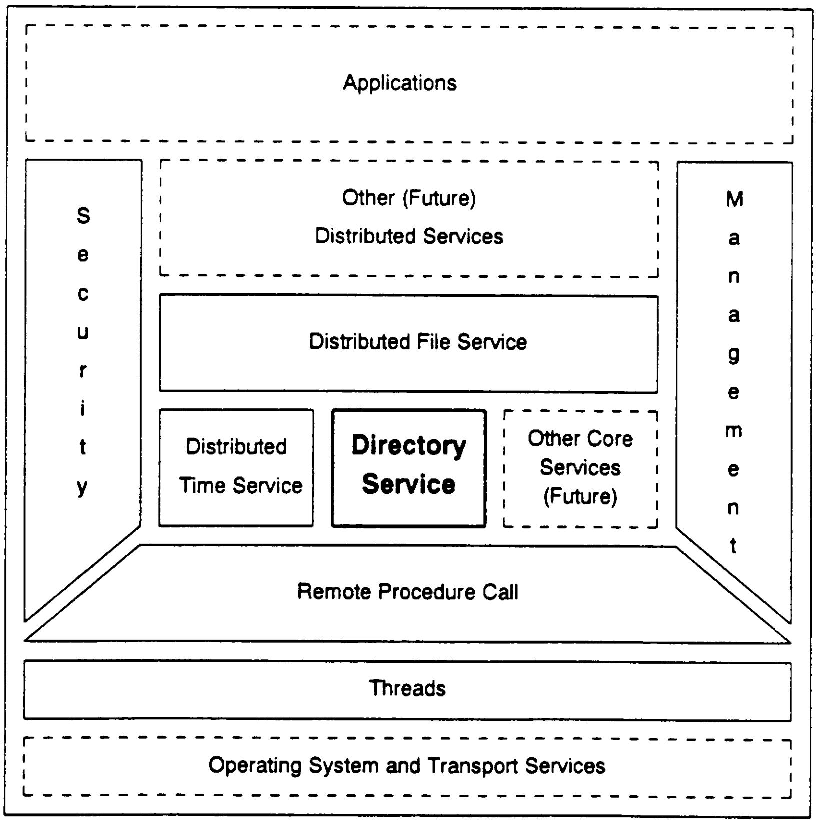


Figure 1: *DCE Component Architecture*

able to cope with the broad variety of entities in the system as well as with the unlimited size and growth.

One may imagine how different this can be. Already after only 12 years of Internet the Domain Name Service (DNS) just hit the 1 millionth registered node, while these nodes are only network addressable hosts, and other to be named and addressed entities such as distributed services are not manged with DNS. Furthermore, the increasing settlement of object oriented modeling and programming in the distributed environment will have to deal with the creation, and deletion, of objects at a much higher rate. Even if only a small part of these objects are permanent and need to be registered, it becomes obvious that the growth rate will be even higher than currently experienced.

While the entities mentioned above, network nodes, services and application objects, are essential for architecting naming in the distributed environment, there are a number of other types of objects such as persistent traditional directory entries (countries, organizations, persons, mail addresses) or data entities like files which should fit coherently into the naming model.

1.2. The Architectural Requirements for Naming

It appears obvious that this wide variety and number of objects cannot be sufficiently dealt with by just one technology. But it is also obvious that the usage of the distributed system requires as much uniformity as possible in accessing the entities.

The first distinct requirement one can observe for name services is the **access model**. Most commonly, name services are used to look up the desired information based on a given name for an entity, where names are either fully distinguished names or names relative to a certain starting point (context) in the namespace. The access model also implies that updates and changes are usually much less frequent than lookup operations. For other special purposes, more complex queries which involve extensive filtering and searching or requirements for a very frequent update rate may become dominant.

The other area where name services behave distinctly is in the **organizational model** of the namespace. While usually a hierarchical tree organization is sufficient, in other cases a flat organization with access through hash tables, or topographical views and directed graphs which, for example, account for multiple inheritance, may be preferable. Also, the desired complexity and flexibility of the structure of the organization may vary. Systems which provide for sophisticated schema management need to be considered.

In this context, the complexity and **semantics** of entries varies widely. For some applications it may be preferable to have just basic attribute values associated with entries while for other applications structured attribute value assertions (type – value attribute pairs) are required.

Other requirements for the name service are reliability, manageability, and security. Since these usually conflict with performance and scalability, one has to be able to balance them according to the particular purposes. Replication, for example, the technology to provide for reliability and availability of the name service information, can be provided with varying degrees of weak to strong consistency. While in some cases lookup failures and retries are cheaper and more acceptable than in others, no single solution can be optimal.

Can Federated Naming Solve the Problems?

An uncountable number of technologies which deal in one or the other way with naming have been developed and have matured. These range from complex directory services to location brokers, network information systems, object inheritance services, and file systems. Although these various naming facilities coexist in the network, and will continue to do so due to the outlined technical reasons and because of political considerations such as corporate and national policies, suppliers interests in particular market segments, and commercial and financial security, this approach is not acceptable in a distributed environment if there is no means provided for a coherent integration of these technologies.

Several approaches for solving these problems have been discussed to a large extent in the community and have been accompanied by several experimental implementations. One of the inspiring architectural discussions was driven by the Advanced Networked Systems Architecture (ANSA), developed at Architecture Projects Management Limited in 1989 [APM89a]. This work introduced the notion of **federated naming** as a fundamental concept of modeling naming. Federated naming allows for coexistence of distinct and diverse name services and provides translators and gateways to communicate beyond the system's boundaries. Rather than approaching a global naming model, the unambiguity of this naming model is based on context relative naming in federated, heterogeneous domains. Interworking is achieved through bilateral cooperation.

Cooperation of federated naming domains, which is more precisely described as interconnection rather than interoperability, is achieved through gateways which wrap and unwrap the appropriate reference information. The wrapper provides an envelope which details the context and mapping of higher level domains. Directed graphs of name contexts rather than the ordering in a hierarchical tree structure is the basis for this structural model.

Federated naming, though, does not completely address the one crucial issue in a distributed environment. It is necessary to mask the complexity and heterogeneity of the system by providing as much transparency and homogeneity to the users as possible. Transparency is certainly not a fixed measure, but final consumers of the system usually do not want to have to deal with differences in the underlying technologies, while programmers or administrators need to have more flexibility to fine tune and tailor the system. However, since pure federated naming does not prescribe basic semantics and certainly not syntaxes, interoperability is limited and requires the knowledge of all involved instances.

Strictly centralized name system such as the X.500 directory system [OSI88a] is the orthogonal approach. It does not provide a means for incorporating other naming systems, but it seems to be much more appropriate and intuitive for a user to think in and deal with hierarchically structured organizations.

2. The Notion of Name Spaces

In refining the naming architecture for DCE we intended to combine the compelling features of both approaches, the federated naming and the centralized global naming model, into one coherent architecture. Instead of addressing the bilateral issues between heterogeneous naming domains, which are actually instances of particular services, the scope of the naming system was defined by discrete namespaces with its distinct properties in the distributed environment. These **composite namespaces** are an abstraction of particular name services or resource managers[†] describing their syntactical and semantical rules.

The primary namespaces are the **global namespace** and the **cell namespace** which are connected via a gateway, the Global Directory Agent (GDA). This decomposes the namespaces into a hierarchical organization (see Figure 2).

Although this paper primarily addresses the DCE naming architecture, specifying the global and the cell namespaces, the service instances, the DCE Global Directory Service (GDS) and the DCE Cell Directory Service (CDS) will also be discussed. It should be noted that the current implementation of the interfaces and protocols already provides the core features, but some of the functionality outlined in this paper are under development and will be noted accordingly. For instance, the refinement of the junction protocol and the specification of a uniform API are currently in the design phase.

2.1. Assumptions for the Cell Name Space

The cell name space is the basic entity which describes an organizational unit of DCE, an administrative domain. Policies and various degrees of protection levels, the security realm is determined by these cell boundaries. Administrative regions and domains may share the same boundaries or they could be organized as substructures.

Typical considerations for the requirements are:

- The intra-cell connectivity is usually higher than the degree of connectivity with systems located outside the cell. LANs rather than WANs are the primary communication channels of systems inside the organizational boundaries of a cell.
- There is implicitly more trust between a principal and an object within the cell than those across cell boundaries.

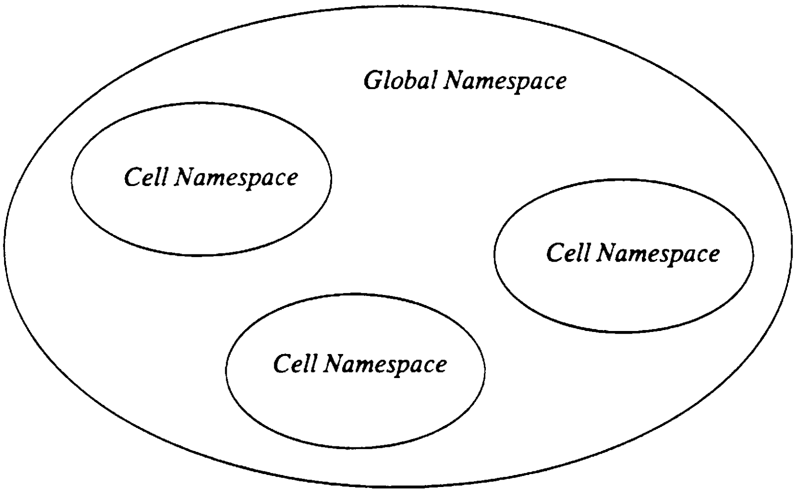


Figure 2: *DCE Name Spaces*

[†] In many instances the term resource manager is being used since the services supporting these composite namespaces are not necessarily name services in the generic sense.

- The cell naming environment is unified regarding the semantics and syntaxes applied to this namespace.

2.2. The Global Name Space

To provide uniformity and worldwide interconnectivity, cell namespaces are always part of and integrated into a global namespace which provides at minimum the location service for foreign cells. The global namespace is represented by a defined small set of globally available centralized name or directory services. Supported references of the global directory services are the Internet Domain Name System (DNS) and the X.500 directory system.

Although a single directory service technology would not be a sufficient and acceptable solution for the entire namespace, it appeared to be sensible to use the few dominant and matured central directory services as facilities for the global namespace. The Internet Domain Name System (DNS) and the IEEE X.500 directory system are clearly the preferred name services which provide worldwide access and interoperability. Both are supported as reference in the DCE implementation. It is likely that X.500 will be the dominating system in the future. Nevertheless, the DCE naming architecture does not restrict the instantiation of the global namespace to just one directory system. Multiple global directory systems can coexist and provide interconnectivity between cells[†] which are represented by different global directory systems.

Additionally, the architecture allows a single cell to be registered with multiple global directory services, however, it is still debatable whether this is desirable. The current implementation of DCE restricts the cell registration to one instance of the global directory service.

The registration of a cell in a global directory service differs for each particular service, depending on registration policies and semantics. In X.500, two attributes in the directory entry composing the cell name (CDS-Cell and CDS-Replica) contain the cell identifiers and protocol sequences necessary to access the instance of the cell directory service. Analogous in DNS, cell names and other relevant information are associated with resource records.

2.3. The Global Directory Agent (GDA)

The connection between the global namespace and the cell namespaces is handled through a Global Directory Agent which acts as a gateway. The primary function of this gateway is to resolve the global part of a given pathname by issuing a referal protocol exchange to the appropriate global directory service. A successful path resolution returns a set of binding handles for the directory service in the targeted cell. The second task of GDA is its function as protocol translator.

While the cell namespace is RPC protocol based, the global namespace is considered to use the respective native protocols such as Internet or the OSI stack.

The initial purpose and design of the GDA is its function as trader. It delivers the binding information of the remote service provider to the

[†] This can be achieved since cells, and more generally each DCE entity, is uniquely identified by an Universal Unique Identifier (UUID) which is guaranteed not to be duplicated or reused, even if generated in independent and disconnected systems. The UUID is an automatically generated 128-bit identifier using IEEE node identifiers and time stamps for its generation and therefore is unique across both space and time. It does not contain the location information of an object. The UUID, with its fixed length, is not infinite, but it is large enough to allow the generation of more than 10^6 objects per second on each node over the next 10^4 years.

service requester. It is conceivable that a GDA could also act as a gateway for performing other directory service operations in the global namespace such as lookup on arbitrary attributes or modification of entries. This is an area which requires some more investigation; in particular the expected benefits need to be defined.

3. The Properties of the Cell Namespace

The DCE Cell Directory Service (CDS) represents the cell namespace. It plays a key role in the DCE naming architecture. While the cell is the administrative domain in the distributed computing environment, CDS is particularly suited to provide the following services:

- Providing the information for distributed applications to locate and bind the service requester (client) with the service provider (server). These operations which generate and obtain the binding information are also called **trading**.
- Connecting multiple heterogeneous naming domains into one coherent environment. This includes the resolution of references to the global namespace and **junctions** to descendent namespaces which can be subcells of a hierarchy of cells or namespaces represented by distinct services such as the file system. CDS can essentially be considered as the central instance for connecting composite namespaces.

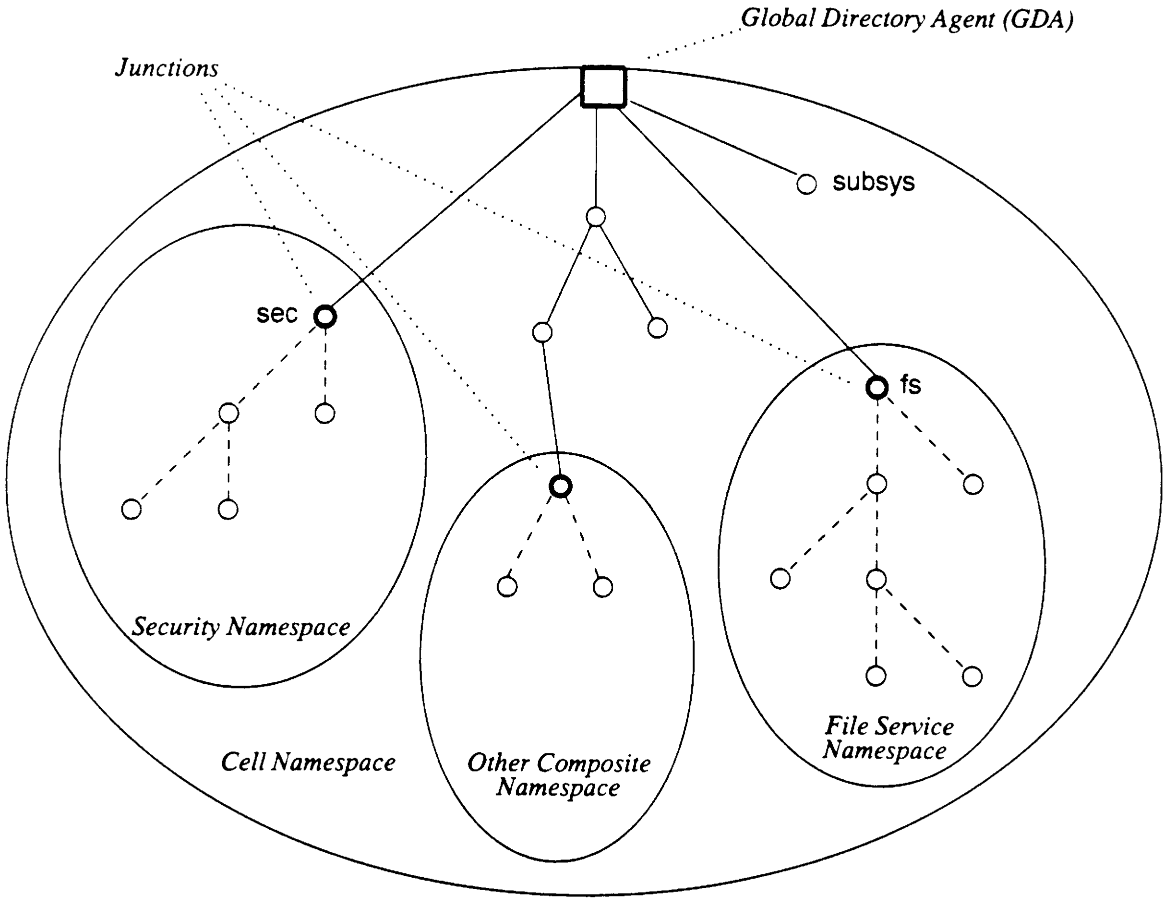


Figure 3: *Decomposition of the Cell Namespace*

Figure 3 shows the organization of a cell namespace. The composite namespace within the cell namespace are accessed through the junction points which are actually directory entries in CDS. Directories and objects registered in CDS (connected by solid lines) are primarily used for the trading purposes. For instance, *subsys* decomposes a subtree containing protocol towers (see below).

3.1. DCE Cell Directory Service

Although the design of the cell directory service is explicitly tailored to meet the naming needs for the distributed system itself, to act as trader and as a service which integrates the various namespaces, there is no reason why it could not be used as a general purpose directory service as well. Its semantic is rich enough to support arbitrary attributes and application-specific naming needs. It is the application design which will determine whether this is sufficient or whether special purpose naming services may be more applicable.

In contrast to a universal directory service, the name service supporting the cell namespace is tailored to meet the special needs in the cell environment. Key requirements are protection, reliability and availability, lookup speed, ease of management, and sufficient communications protocol.

While the binding information retrieved from the cell name service is essentially a reference pointer, the **authentication** and **authorization** verification of a client/server application is a function of the respective applications. In DCE, the name services are not part of the trusted computing base (TCB), and therefore, cannot guarantee integrity of the contents of the information which has been generated by the service providers. This raises interesting security considerations for name services which have similarities to databases. Not the (name service) database but the services which are permitted to access and modify its contents are the trusted entities. Services should always rely on the DCE Security Service, which is part of the trusted computing base in DCE. Assuming the usage of the mutual authentication and authorization capabilities of the DCE Security Service, denial of service attacks would be the worst case scenario with compromised Cell Directory Services. This is usually acceptable, but if security sensitive environments require protection against denial of service attacks as well, their name services must explicitly be integrated into the TCB (for example, installing them on secure hosts).

Availability and **reliability** of the cell directory service information is one of the other requirements with high priority. One can make the observation that the distributed system can only be as reliable as the cell directory system. While availability is usually achieved through replication of complete databases or subsets of them. the degree of consistency depends on the underlying replication technology. For the purposes of the cell directory service it is usually not necessary to provide a strong transaction based replication but the probability of acquiring reliable data should be very high. The system must be capable of detecting mismatches and must provide the flexibility of invalidating and refreshing cached information.

The ratio between **update** and **lookup frequency** is an important factor in the name service layout. A well designed distributed environment must be designed in such a way that bottlenecks in large scaled systems and additional overhead introduced by distributing applications is kept to an absolute minimum. Crucial areas here are additional computation and traffic for security operations, computation for converting data representations, the speed of the physical links, and in particular the latency during association setup and binding. For the purposes of the cell name service, an optimized high speed retrieval of registered binding information has a clear priority over the registration operations, which is an administrative activity and performed much less frequently.

The communications protocol for CDS is **RPC based.** DCE RPC provides the desired degree of transparency independent of the underlying communications structure. Both Local Area Networks and Wide Area Networks (primarily used for inter-cell communication) are efficiently supported. The semantics of the DCE RPC protocol provide the required reliability and guarantees of delivery. Both a security interface for providing mutual authentication and an interface for trading are inherent in the RPC support.

3.2. Trading

Trading in the client/server model is the notion of establishing the association between matching service requesters and the service providers. This process of associating a client to its server peer is of a dynamic nature in the distributed system and it is essential that this is kept as transparent to applications and users of the system as desired. Trading involves a number of considerations:

- Clients and servers may have been developed independently, at different times and sites, by different developers.
- Multiple instances of servers may offer the same services, and they may operate on the same or on distinct objects.
- The number and sites of clients is not predetermined.
- The selection of servers can depend on factors such as load, network latency and bandwidth, availability.
- Servers may crash or hang, and a rebinding may be appropriate, assuming the context has been preserved.

Trading is a generic service which can provide arbitrarily complex information and sophisticated operations. Algorithms can be applied to trading for very dynamic binding behavior, depending on a number of parameters to allow for load balancing or process migration in the distributed sense; or the binding to methods and operations on objects can be supported by following inheritance rules. Such elaborated systems can be built using the DCE naming architecture and atop of the DCE technology implementation. This paper primarily outlines the more primitive operations and facilities necessary to provide for automatic binding, transparent to applications and users.

For resolving the trading issues, the client/server model uses object oriented concepts and treats server instances as objects. Servers **export** an **interface** or a set thereof which specify the remote operations which can be performed on these servers. These interfaces are uniquely and unambiguously identified by Universal Unique Identifiers (UUIDs), and the trading operations essentially resolve the match of interfaces supported by clients and servers. Servers therefore export their interfaces to the cell directory service.

Since instances of servers may operate on one or multiple objects, a further determination of the target in the client/server connection allows for additional **object UUIDs.** For instance, a printer server may support various different printers such as postscript or line printers. While the interface UUID defines the printer server, object UUIDs optionally can further detail on which printer a certain job shall be performed.

The client, which is the service requester, on the other hand, **imports** the **binding information** for establishing the association with any available and matching server from the cell directory service. The algorithm for selecting one of the available servers can be predetermined or specific to a particular instance of an application.

The binding information obtained from the name service contains the location, the network address of the server, and the set of matching protocol stacks, since it is possible that servers may have exported multiple sets of supported protocols. These protocol sets are actually stored as **protocol towers** in the name service. The number of levels of these towers certainly varies, but a tower must sufficiently define the matching set of protocols. A protocol tower could, for example, contain protocols for the data representation, RPC, transport, and the network, in addition to the interface UUID.

For navigating through the namespace and finding the right set of servers, the trading facility provides for three classes of entries: server, group, profile. **Server entries** contain the protocol and addressing information of one server instance, while **groups** are an unordered set of equivalent server entries (servers with common interface and object UUIDs). Groups can be nested.

The **profile** entry in the name service contains a collection of profile elements. These elements are determined by a single interface and contain an ordered list of members which can be servers, groups, or profiles. The ordering of these providers for the particular interface is achieved through priorities. A profile may also contain a default profile element, which is a pointer to another (default) profile which will be searched if no compatible binding could be found in the profile. This allows for customizing the ordering of the search for a compatible server. This could follow topological views or other means of structuring the organization.

The outlined model demonstrates how the cell directory service provides a means of hiding the complexity of the distribution for the consumer of applications. Administration and mangement of these entries, such as profiles, can be centralized within cells but it is also permitted and possible to overwrite certain default setups.

As clearly as the final consumers of the system must be supported, the programmers of distributed applications must also be provided with instruments able to handle the complexity but also flexible enough to develop applications which can dynamically be installed and configured. A special purpose **application programming interface** (API) must be sufficient for exactly performing the set of name service operations necessary for the binding or trading operations. Only designated system administrators should be required to further manage the name service setup and only those application writers who want to use the cell name service for purposes beyond the trading operations should be required to use a more sophisticated general purpose application programming interface.

3.3. Junctions

We can make the assumption that a generic naming service has properties such as slowly changing namespace and weakly consistent replication. While this may be suitable for objects which reside in global or cell directory services, there are a number of objects in the distributed computing environment which have different naming characteristics. This requires a mechanism to incorporate composite namespaces into the global namespace, or more precisely, the cell namespace.

All object names which are exported to the cell namespace must be resolved via a single name resolution protocol. This includes the capability of deriving from the name the object location and potentially a minimal set of other information about the object. DCE introduces a DCE RPC based junction protocol which currently provides for the incorporation of the security and the distributed file service namespaces into the cell namespace.† Other examples for special purpose resource managers are inheritance trees for object oriented modeling or services for transitive, very short lived entities such as processes. The junctions allow clients to resolve the CDS portion of the name and pass the residual part of the name to the resource managers of the respective composite namespaces. While these junctions are currently specific to each resource manager, DCE will provide a generic junction protocol and the appropriate interfaces to support uniform cross resource manager operations (see Figures 3 and 4).

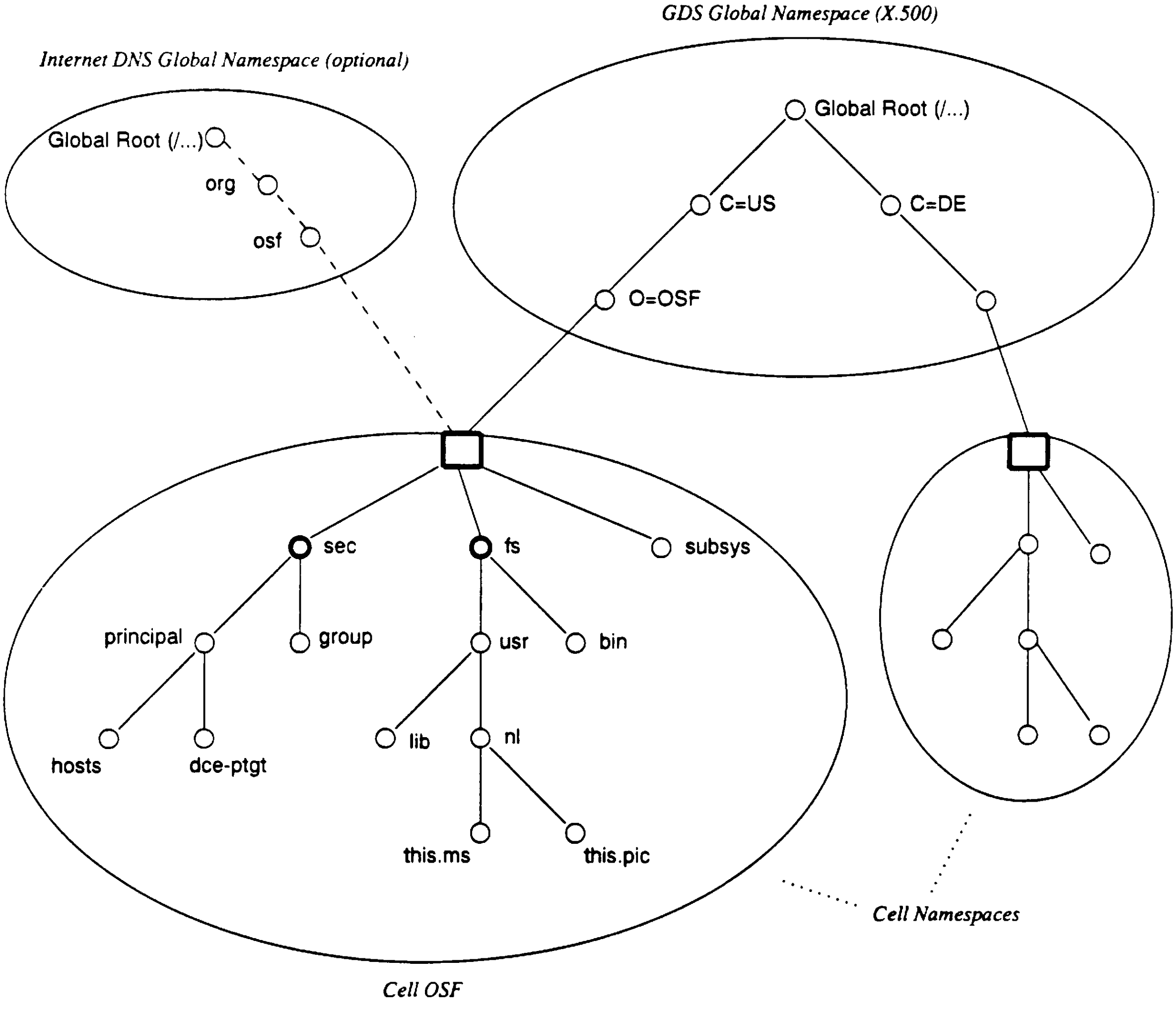


Figure 4: *Example of a complete DCE Directory System*

† The Global Directory Agent (GDA) is a special form of junction which provides the capability of resolving the binding for a foreign cell to the local Cell Directory Service. The referral of a successful GDA operation points to CDS in the targeted cell.

In order to integrate other composite namespaces into the cell namespace, the junction protocol requires the resource managers to support at minimum a set of functions to resolve the residual components of the pathname by using a defined referral mechanism. The binding information of the target object and possible attributes, defined in an attribute schema, will be the result of the junction operations.

It is conceivable that future protocol extensions and agreements on a low level naming interface will allow for rudimentary operations across name services and resource managers. These may be operations such as add, update, delete, and list objects and attributes.

The syntax of the pathname passed to the junction protocol must comply with the DCE name syntax described below. As outlined, components of this pathname (slash separated) may contain a name structure according to the semantics of particular resource managers.

3.4. Hierarchy of Cells

Cells are administrative domains which are essentially determined by the security policies of organizations; thus cells are usually equivalent to realms in the security space. Communication across cell namespaces requires the establishment of additional trust relationships involving the cell-based certification authorities. Independent of the employed encryption technologies, whether they are public or secret key based, it is not likely that independent organizations will agree on an external agent controlling the global certification authority.

Matters are completely different within enterprises and organizational boundaries. For managerial or other organizational reasons it appears that enterprises want to be able to manage subdivisions separately, but in contrast to inter-cell relations apply central authorities for policy determination, administration, and security. For instance, an organization may have security policies and trust relationships for the whole enterprise while a number of separate administrative domains may be in existence.

This requirement can be complied with by allowing cell roots to be registered inside another cell instead of registering a cell in the global namespace. This leads into an arbitrarily nested hierarchy of cells. These hierarchies of cells with the top level cell registered in the global namespace are collected in a particular cell namespace representing an enterprise. Here we have to make the distinction between a cell which characterizes the organizational boundary, administered by an instance of a cell directory service, and the cell namespace which is the union of all cells of a single rooted hierarchy of cells.

There are a number of **benefits** of subdividing the cell namespace into a hierarchy of cells. Higher transparency and less administrative overhead in establishing the security context has already been mentioned. Creating subcells does not require the involvement of an external authority to register a cell and it does not require the availability of a global directory service for inter-cell communication within a cell name space. Furthermore, enterprise-wide central policies and administrative authorities can easily be applied to a hierarchy of cells.

Another aspect which needs to be considered by supporting cell hierarchies is the natural migration of **legacy systems** into a DCE namespace. While the creation of cell hierarchies top down may apply to newly created environments, these namespaces will usually be deployed from bottom up. Disperse subdivisions may create cells independently and want these to be connected to one uniform cell namespace eventually.

Although it may disrupt the transparency goal (the distinguished names for objects changes), a facility for renaming cells must be provided for.

4. Security Implications

Since the security aspects are some of the most sensitive aspects in a distributed computing environment, the naming architecture cannot be separated from the architecture of the security system. From the security perspective, a **realm** is the administrative unit for authorization control. A realm is represented by a common certification authority while a cell represents the collection of resources which belong to a common naming authority. A cell, or in a hierarchical organization of cells the top level cell, usually maps directly to a realm.

Access to entities is granted to authenticated principals (such as users, groups, and services) based on their ability to construct and present a **Privilege Attribute Certificate** (PAC). These certificates sealed in PACs are compared with the identifiers represented in Access Control Lists (ACL).

Since it is not practical that all principals who are potentially involved in cross realm interactions establish direct links to the foreign realms, DCE adapted the Kerberos V5 model of **transitive trust** which allows access to be granted through established trust relationships between the authorization authorities.

To contain potential damage of the system, the security system in DCE imposes a hierarchical structure of transitive trust. It limits transitive trust to either those principals which share a common certification authority within a given structure or substructure of a hierarchy of cells or to those entities which directly share inter-realm keys through their certification authorities.

5. Management Implications

The naming architecture in a distributed environment has to take into account that not only a very large number of nodes and entities needs to be manageable but also that the complexity of the system grows exponentially by introducing diverse technologies which have distinct and sometimes contradicting requirements on naming. Not only must the naming architecture provide for facilities which can be exploited by the management systems, but it also must be carefully designed to avoid additional unnecessary complexity.

The **scalability** is doubtlessly one of the major concerns. As discussed earlier, single naming systems in the order of 10^5 and 10^6 nodes are already in active use and the conceivable scope of addressable entities will be much higher in the near future.

One other, often underestimated, aspect of scalability is the capability of **down scaling**. Distributed system environments involving just a few systems may very well exist. Apart from possible performance overheads, system administrators in those environments must not be burdened with management functions which are only necessary in large systems. But since systems are dynamic and may grow into larger more complex environments, the same uniform interfaces must be applicable.

DCE addresses these administrative problems by providing rules for organizing the namespaces around cells, the basic administrative units and by potentially subdividing cells into smaller administrative regions (hierarchical cells). In addition, there are underlying rudimentary operations which will be specified by a common name syntax, a defined set of Application Programming Interfaces, and access protocols which define cross namespace operations through junctions.

To reduce the demand on system managers and consumers, the required security services must provide as much transparency as possible. Various protection levels must not require completely different activities. Security will not be enforced if there is no coherency, if the model is not intuitive, and if the administrative overhead is disruptive.

This requires a well balanced model which provides the basic set of specified rules and operations but also the right amount of freedom to apply organizational or personal policies. Again, cells are the domains in which policies are being defined.

However, generic administrative tasks such as user management or resource management can sensibly be performed using the technologies associated with the DCE naming architecture. Obviously undesirable behavior, circularities and duplications such as several user accounts for the same person or different views of the directory structure dependent on the workstation are prevented by the architecture.

Other technologies layered on top of DCE such as the Distributed Management Environment (OSF DME) can further exploit the underlying services to provide this uniform and coherent system view.

6. DCE Name Syntax

Since DCE naming deals with composite namespaces and service implementations thereof, a number of distinct semantical rules and syntaxes must be accommodated. Some basic guidelines for the syntax of representing names applicable to the entire global namespace must be followed. These rules provide for a uniform notation of human-readable, character-string-based names. Special rules apply to the global part of a name, the global typed name syntax, describing an abstract data type for representing names and its matching rules for equality [Tuv90a].

The name syntax is intended to represent unambiguously names which are hierarchically ordered as an ordered set of components. This ordering follows the rules left-to-right and top-to-bottom, where top is the superordinate finally representing the root of the tree structure.

The components of a name are separated by slashes (/). A name may have a random number of components which in turn may represent simple names or a complex structure according to the semantics of respective name system or resource manager. The pathname prefix "/..." determines the global root, while the prefix "/.:" is the signature of the root for the local cell (shortform of the global part of the pathname).

For X.500 based global names, equal signs concatenate the type value pair of AVAs (attribute-value-assertions) and comma-separated fields represent multiple AVAs. The global part of the name must not contain empty components. For example,

```
/.../C=US/O=OSF/OU=DEV,L=CAMBRIDGE/
```

is the name of the OSF development cell in Cambridge.

In a global namespace using the Internet DNS, the complete Internet cell name would be represented in one component in the usual dot-separated notation in little-endian format. For example

```
/.../osf.org/
```

would name the cell OSF.

The name syntax can be equally applied to typed names such as X.500 as well as to untyped names such as the DCE Cell Directory Service. A resulting global name representation therefore can be a combination of concatenated typed and untyped names.

An example of a global name for an object (file) in the Distributed File Service would look like:

```
/.../C=US/O=OSF/fs/usr/nl/this.ms
```

or equivalently, the same file would be addressed relative to the cell as:

```
/.:/fs/usr/nl/this.ms
```

Figure 5 shows the distinct components of the pathname in the example above.

One caveat in determining generic syntax rules for naming is the fact that name services and other resource managers use different character sets, need various control characters, and support different **encodings** for the representation of characters. Also, the support of **national languages** requires further considerations. While there has been some progress made in addressing these issues for standalone systems, it is largely unresolved and not standardized for distributed systems yet. This is one challenging area which requires attention in the future.

7. Portability and Seamless Programming Interfaces

While this paper discusses primarily the capability of the naming architecture to provide for interoperability between namespaces and related issues, requirements of programmers for portability of their applications must not be neglected. It is necessary to expose a set of well specified application programming interfaces which can uniformly be used for any integrated name service.

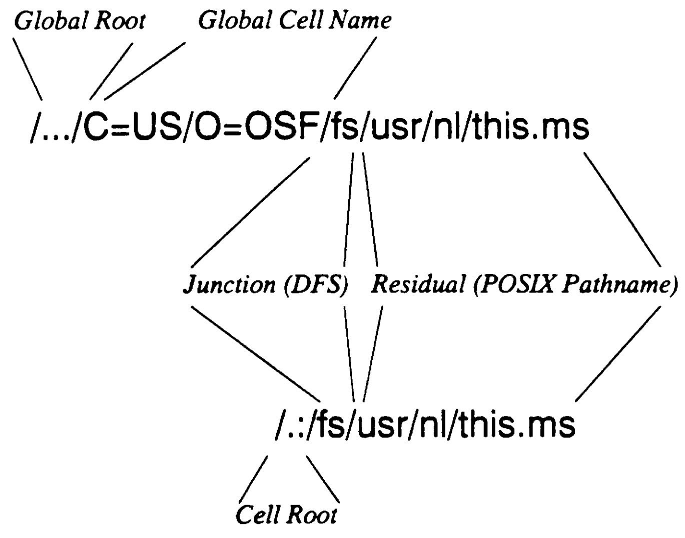


Figure 5: *Pathname Components*

The DCE implementation supports two programming interfaces, the X/Open Directory Service API (XDS) [CAE91a, CAE91b] and Name Service Independent Interface (NSI).

XDS is the generic directory service interface which exposes the full set of semantics supported by directory services such as X.500. It is the primary API used for GDS and CDS based applications.

NSI is an RPC (Remote Procedure Call) interface targeted to manage and obtain server binding information. NSI provides the interface to the trading capability of CDS.

OSF is investigating a generic name service API which provides access to a common set of rudimentary name service operations which can be performed on any resource manager in composite namespaces which comply to the junction protocol. This API would be on a low level, much less rich than XDS requires.

8. Summary

This paper discusses the underlying information model of the DCE naming architecture and addresses the issues involved with incorporating the various distinct needs and solutions for object naming in the distributed computing environment. It points out that the architecture provides a sound framework and that the implementation of the DCE technology has already laid the groundwork to solve naming in distributed systems.

The paper did not address application usages of name or directory services. It is obvious that the DCE naming technologies are useful for other purposes such as storing arbitrary directory information. This would have exceeded the scope of this discussion, but opens a wide field of opportunities. Also, other related technologies such as DCE Time Service and Remote Procedure Call have not been elaborated in this paper, but they should be considered as prerequisites for building the described system.

There are a number of areas which require further efforts to refine the architecture and implementations thereof. Some of these issues are subjects of further research in the community, still need to mature and may need additional standardization efforts. Areas which need most attention are:

- Providing an uniform access model (interfaces and protocols) to explore fully the capabilities of composite namespaces.
- Expanding the trading capabilities to incorporate object-oriented models.
- Finer granularity and even more flexibility in the structural organization of cells by completely solving the modeling of hierarchical cells and the associated security issues.
- Improving the management capabilities of the distributed system by integrating DME.

Finally, I would like to point out that the DCE architecture represents a synergy of requirements to solve the inherent issues in distributed systems with matured and proven technologies. One major effort taken by OSF was the integration of various orthogonal technologies into one coherent and expandable model. The first release of DCE has been shipped in January 1992. Already a number of companies have adopted DCE and its model and gained expertise with DCE in several projects. For example, the Distributed Management Environment, a technology currently being developed at OSF, extensively uses DCE facilities.

Acknowledgements

The DCE architecture and the accompanied technology implementation was only be possible as a unique multinational and multi-company project with hundreds of person-years of effort. I want in thank all the architects and engineers of supplier companies for their involvement and help and the experts of OSF member companies who actively participated in the Special Interest Group (DCE SIG), and in particular, Dr. Walt Tuvell, DCE architect at OSF, provided valuable review comments on this document. My special thanks go to Ann Hewitt for her editing help.

References

[APM89a] APM, *The ANSA Reference Manual,* Architecture Projects Management Limited, Cambridge, United Kingdom (March 1989).

[Bir89a] Butler W. Lampson, Michael D. Schroeder, Andrew Birell, *A Distributed Systems Architecture for the 1990's,* Digital Systems Research Center, Palo Alto, California USA (1989).

[CAE91a] CAE/XDS, *API to Directory Services (XDS),* X/Open Company Limited, United Kingdom (1991).

[CAE91b] CAE/XOM, *OSI Abstract Data Manipulation API (XOM),* X/Open Company Limited, United Kingdom (1991).

[Joh91a] Brad C. Johnson, "A Distributed Computing Environment Framework," in *OSF Technical Paper*, Cambridge, Massachusetts USA (June 10, 1991).

[Les90a] Norbert Leser, "An Open Distributed Systems Architecture," in *OSF Technical Paper*, Cambridge, Massachusetts USA (June 1, 1990).

[OSF91a] OSF, "Application Development Guide," in *DCE 1.0 Documentation*, Open Software Foundation, Cambridge, Massachusetts USA (December 1991).

[OSF91b] OSF, "Administration Guide," in *DCE 1.0 Documentation*, Open Software Foundation, Cambridge, Massachusetts USA (December 1991).

[OSI88a] OSI, *ISO9594, Information Processing Systems – Open Systems Interconnection – The Directory,* ISO/IEC JTC 1 (1988).

[Tuv90a] Walt Tuvell, "Proposal for Global Typed Name Syntax," in *OSF-UI-X/Open Memorandum*, Cambridge, Massachusetts USA (September 17, 1990).

The BERKOM Management Platform

Andreas Dittrich

Gesellschaft für Mathematik
und Datenverarbeitung (GMD)
Forschungszentrum für Offene
Kommunikationssysteme (FOKUS)
Berlin, Germany
dittrich@fokus.berlin.gmd.dbp.de

Abstract

The *BERKOM Management Platform* supports the development of managing and managed applications based on OSI management standards. The *Basic Management Support System* (BMSS) constitutes the core of the platform assisting in the exchange of management information. Additionally, the platform provides two tools which also support the development of managing and managed applications: an MO definition tool called DAMOCLES and a MIB browser which is intended to simplify the testing of managed object implementations and which can also be regarded as a simple manager. This paper presents an overview of the concepts used within the BMSS and the features of the supporting tools.

1. Introduction

The BERKOM project (BERliner KOMmunikationssystem) was started in Berlin in 1986 in order to provide experience in using high-speed 140 Mbit/s broadband ISDN networks that the German PTT, the Deutsche Bundespost, was installing in Berlin. Various projects have been initiated within the framework of BERKOM in order to evaluate its potential, with particular attention being paid to new protocols for high-speed bulk transfer over reliable links, group communication, inter-organisational networking, and the possibilities offered by the high-speed network for transmitting video pictures, text and speech and the innovative applications that can be developed in such an environment.

All these projects need to be concerned with the management of their particular application and in the first version of the BERKOM reference model [Pop88a] it was acknowledged that a unified management concept should be developed for the BERKOM integrated services. In order to prevent each project developing its own management solutions that are incompatible with the other BERKOM projects, the BERKOM project "Management of Distributed Applications in B-ISDN" (BER-

MAN) was started in the autumn of 1988 within the GMD-FOKUS Open Management Architecture Group.

The aim of the BERMAN project is to develop a model for managing distributed applications in an open broadband ISDN environment and to provide the *BERKOM Administration Infrastructure* (BAI) which consists of the *BERKOM Directory* [BER91a] and the *BERKOM Management Platform* [BER91b]. The *Basic Management Support System* (BMSS) constitutes the core of the BERKOM Management Platform enabling the exchange of management operations. Furthermore, the platform includes a tool for the definition of managed objects called *DAMOCLES* and a MIB browser which allows the inspection and manipulation of managed objects located in a particular MIB.

Section 2 describes the architecture of the BMSS. Section 3 presents the features of the DAMOCLES tool and Section 4 discusses the characteristics of the MIB browser. Finally, section 5 outlines future developments within the framework of the BERKOM Management Platform.

2. The Basic Management Support System

The *Basic Management Support System* (BMSS) is based on OSI management thus enabling cooperation with other management systems which support this standard. It encompasses (see Figure 1):

- A CMIP [ISO90a] implementation based on ISODE,[†]

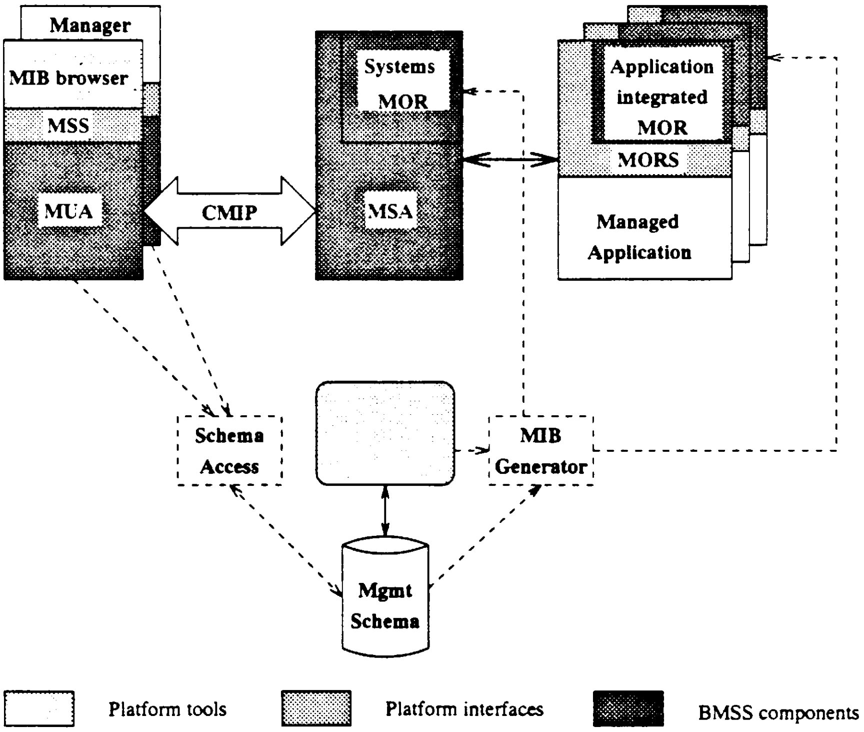


Figure 1: *Architecture of the BERKOM Management Platform*

[†] ISODE (ISO Development Environment) is a non-commercial, freely available development environment for distributed applications, which was developed by a number of companies and institutions (including University College of London and Wollongong Group). The aim of ISODE is to facilitate the development of applications which communicate using OSI protocols.

- A *Management User Agent* (MUA) offering the *Management Support Service* (MSS) which gives managing applications access to OSI management,
- An OSI management agent called the *Management Support Agent* (MSA) and
- A programming interface which supports the implementation of MOs and the establishment of *Managed Object Repositories* (MORs). This interface provides the *Managed Object Repository Service* (MORS).

Additionally, a library of C++ classes representing generic attributes and basic managed objects is provided.

The various BMSS components are described in the subsequent subsections.

Managed Object Repositories

OSI management is carried out by accessing and manipulating managed objects (MO). Managed objects are abstractions of data processing and data communications resources [ISO91a]. A managed object class is defined as a collection of attributes, actions and notifications and related behaviour. The value of an attribute can determine or reflect the behaviour of a managed object. The attribute is observed or modified by sending a request to the managed object to read ("get") or write ("set") the attribute value. Actions are operations on a managed object, the semantics of which are specified as part of the managed object class definition. Notifications are emitted by a managed object and contain information relating to an event that has occurred within the managed object. The *Management Information Base* (MIB) is the conceptual repository of all managed objects located within an open system. Thus, a MIB can be seen as a deposit of information required for management purposes.

Frequently, information which is of interest to management is already maintained in volatile memory by an application for application-specific purposes. BERMAN is particularly concerned with such applications because integrating MOs into these applications avoids the duplication of this information. Moreover, if this information is of a highly dynamic nature this method ensures minimal access time by the application and the most consistent view of the application state for a manager. It also simplifies controlling the application by management requests since write operations on attributes and action operations on managed objects will be performed directly on the application itself and not on any representative, located for example in a database, which would require additional effort in order to process the requests.

There is no prescription as to how a MIB should be implemented. The BERMAN approach for a MIB implementation is called *Managed Object Repository*. A Managed Object Repository (MOR) contains a set of MOs and administers the containment relationship between these MOs. The local root of a part of the overall containment tree maintained in a MOR is called a *subtree-root MO* (SRMO). These SRMOs play an important role for the administration of Managed Object Repositories which is explained in subsection 2.2. A MOR controls the access to its MOs and ensures the synchronisation of operations upon them. It is generic since the type and number of MOs which can be stored in a MOR is unrestricted. The union of all MORs established within an end system constitutes this system's MIB.

The Managed Object Repository integrated into the Management Support Agent is called *Systems Managed Object Repository* (SMOR). The MORs to be integrated into managed applications are named *Application-integrated Managed Object Repositories* (AMOR). The SMOR is intended to store the root of the local MIB, i.e. an instance of the MO class "system" or one of its subclasses. Furthermore, system managed objects like discriminators and logs are to be maintained within the SMOR. Within the BMSS the assumption has been made that within one end system exactly one SMOR, but any number of AMORs can exist.

Since a MOR's task is to enable managed applications as well as managers to access the managed objects it maintains, two interfaces have been defined:

1. The manager interface is used if a request from a manager has to be performed on managed objects stored in the MOR. Thus it provides operations similar to CMIS, i.e. managed objects can be created and deleted and they can be selected by means of scoping and filtering. Actions can be invoked on the selected MOs or their attributes accessed. Moreover notifications will be emitted via this interface.
2. The application interface enables the managed application to access and manipulate the stored managed objects. It allows the creation, manipulation and deletion of managed objects. However, only a single MO can be accessed by an operation invoked at this interface since no scoping or filtering is supported. No

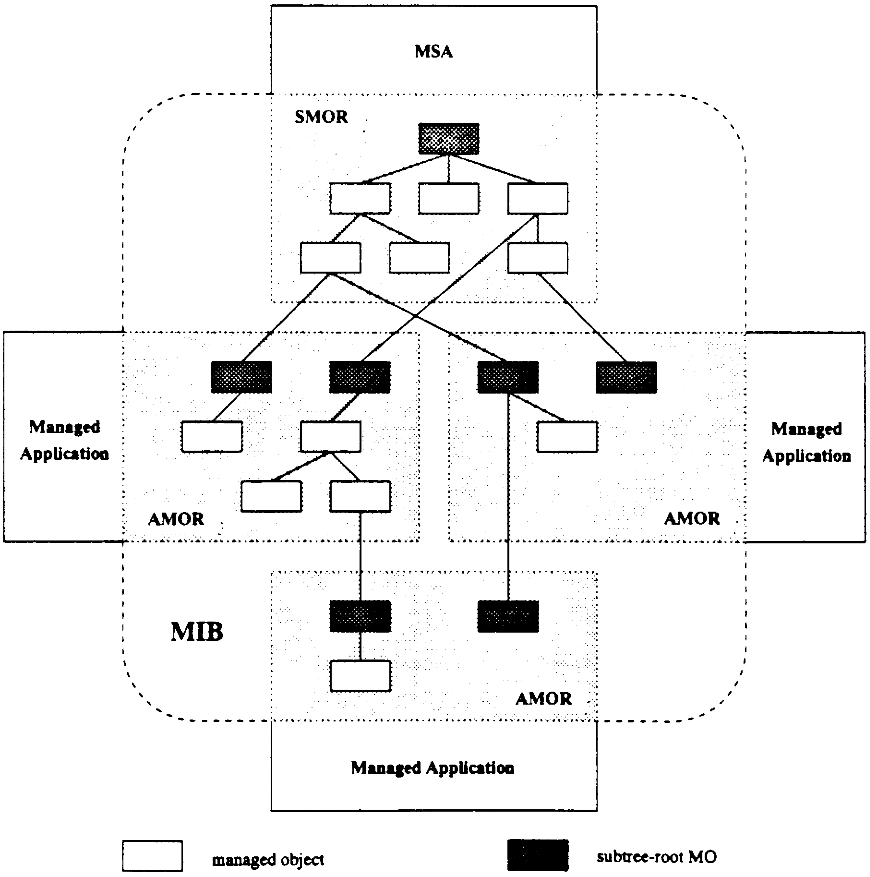


Figure 2: *The concept of Managed Object Repositories*

actions can be performed via this interface but the emission of notifications can be explicitly triggered.

Although similar in functionality, the operations available at both interfaces are mapped onto different operations supplied by the managed objects, i.e. the distinction between a manager and an application interface at the MOR level is reflected at the MO level. The reason for this separation is to clearly distinguish the different views and required operations of a manager and a managed application. For example, managed objects need not always be created or deleted on behalf of a manager's requests, but then, of course, the managed application must be able to initiate the MO creation or deletion. Different accessibility of attributes has to be supported, too. A non-resettable counter can only be read by a manager, but the application must be able to increment, i.e. to write, this counter whenever the counted event has occurred.

The Management Support Agent

The *Management Support Agent* (MSA) is the BMSS representation of an OSI management agent. Therefore it

- Receives CMIS requests from local and remote managers,
- Performs access control and authentication,
- Determines which MORs are affected by the request, i.e. it computes in which MOR(s) the requested managed objects are located,
- Distributes the CMIS requests to the selected MORs,

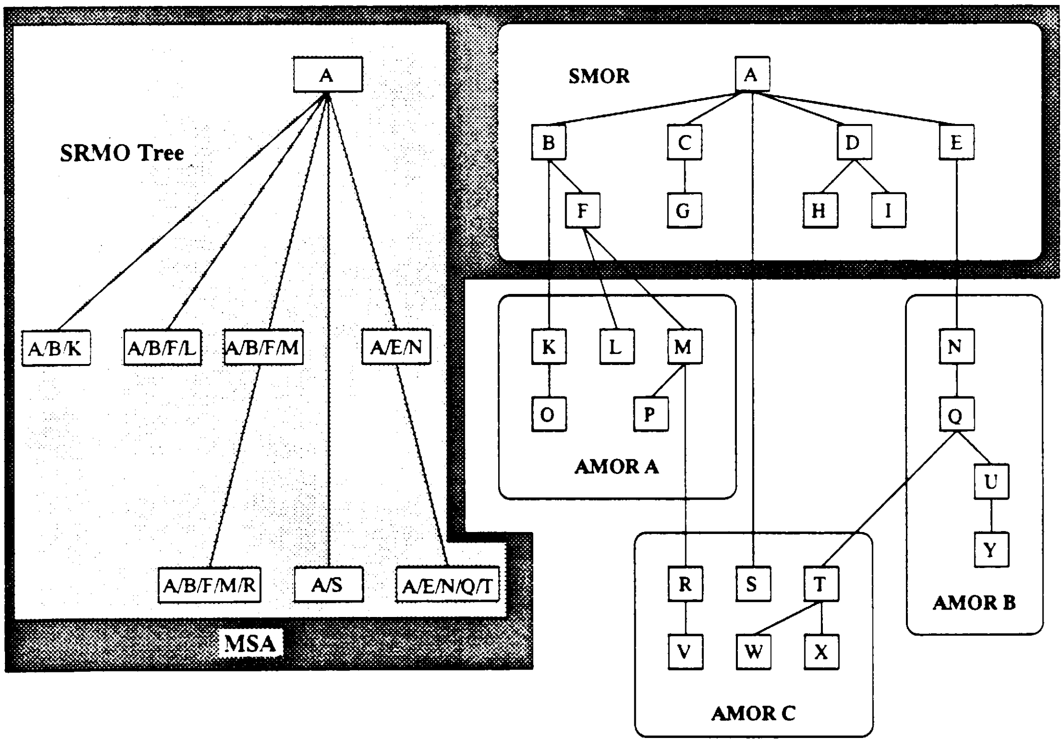


Figure 3: *The SRMO tree*

- Processes the responses of the MORs and returns the result to the requester.

The MSA also receives notifications emitted by managed objects and decides whether they have to be disseminated to managers or whether they have to be logged locally. In the former case it is responsible for establishing an association to the manager and for transmitting the notification. In the latter case the MSA creates the new corresponding MO(s).

The selection of the repositories affected by a management request is based on the *SRMO tree* maintained by the MSA. Each time a subtree-root MO is created within a MOR its local distinguished name is reported by the MOR to the MSA. This name and the identification of the MOR is stored in a newly created leaf of the SRMO tree. If a management request is received, the MSA looks for the longest matching prefix of the specified name of the base MO. For example, according to Figure 3 the longest match for the MO "`A/B/F/M/R/V`" within the SRMO tree is "`A/B/F/M/R`". Thus, the management request is sent directly to AMOR C.

If the CMIS scope parameter is used, the MSA computes for each affected MOR the "local" base MO and the adequate value of the scope parameter. For example, if the originally specified base MO is "`A/E/N`" and the scope value is specified as "individual nth level" with *n* set to 3, the MSA sends a request to AMOR B with base MO "`A/E/N`" and an unchanged scope level, and a request to AMOR C with base MO "`A/E/N/Q/T`" and the scope level set to 1.

When a subtree-root MO is deleted, its local distinguished name is reported by the MOR to the MSA which in turn deletes the corresponding entry within the SRMO tree.

The Management User Agent

The *Management User Agent* (MUA) offers the *Management Support Service* (MSS) which permits an application to assume the manager role. It is supplied by a function library which currently only supports the *pass-through services* as defined by the Object Management Function [ISO91b]. That is, it supports the basic services for creating and deleting managed objects, reading and modifying attribute values as well as for receiving notifications about the occurrence of predefined events within a particular managed object.

In addition, functionality for handling management associations is provided, permitting a managing application to determine by itself when connections should be established or terminated. If these services are not used, management associations are established and terminated automatically by the MUA if a management request is to be sent. However, in this case the use of *global distinguished names* is required.

The MUA uses the Directory [ISO89a] to obtain presentation addresses and further useful information about the capabilities of a managed system. The managed system is specified by supplying the Distinguished Name (DN) of a directory entry of class *managementSupportAgent* which describes the remote MSA [BER91c]. Listing 1 shows the definition of this class.

The superclass *applicationProcess* is defined by the Directory standards [ISO89b] and is intended to represent the part of an OSI application which performs the actual information processing. The attributes added to the "managementSupportAgent" class give information about the standardised and non-standardised, i.e. BERKOM-specific, capabili-

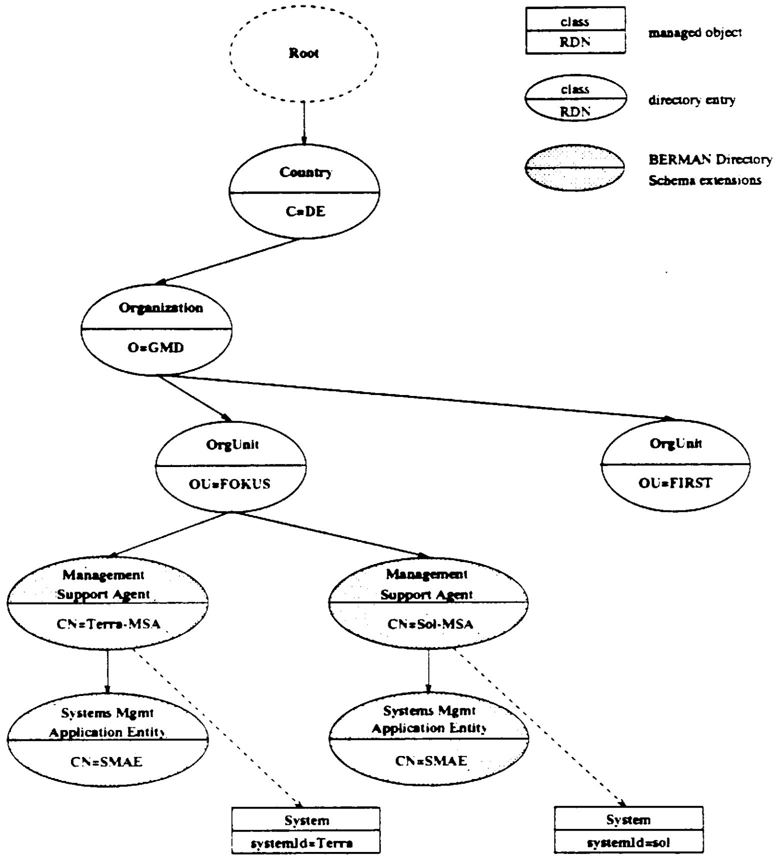


Figure 4: *Use of Directory within the BMSS*

ties of the MSA. Their values are a set of object identifiers (OID) specifying the supported management functions and managed object classes.

Within the Directory Information Tree an entry of class *systemsManagementApplicationEntity* is subordinated to the MSA entry (see Figure 4). This is a subclass of the predefined class *applicationEntity* which holds among other things the presentation address of the entity. The subclass "systemsManagementApplicationEntity" additionally gives information about the supported CMIS capabilities, e.g. if scoping and filtering can be selected for use on a connection.

Within the BERKOM Management Platform the Directory is also used to construct global distinguished MO names. This is done by virtually linking a system's MIB to the directory entry of the MSA responsible for this MIB. According to Figure 4 the global distinguished name of

```
managementSupportAgent OBJECT-CLASS
    SUBCLASS OF applicationProcess
    MUST CONTAIN {
        supportedOsiManagementFunctions,
        supportedOsiManagedObjectClasses,
        supportedManagementFunctions,
        supportedManagedObjectClasses }
    ::= {berkomObjectClass 5}
```

Program 1: *Definition of the directory object class "managementSupportAgent"*

the system MO "`systemId=sol`" is
"`C=de@O=gmd@OU=fokus@CN=sol-MSA@systemId=sol`".
The MUA scans a global distinguished name for the presence of a Relative Distinguished Name (RDN) concerning the attributes "systemId" or "systemTitle". The preceding sequence of RDNs is assumed to be the directory name of an entry of class "managementSupportAgent".

3. DAMOCLES

Within the framework of the BERKOM Management Platform the tool DAMOCLES [Wit91a] has been developed which simplifies the definition of management information. The definition process can be divided into several steps and DAMOCLES supports these steps by providing the following capabilities:

1. After the "management" aspects of an application to be managed have been determined, i.e. which information is required and which operations have to be performed, an "informal" definition can be made. An important issue within this phase is to check whether any existing definitions could be used, either directly or by means of inheritance and specialisation. DAMOCLES supports this task by providing a *Management Schema Browser* and a *Management Schema Library* which includes all the definitions specified within [ISO91c] and those definitions already made by means of DAMOCLES. The Schema Browser allows to examine the contents of the Management Schema Library and is also able to display the inheritance relationships of particular entries.
2. After that, the "formal" definition can be made according to the templates and rules defined in the "Guidelines for the Definition of Managed Objects" (GDMO) [ISO91d]. DAMOCLES supports this step by providing a *Template Editor.* The Template Editor provides a template for each desired definition type, the syntax of which is adapted to the GDMO standard. Since the templates already contain most of the defined keywords, the number of possible syntax errors is reduced. Moreover, the Template Editor guarantees the consistency of the definitions. For example, it prevents the definition of an MO class from referencing an unknown attribute. It also ensures that no object identifier is assigned twice.
3. In the last phase, the management information has to be implemented and integrated into the managed application. This could be supported by the "automatic" generation of code based on the formal definitions made with DAMOCLES. However, the current version of DAMOCLES does not support this.

DAMOCLES divides the process of defining new management information into two phases (see Figure 5):

- In the first phase new definitions are added to a *local* management schema. The definitions can be entered interactively with the Template Editor or read in from an ASCII file. In the latter case, a syntax check is made as to whether the structure of the definitions corresponds to the notation defined in [ISO91d]. In the former case, only the syntax of the specified labels and object identifiers is checked. During the subsequent semantic analysis the uniqueness of the specified labels and object identifiers is ensured. In addition, a type check of the references is made. For example, it is proved that a label used within the ATTRIBUTE section of an MO class definition really names an attribute definition. However, since the local management schema can be incomplete, i.e. the contained definitions can reference other definitions that do not already exist, this check can only be done if the referenced definition already exists.

- In the second phase schema definitions from the local schema are added to the *global* management schema represented by the Management Schema Library. However, this can only be successful if all referenced definitions are now available either in the local or in the global management schema and the reference check does not indicate any type errors. In order to ensure the consistency of the global management schema at any point in time, the definitions from the local management schema are inserted into the global management schema in a specific order. First of all, the behaviour definitions and all definitions which do not reference any other definitions are stored. Then, definitions which exclusively reference definitions already available within the global management schema are inserted. This step is iterated until all definitions made within the local management schema are processed.

DAMOCLES is implemented in ANSI-C and runs under SunOS. Its graphical, interactive user interface is based upon X11R5 and OSF/Motif 1.1. The global management schema is kept within the UNIX filesystem so that DAMOCLES is independent of any commercial database.

4. The MIB Browser

The need to test MO implementations led to the development of a so-called *MIB browser* which is a tool for inspecting managed objects of any management information base and for supporting the execution of operations defined on these MOs. Figure 6 shows an example display of the MIB browser. When invoking the program a list of hosts, which is read from a configuration file, is displayed (the window in the upper left corner of Figure 6). After selecting the desired host, a management association is established to the corresponding management agent and

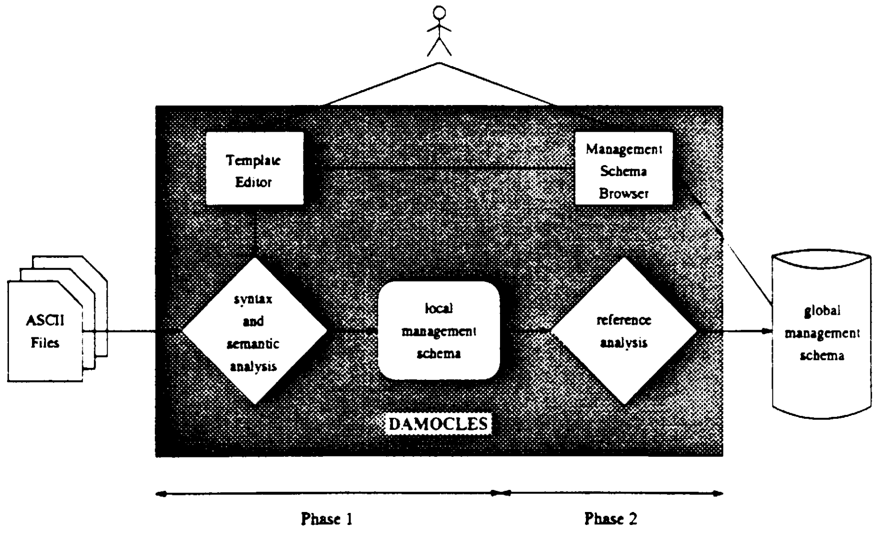
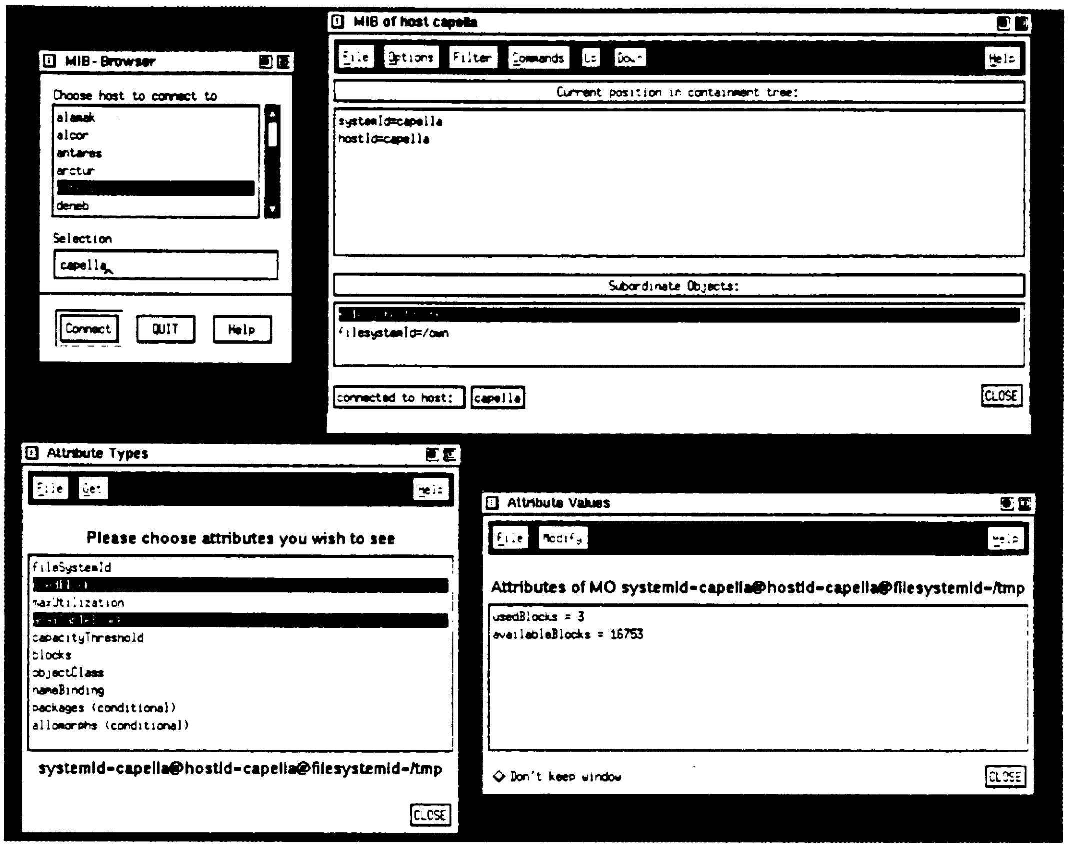


Figure 5: *Processing of MO definitions by DAMOCLES*



MIB-Browser

Choose host to connect to

alamak
alcor
antares
arctur
[illegible]
deneb

Selection

capella

MIB of host capella

Current position in containment tree:

systemId=capella
hostId=capella

Subordinate Objects:

[illegible]
filesystemId=/own

connected to host: capella

Attribute Types

Please choose attributes you wish to see

fileSystemId
[illegible]
maxUtilization
[illegible]
capacityThreshold
blocks
objectClass
nameBinding
packages (conditional)
allomorphs (conditional)

systemId=capella@hostId=capella@filesystemId=/tmp

Attribute Values

Attributes of MO systemId=capella@hostId=capella@filesystemId=/tmp

usedBlocks = 3
availableBlocks = 16753

◇ Don't keep window

Figure 6: *Example display of the MIB browser*

the root MO of the remote MIB and its immediate subordinates are shown in a separate window (the window in the upper right corner of Figure 6). Several MIBs can be inspected simultaneously in this way.

By moving a virtual pointer, which initially refers to the root MO, it is possible to select any managed object contained within the MIB in order to invoke management operations on it. The current version of the MIB browser only supports the "Get", "Set" and "Action" operation. According to the definition of the class of the selected managed object the list of attributes or actions which may exist within the MO is displayed (the window in the lower left corner of the figure). In the current version, the MIB browser only has a schema knowledge about a fixed set of managed object classes. By having access to the global management schema maintained by DAMOCLES, future versions should be able to handle every defined managed object class. Moreover, the set of known attribute syntaxes, i.e. a set of encoding and decoding functions, has to be enhanced. Currently, attributes with the syntax "string", "counter", "gauge", "threshold", "tide-mark", "object identifier" or "set of object identifier" are supported.

The MIB browser is implemented in ANSI-C and runs under SunOS. Its user interface is based upon X11R5 and OSF/Motif 1.1. The Management Support Service provided by a Management User Agent is used to examine and manipulate the contents of a MIB.

5. Future developments

The BERKOM Management Platform has been successfully used within the BERCIM project which is another sub-project of BERKOM [Tsc92a]. However, further development of the services offered by the BMSS and improvements to the features of DAMOCLES and the MIB browser are necessary:

- The MSS is to be extended by adding further OSI management functions, e.g. the event report management function [ISO91e].
- The provided library of managed object and attribute implementations is to be enlarged.
- DAMOCLES is to evolve from an MO definition tool into an MO development tool. It is planned to develop a GDMO compiler which will automatically produce C++ code from the GDMO templates. This code may be used as a skeleton for the implementation of specific managed objects. Additionally, data structures and the associated encoding and decoding functions for parameters and attributes have to be created. In order to complete an MO implementation all that is then needed is the binding to a real resource. However, the extent to which automatic code generation is possible for specific MOs must still be studied in detail.
- Currently, DAMOCLES is a single-user tool. An important work item is the development of a "Distributed DAMOCLES" which allows multiple user to work simultaneously with DAMOCLES. An issue to be studied within this framework is whether the Directory can be used for storing standardised definitions of management information.
- Moreover, besides the interactive access to the management schema, a programming interface (API) is desirable. This API is to allow components of the Management Platform to query information from the schema. Such capabilities may be feasible, for example, in order to check the validity of management operation calls.
- The MIB browser is to use this DAMOCLES API for the retrieval of management schema information. Furthermore, the creation and deletion of managed objects should be supported.

Another objective of the BERMAN work is the integrated use of Directory and OSI Management services. In the future, a new service called *Management Information Service* (MIS) is to be developed which is intended to hide the location of management information whether it is stored within the Directory or in a MIB.

References

[BER91a] BERMAN, "Guide to the BERKOM Directory, Version 2.0," in *BERMAN Project Deliverable 4, Part 1*, Berlin (December 1991).

[BER91b] BERMAN, "Guide to the Basic Management Support System (BMSS), Version 2.0," in *BERMAN Project Deliverable 4, Part 2*, Berlin (December 1991).

[BER91c] BERMAN, "BERKOM Schema Definitions for Directory and Managed Object Classes, Version 2.0," in *BERMAN Project Deliverable 4, Part 3*, Berlin (December 1991).

[ISO89a] ISO, *IS 9594-1 – Information Processing Systems – Open Systems Interconnection (OSI) – The Directory – Part 1: Overview of Concepts, Models and Services*, 1989.

[ISO89b] ISO, *IS 9594-7 – Information Processing Systems – Open Systems Interconnection (OSI) – The Directory – Part 7: Selected Object Classes*, 1989.

[ISO90a] ISO, *IS 9596 – Information Technology – Open Systems Interconnection – Common Management Information Protocol*, November 1990.

[ISO91c] ISO, *IS 10165-2 – Information Technology – Open Systems Interconnection – Structure of Management Information – Part2: Definition of Management Information*, October 1991.

[ISO91d] ISO, *IS 10165-4 – Information Technology – Open Systems Interconnection – Structure of Management Information – Part 4: Guidelines for the Definition of Managed Objects*, August 1991.

[ISO91e] ISO, *IS 10164-5 – Information Technology – Open Systems Interconnection – Systems Management – Part 5: Event Report Management Function*, October 1991.

[ISO91a] ISO, *IS 10165-1 – Information Technology – Open Systems Interconnection – Structure of Management Information – Part 1: Management Information Model*, August 1991.

[ISO91b] ISO, *IS 10164-1 – Information Technology – Open Systems Interconnection – Systems Management – Part 1: Object Management Function*, October 1991.

[Pop88a] R. Popescu-Zeletin, "A Global Architecture for Broadband Communication Systems: The BERKOM Approach," pp. 366-373 in *Proceedings of the Workshop on Future Trends of Distributed Computing Systems in the 1990s*, Hong Kong (September 1988).

[Tsc92a] V. Tschammer, H. Fredrich, L. Henckel, H. Linnemann, and D. Strick, "BERCIM – a Broadband Communication Experiment in CIM," pp. 85-97 in *Proceedings of the Eighth CIM-Europe Annual Conference*, Birmingham, UK (May 1992).

[Wit91a] M. Wittig and M. Tschichholz, "DAMOCLES: A Tool for the Definition of Managed Object Classes," in *Proceedings of the Fifth RACE TMN Conference*, London (November 1991).

The Multi-Threaded X (MTX)

Window System and SMP

John Allen Smith

Data General Corporation
Research Triangle Park, NC USA
smithj@dg-rtp.dg.com

Abstract

The current UNIX implementation of the X11R5 Window System provides only single-threaded support of client requests, event processing, and device input. Each request and event is processed one at a time to the exclusion of all other processing. This approach leads to a non-interactive server. With X clients that use the Phigs+ X Extension (PEX), X Image Extension (XIE), and integrated multimedia, the performance and usability of the X11R5 server is severely degraded.

This paper presents the object-oriented design and implementation of an X Window Server with multi-threaded concurrent support and shows how multimedia X clients can take advantage of the resulting gains in interactivity using Symmetric Multi-Processor (SMP) platforms such as OSF/1 and DG/UX. Lessons learned can be applied to client side threads design.

1. Introduction

The X Window System [Sch92a] is based on the client-server model of process interaction. This paradigm is the basis of most network communication, and is followed in X to give users the appearance of seamless applications concurrently using distributed resources. In actuality, this model permits independent applications to display data and communicate over a local-area network by means of a well-defined client/server protocol.

1.1. Limitations of X11R5

For SMP platforms, it is highly inefficient, in terms of processor utilization, to execute processes without a kernel that supports multi-threading. Since DG/UX is an SMP operating system, it treats all processors as equivalent and capable of sharing memory. The X Window System can benefit from multi-threading by partitioning concurrent tasks across the processors so that they are executing in parallel.

In particular, the X11R5 (R5) Window System includes a server that processes one client protocol request at a time. If there are multiple clients in the client queue that are simultaneously sending requests to the server, the server processes requests using a round-robin scheduling policy. Each client can have up to ten requests processed before the server preempts that client and starts working on the next client. Although the current server is implemented with a single-thread of control, it tries to give the appearance of concurrency by switching between clients after each block of requests. The issue then is that the R5 server multiplexes rather than batches the client requests.

The R5 Server creates problems for client requests that require a large amount of execution time. For instance, if a client executing Phigs/Phigs+ Extension to X (PEX) requests were to generate a structure traversal, the server would walk the entire structure tree, and depending on the size of the structures and the type of traversal, this walk could take several hours to complete. While the server is walking the structure tree, all other pending requests and input events are queued. All client activity, including the window manager, and mouse motion is frozen. This policy supports serve-to-completion rather than any type of preemption.

The other component of the R5 Window System is the X client. The R5 X client requests services of the X server, but is also limited to executing only one task at a time. A multi-threaded X client can perform concurrent activities only if the underlying Xlib libraries are made reentrant and thread-safe.

Threads are used to correct these limitations with the client and server sides of the X Window System. Multi-threading allows a greater degree of concurrency than is possible in X11R5.

Although the client and server have common threads design issues, multi-threaded X clients must be designed with the application in mind while the single X server must be designed so that it can handle the requests of any client. This makes the X server the bottleneck in X Window System performance and interactivity improvements. Since the X server is the bottleneck, and is also the harder of the two threading problems, the remainder of this paper focuses on the design of the multi-threaded X (MTX) server. Lessons learned about the design of the MTX server can be applied to the design of MTX clients.

1.2. Objectives

The primary goals of the MTX server are:

- Conform to the existing server protocol.
- Increase server interactivity.
- Efficiently use SMP workstation platforms.

The secondary goals of the MTX server are:

- Do not significantly degrade server performance from the R5 levels.
- Design the product with a CASE toolkit [Int90a] that supports the object-oriented (OO) paradigm. We feel this design approach is a natural consequence of the client/server model as embodied in the X Window System and will result in a cleaner implementation. A beneficial side-effect of using a structured methodology is automatic documentation of the MTX server as the product is developed.

2. Object Oriented Design

After the limitations of the current X server were identified, it was realized that a new design was required for a concurrent implementation. Since the Multi-Threaded X (MTX) server requires the implementation of concurrent programming ideas, we were motivated [Smi91a] to consider Object Oriented Structured Design (OOSD) techniques [Hen90a] in the project life-cycle. Software through Pictures (StP) [Int90a] is one such tool that provides a variant of the OOSD design process.

At the heart of OOSD, is the object-oriented (OO) paradigm. The OO paradigm [Kor90a] focuses on objects and emphasizes the relationship between those objects as fundamental to the system architecture.

Objects are treated as the basic run-time entities in this design approach. In the X server, these objects include the window, screen, region, drawable, pixmap, visual, device, cursor, colormap, fonttable, resourcetable, selection, client, and others. Each of these objects work at different layers of the server. For example, the region object is active at the machine independent (MI) layer and not the device independent (DIX) layer while the window is primarily a DIX object.

The objects are pieces of the design that are conceptually grouped into classes. Each class defines a set of permissible objects that are eventually implemented as user defined types. Hence, a class becomes an implementation of an Abstract Data Type (ADT). Further, a good design results in the encapsulation of the ADT and all access to that ADT through a monitor.

Objects and classes are the first two facets of OOSD. Inheritance, polymorphism, and dynamic binding are the others. Inheritance allows the reuse of objects and other software entities such as modules. The current X server defines the Window and Pixmap as Drawables. The Window and Pixmap classes are derived from the more common class of drawables. Code to render to a drawable can be used for both a window object as well as a pixmap object because render is a polymorphic operation on the class drawable. Lastly, the function pointers defined in the routing of protocol requests is an example of using dynamic binding.

In addition, an Entity-Relationship diagram (ERD) was developed as part of the MTX data object design. This ERD conforms to the methodology used by StP and is described in the MTX Component Design Specification.

3. Concurrency Mechanisms

Object-oriented techniques promote program modularity through the need for data abstraction. Concurrency is also concerned with data abstraction since this concept allows for the efficient management of resources in an environment requiring resource sharing and distributed problem solving. Data abstraction in turn leads to a development based on the concept of objects. Hence, we discover that concurrency is a natural consequence of the OO paradigm.

Solving problems using concurrency [Agh90a] can be divided into three types, pipeline, divide and conquer, and cooperative. The cooperative problem solving technique is very close to the description of how the client/server model operates. Since the MTX server is based on the client/server paradigm, we concluded that the cooperative technique is our best choice for the MTX server.

Since the application of cooperative problem solving involves the sharing and synchronization of resources among objects, we manage those resources through the use of mutexes and condition variables. These synchronization tools are necessary in order to avoid deadlock and starvation.

In a concurrent environment such as DG/UX, access to shared data must be arbitrated by some [Kel89a] access control policy. Mutual exclusion is one such policy that prevents two concurrent activities from accessing the same shared resources at the same time. These shared resources may include data as well as code segments called regions.

Mutual exclusion prevents activities from colliding over regions. If we want to prevent an activity from continuing until some general condition is met, then we must synchronize that activity with the condition. Hence, synchronization is a generalization of mutual exclusion. The use of conditions implies a causal dependence on the execution of activities. For instance, reading the drawables associated with a window object depends on the condition that the window object exist. If the create activity has not completed access to the window object, then the read activity will conditionally wait for synchronization from the create.

Read and write access to data structures that must be shared by many activities can be effectively managed using the Hoare monitor synchronization mechanism. This programming construct [Fin88a] encapsulates the shared data object in a protective wrapper. The wrapper enforces mutual exclusion by allowing only one activity access to the shared data object. The monitor is a global object that advertises all of the public access routines of the object to the current set of activities.

The Hoare monitor is a poor performer when there are many more readers than writers. We would like to allow multiple reads to occur concurrently while allowing only one write access at a time. This solution to the reader-writer problem is handled nicely by the crowd monitor. The crowd monitor has guard procedures that decide when each activity may enter or leave a reader or writer access group. These guard procedures arbitrate access to the protected data object. The crowd monitor determines which group currently has access permission, and queues those activities that must wait.

Also, activities should be able to exclusively lock multiple data objects before execution proceeds. For example, a ReparentWindow request requires that particular window objects be exclusively locked before a window is moved in the window hierarchy. This situation is similar to the readers/writers problem. Also, there are event queues that are filled by input device activities and drained by activities that report the events to the appropriate clients. This is the producer/consumer problem.

4. Threaded X

The previous sections discussed how the MTX server is based on the client/server model and how this model is best expressed using an object-oriented methodology. In turn, this server model was shown to require concurrent activities that are best managed using concurrent programming constructs such as crowd monitors. This section discusses how the OO paradigm and concurrency can be implemented with DG/UX threads [Alf91a] and locking primitives.

A thread is a sequential flow of control. There may be more than one thread executing within a process. Each thread shares the address space of the process with all other threads that are created in the process. A thread has its own execution stack, errno, and thread id. The benefits to using threads are that disjoint sets of code may be executed in parallel while sharing a common code and data address space. Using threads increases the concurrency and interactivity of a process, and allows for more efficient use of multiprocessor architectures.

DG/UX was designed to support multiprocessors, and conforms to the POSIX 1003.4a Pthreads standard. This standard supports,

- The creation, control, and termination of threads;
- The use of synchronization primitives by threads in a common, shared address space.

There are several issues to consider when threading the X server.

- What are the implementation constraints
- What level of granularity is needed to enforce resource locking
- What mechanism should be used to pass messages between threads
- What is the impact of reentrancy.

4.1. Constraints

The design of the MTX server results in a list of objects and their related functions. The server must be designed so that those functions that do not collide over data objects execute in parallel, and those functions that are related are given fair access to data and are synchronized when required. There are several constraints that affect the implementation of the server functionality.

4.1.1. Protocol Conformance

The MTX server must continue to conform with the existing X11 core protocol. The X11 core protocol [Sch92a] treats the service given to any one client as separate from all other clients. Hence, the protocol requests from each client must be treated as atomic and serial.

The MTX server must simulate the atomic behavior by not allowing different threads to collide over the same data object. In this way, the server executes each client request as if it were the only one being serviced.

The protocol demands that requests from the same client be executed in serial order. The effect of this requirement is that the MTX server should concurrently execute requests from the same client only if the appearance to the user is one of ordered changes. So, a client that generates a request for a font load and also a request to install a new colormap can expect the server to execute these requests concurrently since the font and colormap resources are distinct. Requests to change window attributes and reparent a window must occur in serial order since these requests access the same window resource.

4.1.2. System Resource Usage

The primary goal of the MTX server is to increase interactivity but not at the expense of the R5 server level of performance. Implementation techniques that would increase the level of interactivity but degrade MTX performance below that of R5 were not used. Related to performance is memory usage. A server that uses great amounts of memory or generates excessive paging is not acceptable. Partitioning the server data and code into logical pieces should keep memory requirements at an acceptable level.

4.2. Resource Locking

Use of sharable data objects pervades the MTX server. The concurrency paradigm refers to these as sharable resources and suggests that access to the resources be synchronized between multiple contending threads. The MTX server wants to avoid having two clients change the same colormap at the same time.

During the OO design of the MTX server, an object hierarchy was generated along with a description of how the objects should be accessed. Part of the process of building the list of objects is discovering the locking requirements on those objects. Objects may theoretically be accessed in any of the read and write mode combinations, but practically, we want to impose resource locks to insure mutual exclusion while ensuring maximum concurrency. Locks on shared objects are enforced by crowd monitors. For example, the Resource Database Monitor protects the Client/Resource Table.

Another issue is the lock granularity. Lock granularity can be defined in terms of the size of the resource to be locked and the length of time that a resource is protected from mutual access (i.e from lock to unlock). Granularity is either fine or coarse grained. In fine grained locking, a small resource is locked frequently for very short periods of time. In coarse grained locking, a large resource is locked for long periods of time, but much more infrequently. Determining the lock granularity of each object is dependent on the expected use of that object and the read/write access level required. The granularity for each object is coded in that object's access routines as defined by the monitor.

Choosing the correct level of lock granularity for each resource is important in maximizing performance and interactivity. For instance, if multiple threads reading trackball input were to lock around each read while updating the device record, then more time would be spent in locking/unlocking than if the thread were to execute a lock, multiple reads, and then an unlock. Since there is overhead in locking and unlocking, fine grained locking consumes many system resources but provides quick access to objects. Generally, coarse grained locking consumes fewer system resources but results in a higher probability of contention of threads. Fine grained locking increases interactivity at the expense of performance and memory usage while coarse grained locking increases performance at the expense of interactivity.

To avoid deadlock, each lock is accessed in a strictly defined order. For acquisition of nested locks, lock precedence insures that deadlock will never occur in the server. In addition, deadlock avoidance keeps the server from having to manage costly rollbacks when data changes.

In order to increase the interactivity of the server but not degrade performance, the granularity and position in the lock hierarchy is determined for each server object while keeping the above tradeoffs in mind. A important factor in this determination is the expected usage pattern for each object in relation to each protocol request. Frequency of object use directly affects how that object is accessed.

4.3. Reentrancy

Since threads share code and global data, the procedures that are callable from within threads must be reentrant or within a critical region. Requiring reentrancy means that a large part of the R5 server code can not be reused, even at the MI layer.

4.4. Development Platforms

The MTX prototype was developed on an Omron Luna88k QuadProcessor workstation running Carnegie-Mellon's Mach [Ras87a] distributed systems kernel. Full implementation of MTX is proceeding on workstations that support OSF/1 and DG/UX. Mach uses Cthreads [Coo88a] while DG/UX and OSF/1 support the pthreads POSIX 1003.4a standard library.

Both Cthreads and it's descendent Pthreads provide a high level C programming interface to the low level kernel thread primitives. These thread packages support multiple threads of control for concurrency and parallelism through shared variables, mutual exclusion of critical regions, and condition variables for thread synchronization.

4.5. MTX Threads

The functionality of the MTX server is approximately the same as that defined by the current R5 server. Although the functionality is equivalent, the implementation is not. The MTX server is implemented with threads and concurrency support whereas the R5 server has a single thread of control. The current X server looks for client requests, input events, and new client connections within the dispatch code. By combining these three unrelated functions into one serial loop, the performance of the server suffers. We can divide these functions into separate flows of control [Tev87a] by using threads.

A sample implementation of the use of threads in the MTX server is described in the following sections and is diagrammed in Figure 1. In the diagram, circles indicate threads, boxes indicate external objects, dark directed lines indicate inter-thread data flow, grey directed lines indicate data object access, dashed directed lines indicate thread creation, and two horizontal lines enclose data objects.

4.5.1. Main Server Thread (MST)

The MST manages the global MTX server environment. This thread initializes the MTX server and the input devices, and creates the Device Input Thread. It also creates the Client Connection Thread so that X Clients can establish communications with the server. If the MST receives a wakeup command, it cleans up the global MTX server environment and checks to see if the server should reset or terminate. If it should reset, the MST reinitializes the environment and starts the MTX setup over again.

The functionality of the Main Server Thread is similar to that found in R5's main.c. In the R5 implementation, however, new client connections, new protocol requests, and new device input are processed serially in a complicated dispatch loop. The complexity of the overloaded dispatch loop is a direct result of the need to handle many different asynchronous functions in one location. This area of the server was greatly simplified in MTX by creating separate threads for each of these functions.

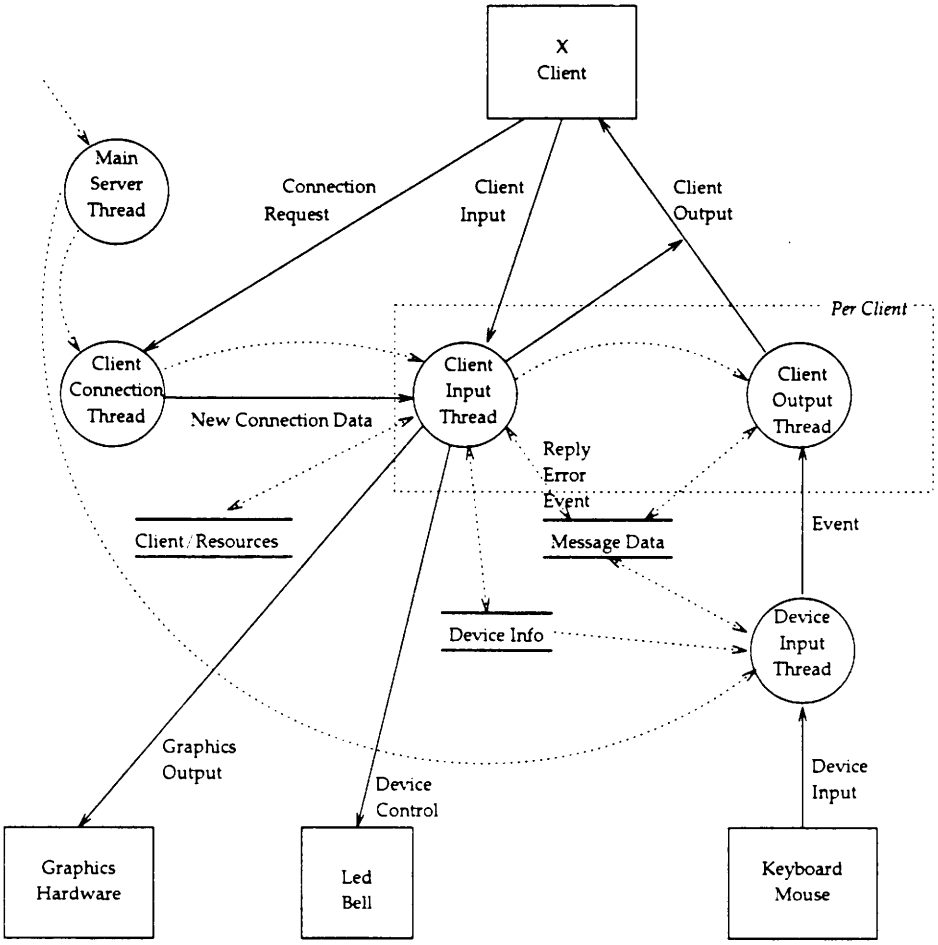


Figure 1: *MTX Server Threads*

4.5.2. Client Connection Thread (CCT)

New client connections are processed by the CCT. This thread accepts the client connection request, validates the connection dependent information (such as socket or shared memory), and creates the Client Input Thread to handle new requests if validation was successful.

4.5.3. Client Input Thread (CIT)

There exists a CIT for each client connected to the MTX server. The function of this thread is to process requests for its assigned client while adhering to the atomicity and serial execution constraints described in the protocol. When the CIT initializes, it determines byte order for the client, accepts the client connection, and sends server information to the client.

After thread initialization, this thread blocks until there is a client request. This thread also handles the byte ordering of the OS connection to the client. The request is processed similar to the function pointer mechanism used in the X11R5 dispatch loop.

All CITs are able to process any protocol request. Since each protocol request accesses both shared and private resources, locking policies are invoked to protect multiple CITs from accessing the same resources. Resources in the Resource Database are considered to be shared objects. Access to these shared objects are through the Resource Database Monitor.

If a CIT wishes to gain access to event related objects (such as the event mask, GrabRec, or DeviceIntRec), the CIT requests access through the Device/Event Object (DEO) Monitor. The DEO Monitor insures exclusive access to the event related objects and blocks other requesting threads to enforce serial access to the event related objects. The DEO Monitor returns control to the requesting CIT when the monitor finishes.

All events that originate in the CIT are processed by the DEO Monitor before routing to the X Client. Errors and replies are sent directly to the X Client.

The CIT also handles client shutdown activities. When a client dies or is killed, the appropriate CIT frees resources from the Resource Database and frees local data structures. If the CIT represents the last client connected to the MTX server, then it acquires a lock to insure that no new connections are made to the server while it signals the MST to wakeup and reset the server.

If there is output to be sent to any devices, this thread manages that activity through the Device Dependent X (DDX) layer. This includes rendering to the graphics output device (e.g. framebuffer or graphics processor) and generating feedback for the feedback devices (e.g. led and bell). Rendering to the graphics output devices is a frequent operation and could generate an unacceptable number of thread context switches if this functionality were placed in a thread other than the CIT.

Only one input buffer exists per CIT rather than a pool of buffers as currently exists in the R5 server. This approach allows us to localize the input buffer processing to individual CITs and remove dependence on the slower `select()` call.

4.5.4. Client Output Thread (COT)

COTs are created whenever the CIT or DIT (producer threads) must buffer messages for delivery to X Clients. The DIT always creates a COT since we do not want the DIT in a arbitrarily long blocking write to the X Client when input could be processed. CITs usually deliver messages directly to the X Client except when buffering would be more efficient.

The producer threads deliver messages to the COT via an output buffer. The COT flushes messages from the output buffer to a client's socket when necessary. If the socket is full, the thread will block.

The COT decouples message delivery from more essential server operations. When the server is very busy, COTs are created more frequently. Likewise, when the server is at low throttle, our design bypasses the overhead of creating a separate thread if that would be more efficient.

4.5.5. Device Input Thread (DIT)

The DIT waits for device input and creates a COT thread to route the device events to the appropriate X Clients. The DIT(s) are created at server initialization in the MST.

There is at least one DIT to handle all device input to the server. Device input from the mouse, keyboard, trackball, etc. is currently placed in the ProcessInputEvents function. This function has device independent and device dependent code based on the type of device. The DIT accepts input from all registered devices.

4.6. MTX Monitors

Read and write access to data structures that must be shared by many threads can be effectively managed using the Hoare monitor. This programming construct encapsulates the shared data object in a protective wrapper. The wrapper enforces mutual exclusion by allowing only one thread access to the shared data object at any one moment. The monitor is a global object that advertises all of the public access routines of the object to the current set of threads.

The Hoare monitor is acceptable if the object requires exclusive access. But, it is a poor performer when there are many more readers than writers. Typically, we would like to allow multiple reads to occur concurrently while allowing only one write access at a time.

This solution to the reader-writer problem is handled nicely by the crowd monitor. The crowd monitor has guard procedures that decide when each thread may enter or leave a reader or writer access group. These guard procedures arbitrate access to the protected data object. The crowd monitor determines which group currently has access permission, and queues those threads that must wait.

A simple Hoare monitor is used to control access to the DEO. A crowd monitor is used to control access to each client's resources in the Resource Database.

4.6.1. Device/Event Object (DEO) Monitor

The DEO Monitor arbitrates exclusive access to the device database and any event related data objects. This includes the window event mask, the window optional donotpropagate mask, the grab record, the deviceint record, and others.

The DEO Monitor also allows serial access to device and window objects when device events are propagated from the event window up the window tree to the root.

4.6.2. Resource Database (RDB) Monitor

All shared resources such as windows, pixmaps, graphics contexts, colormaps, etc. are kept in the RDB. When a client creates a resource, the client id is used as a unique index into the table. For each client, there is a hash table of resources where the resource id is used to hash into a table of buckets that contain a list of resources owned by that client. Each entry in the list in turn points to the actual resource object.

The resource-id is the unique identifier that is both understood by the Xlib and the MTX server sides of the client/server model. Since the X protocol must be preserved and the format of this resource-id is tied to the protocol, we must preserve this table in the MTX server.

The RDB Crowd Monitor maintains this table by processing add, read, and free requests from threads that manipulate these shared resources. The CITs and the DEO Monitor can manipulate shared resources and therefore use this monitor.

4.6.3. Pending Operation Queue (POQ) Monitor

The POQ is a database describing how server objects are locked by currently running server operations (such as CITs or the DIT). By looking in the POQ, the server can determine if a new protocol request from the X Client would conflict with any other running thread. If no conflict exists, the server operation puts an entry in the POQ, and continues execution. If a conflict occurs, the conflicting request is blocked until the conflict is resolved. When execution of the operation is complete, the thread removes the POQ entry and waits for further requests from the X Client.

5. Multimedia Applications and X

One of the original driving problems for the design of a multi-threaded X server was the efficient use of PEX. Multiple PEX applications severely degrade the interactivity of the current X server as each 3D image is rendered. An application that uses all of the PEX graphics features, such as lighting, shading, and depth cueing, can make the other day-to-day applications unusable.

We can also envision that other proposed media extensions to X will compete in a similar way for server bandwidth.

The Video Extension (VEX) [Bru90a] was proposed as an extension to X to provide a standard interface to video operations. These operations include video input and output, manipulating a video picture on the screen, acquiring digitized pixels from video frames, cutting portions of the screen for video recording devices, and control of external video devices. Full television resolution video processing is not yet a feasible application because video signal rates are still high compared to disk storage rates. But, applications that require lower video rates such as video post-production, simulation, video teleconferencing, and image processing could benefit from the use of VEX.

The X Image Extension (XIE) [Web91a] is motivated by the growth in applications requiring efficient image rendition, document image management, and interactive continuous tone color enhancement and analysis. Cartographic and Geographic Information Services (GIS) applications would benefit mostly from the XIE standard.

By integrating PEX, VEX, and XIE applications with those requiring audio and graphic capabilities, the X server is able to support a full range of multimedia services. This paper has demonstrated that the current X server does not give PEX users, let alone VEX and XIE applications, the performance and interactivity that they should expect. The MTX server attempts to address this problem by providing a server that can take advantage of SMP capabilities on platforms designed to support multimedia services.

A multi-threaded DG X server allows users to manipulate multimedia applications while simultaneously reading mail or news, and editing a emacs document. Users can expect that highly interactive applications can coexist in an environment with reasonable performance if the server providing these services is designed from the beginning to solve these problems of concurrency.

6. Conclusion

This paper has shown how using PEX and multimedia services render the current X server incapable of adequately supporting these services. The reason for this breakdown in service is due to the single-threaded, one request at a time, implementation of the current X server.

We have shown that in order to provide sufficient services in a client/server environment, the server must be designed with concurrency mechanisms in place to take advantage of SMP and multiprocessor operating systems. These concurrency mechanisms are best implemented after an object-oriented approach uncovers the objects and their inter-object relationships in the server.

The MTX server is being developed with concurrency mechanisms to solve the problems of interactivity and performance that can be expected from high demand applications such as PEX.

No project of this complexity can proceed without proper tools. StP was chosen as the CASE tool to automate design and implementation. The Mach kernel was chosen for implementation of the MTX server prototype, while final development is proceeding on DG/UX, a pthreads capable kernel.

Although the MTX server carries the overhead of thread creation and object locking, the more efficient threads design has allowed the MTX server to perform favorably compared with the R5 server. Interactivity has been increased without sacrificing performance.

Acknowledgements

This project is being developed in association with the MIT X Consortium. Thanks go to the MIT X Consortium members and staff for their help and support of the MTX project. Finally, greatful recognition is given to the DG MTX project team for making this paper possible.

References

[Agh90a] Gul Agha, "Concurrent object-oriented programming," *Communications of the ACM* **33**(9), pp. 125-141 (September 1990).

[Alf91a] Bob Alfieri, *Threads Functional Specification,* Data General Corp. (July 1991).

[Bru90a] Todd Brunhoff, "Vex Your Hardware VEX Version 5.6," in *Xhibition 90 Conference Proceedings, San Jose, CA* (May 1990).

[Coo88a] E. Cooper and R. Draves, "C threads," in *Technical Report CMU-CS-88-154*, School of Computer Science, Carnegie Mellon University (February 1988).

[Fin88a] R. Finkel, in *An Operating Systems VADE MECUM, Chapter 8*, Prentice Hall, Englewood Cliffs, NJ (1988).

[Hen90a] B. Henderson-Sellers and J. Edwards, "The object-oriented systems life cycle," *Communications of the ACM* **33**(9), pp. 142-159 (September 1990).

[Int90a] Interactive Development Environments, *Software through Pictures User Manual*, March 1990.

[Kel89a] M. Kelly, "Multiprocessor Aspects of the DG/UX Kernel," pp. 85-99 in *Proceedings of the Winter 1989 USENIX Conference*, The USENIX Association, Berkeley, CA. (1989).

[Kor90a] T. Korson and J. McGregor, "Understanding object-oriented: A unifying paradigm," *Communications of the ACM* **33**(9), pp. 40-60 (September 1990).

[Ras87a] R. Rashid et al., "Machine-independent virtual memory management for paged uniprocessor and multiprocessor architechures," in *Technical Report CMU-CS-87-140*, School of Computer Science, Carnegie Mellon University (July 1987).

[Sch92a] R. Scheifler and J. Gettys, in *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD*, Digital Press (1992).

[Smi91a] J. Smith, *Engineering a Multi-Threaded X Server,* Xhibition Technical Conference, San Jose, CA (1991).

[Tev87a] A. Tevanian et al., "Mach Threads and the UNIX Kernel: The Battle for Control," in *Proceedings of USENIX Summer Conference*, The USENIX Association, Berkely, CA. (1987).

[Web91a] John Weber, "XIE, a Proposed Standard Extension to the X11 Window System," in *5th Annual X Technical Conference Technical Papers, Boston, MA.* (Jan 1991).

Experiences in Fine Grain Parallelisation of Streams-Based Communications Drivers

Ian Heavens

Spider Software
Edinburgh, Scotland
ian@spider.co.uk

Abstract

The Streams architecture has been extended (as **Parallel Streams**) to take advantage of symmetric multiprocessor environments in UNIX SVR4/MP and other operating systems. This paper describes our experiences in providing fine grain parallelism for protocol drivers in Parallel Streams. A multiprocessor simulator forms part of the development environment. The design of the TCP and IP drivers is discussed in detail, focussing on certain interesting race conditions in TCP. Performance tuning, measurement and analysis are covered. The parallel TCP/IP stack has been ported to Windows/NT™.

1. Introduction

The growth of distributed computing, particularly networked file systems, and high bandwidth media, mean that networking performance in operating systems is increasingly important. Shared memory symmetric multiprocessors use off-the-shelf components, are extensible, have the programming advantages of a global address space, and contain some fault tolerance. They are increasingly popular for high performance systems, ranging from tens of processors utilising a time shared bus for processor-memory interconnect (such as the Sequent Symmetry series [Seq87a]), to hundreds of processors with an interconnection network or other highly scalable architecture (such as the BBN TC2000 [BBN90a], NYU Ultracomputer [Got83a], and Stanford Dash [Len92a]).

The networking drivers are candidates for parallelisation in a symmetric multiprocessor architecture. Protocol processing is only one bottleneck in network I/O. As important, if not more so, are a fast network controller designed for a multiprocessor, adequate memory and I/O bus bandwidth, an architecture that minimises data copying, and an appropriate memory allocation policy [Cla91a, Rud90a, And91a]. These factors have been modelled in detail [The92a]. Previous approaches to parallel processing of network I/O have addressed the needs of gigabit networks [Jai90a], utilised hardware optimised for a specific protocol

[Kan88a], or dedicated processors to particular protocol functions [Zit89a, Jen88a, Gui90a]. The symmetric multiprocessor architecture is not optimal for standalone communications processors. Fast processors can handle even fibre optic bandwidth, and an asymmetric approach is better suited for high performance packet switches. It may be preferable to run host protocol processing on a front end, although symmetric multiprocessing offers undeniable flexibility.

We consider the need for providing scalable performance for standard networking protocols (TCP/IP, X.25 and OSI) on a conventional symmetric multiprocessor, at current network bandwidth (up to 100 Mbit/s). Our definition of high performance is the completion of many concurrent requests for network I/O by users in little more than the time taken to satisfy a single such request. Increased throughput or decreased latency for a single user is a secondary goal. On an otherwise lightly loaded system, the throughput per transport connection should degrade only slightly as the number of simultaneous connections increases towards the number of processors, with a ceiling of the network bandwidth.

To achieve this performance, consider the following loosely defined hierarchy of parallelism. A protocol machine provides the service of a single communications protocol.

1. Serialise access to the protocol stack. No two processors are simultaneously processing any part of the protocol stack.
2. Serialise access to a particular protocol machine, but permit concurrent processing of different protocol machines (*vertical parallelism*).
3. Serialise processing of outgoing packets on a particular transport connection, but permit concurrent processing of different transport connections in a particular protocol machine (*horizontal parallelism*). Permit concurrent processing of incoming packets on different transport connections; demultiplexing at the transport layer means that all incoming packets are processed concurrently up to this point.
4. Permit concurrent processing of outgoing and incoming packets on a particular transport connection.
5. Permit concurrent processing of different parts of the same packet, outgoing and incoming [Jai90a].

The third level in the above hierarchy, which we call *fine grain* parallelism, satisfies the throughput goal.† This can be provided for protocols in the Streams environment, with standard parallel extensions to Streams. The rest of this paper describes our experiences in doing this for the TCP/IP protocol stack.

Background

The Stream I/O system (**Streams** [Rit84a]) is an environment allowing modular development of layered communications protocols. First made commercially available in UNIX V.3 [AT&86a], extensions for multiprocessor support were carried out by Sequent Computer Systems and incorporated with minor modifications in UNIX SVR4/MP [AT&89a] as **Parallel Streams**. An introduction to Streams and discus-

† [Jai90a] defines this to be *coarse grained*, reserving fine grained for level 5. Most current protocols cannot efficiently exploit this level of parallelism. The increasing relative cost of communication versus computation in symmetric multiprocessors means that fine granularity may not be appropriate for large multiprocessors [Mar92a].

sion of these extensions may be found in [Gar90a]. Appendix A explains relevant Streams terminology.

The Streams environment has also been independently implemented as *SpiderStreams* by Spider Software.[†] This has been ported to many other operating systems, such as QNX [Hil92a], pSOS [SCG90a], and Vrtx [Rea88a]. Multiprocessor support conforming to SVR4/MP Streams has been added to SpiderStreams, and this has been ported to Windows/NT [Cus92a] on Intel 80486 and MIPS R4000 architectures, and to the Sequent Symmetry™ 2000 [Seq87a] as a multithreaded user process.

Within Parallel Streams, the protocol drivers must themselves synchronise access to shared data. This has been described for TCP/IP in a non-Streams kernel [Boy89a], and for Streams drivers [Dov90a, Nuc91a]. The requirements of supporting many operating systems and architectures, allowing future functional enhancements, and maintaining a scalable parallel protocol stack dictate the following goals:

1. All multiprocessor and single processor environments should share a common code base. Performance in a single processor environment should not degrade.
2. Modifications to the current implementation should be limited. As our Streams-based protocol drivers were developed at Spider, we are not constrained to track future releases of operating system code, such as Berkeley TCP/IP. Some performance enhancements and minor algorithmic modifications were made for better support of multiprocessor environments, but we did not wish to rewrite the protocol drivers.
3. There should be minimal assumptions about the underlying architecture:
   - There is hardware support for interprocess coordination.
   - That memory reads and writes are atomic for 8 bit and aligned 16, 32 bit and word quantities.[‡]
   - There may be hardware support for distribution of interrupt servicing among all processors.

   The protocol stack should be scalable up to the number of processors that can efficiently support other parallel applications.

To date, the parallel implementation of TCP/IP *(SpiderTCP/MP)* has been ported to *SpiderStreams* in *Windows/NT*, and user space *SpiderStreams* on the Sequent Symmetry. A port to SVR4/MP on an Intel 80386 is in progress.

The next section reviews the architecture of Parallel Streams. Section 3 describes the development environment, and Sections 4 and 5 discuss the design of multiprocessor support for TCP/IP protocols in Parallel Streams. Section 6 describes performance measurement and analysis, tuning for the Sequent Symmetry. The paper concludes by assessing achievement of the above goals, and points to future work.

[†] There are several other independent implementations of both single processor and multiprocessor Streams, although we claim to have the earliest (1988). Appendix B provides an overview of the architecture of SpiderStreams.

[‡] This is true for most symmetric multiprocessors, with the notable exceptions of early versions of the BBN Butterfly, and the Alliant FX/8. Relaxing this assumption precludes much algorithmic lock avoidance.

2. Parallel Streams

Parallel Streams supports fine granularity in Streams-based drivers, through the concurrent execution of Streams *service procedures* operating on different queues in the same module or driver. This *horizontal parallelism* is necessary for scalable performance; in addition, *vertical parallelism* is supported, in that different drivers can execute in parallel.

SpiderStreams is upwardly compatible with SVR4/MP Streams, with two enhancements:

1. Levels of parallelism

The protocol drivers may run at one of several levels of parallelism. The level is an additional field in the driver *qinit* structure, which holds other driver information. **Full Parallelism**, the default, is equivalent to the SVR4/MP Streams architecture; **Subsystem Parallelism** serialises all *service* procedure execution on any queues in a subsystem, defined as a stipulated collection of modules and drivers. The driver will run safely on a multiprocessor without explicitly synchronising access to its private data.† Streams drivers in different subsystems execute concurrently, so there is a gain from *vertical parallelism*. Drivers supporting both levels may be mixed, allowing incremental development of fine grain parallelism.

2. Asynchronous close without context switch

A mechanism has been added to support *asynchronous close* of a stream without recourse to *sleep* and *wakeup*. These cannot be provided in context-free environments, and their use complicates the locking in Parallel Streams drivers, as spin locks cannot be held across *sleep* calls. In fact, SVR4/MP allows this, as it automatically releases locks before *sleep* and reacquires them on *wakeup*. When the application closes a stream, the Stream head issues a special I_CLOSE *ioctl*,‡ and keeps the stream open. The *ioctl* is acknowledged asynchronously by the driver, and at this point, the stream is closed down, and the driver close routine is called and returns immediately.

The rest of this paper assumes the Parallel Streams architecture as defined in [Gar90a], with the optional SpiderStreams extensions. We merely note that this architecture is a fertile subject for further study itself. Issues include the following.

- Scalability of Parallel Streams itself is a more challenging problem, as all Streams drivers share access to message, timer and other free lists, queues and streams. In contrast, the driver's private structures are only shared by *put* and *service* procedures operating on that driver's queues.
- Serialisation of *service* procedure execution of messages on the same queue, while appropriate for protocols containing much connection state, does not allow connectionless protocols such as UDP to process datagrams in parallel. In addition, queuing messages to a *service* procedure in a multiplexed protocol like IP terminates the parallelism of transport layer processing, as later message processing is serialised.

† Other schemes exist for running Streams modules without adding fine grain parallelism, for instance [Kle92a], where they are referred to as *MultiThreaded(MT)-unsafe.*

‡ Only drivers that register for this at open time via an M_SETOPTS message receive it. Otherwise, the driver close routine is called as usual, so that orthodox Streams drivers do not get confused.

- The SVR4/MP architecture does not provide cross-multiplexor flow control, nor protect against race conditions in multiplexors during stream close. This is because it is not possible to trace the path traversed by a message across a multiplexor without access to protocol-specific addresses contained within the message. Streams itself does not know when a *put* procedure executing within a driver demultiplexes to an upper or lower stream, nor the stream on the other side of the multiplexor where a message originated. Similar considerations affect interrupt handlers, *timeout* routines, and *bufcall* and *esballoc* callback routines. It is necessary to announce the entry into a stream or queue via a Streams primitive; this enhancement was adopted in Plan 9 [Pre90a] and SunOS 5.0 [Sun92a]. It is then possible to serialise all *service* and *put* procedures operating on input and output sides of the same stream, so that data private to the stream does not require synchronisation.[†]

3. Development Environment

Introduction

Parallel support was debugged and tested on a multiprocessor simulator and an implementation in user space on the Sequent Symmetry. A previous implementation of uniprocessor Streams as a UNIX V.3 user process was enhanced to provide a similar development environment for parallel Streams drivers. Fine grain locks were added to each driver separately, using drivers running at both **full** and **subsystem** parallelism as defined in Section 2. In addition, both development environments allowed multiple instantiations of the Streams environment to run in a stable operating system, permitting independent work on the same machine. Initial development and informal testing were carried out using the simulator, and final testing was done on the Sequent Symmetry. The implementation was then ported to *Windows/NT* by Microsoft Corporation.

Multiprocessor Simulator

A multiprocessor architecture was simulated by a server process containing the Streams environment, with semaphores and coroutines based on *setjmp* and *longjmp*, similar to those implemented by [Pik92a]. Applications communicate with the server by emulating QNX intertask communication [Kol89a], using message queues in UNIX System V. The system comprises several processes, shown in Figure 1. A separate process handles messages from applications, as the server cannot sleep in a system call without blocking all coroutines. This *message receiver* process communicates with the Streams server using a buffer pool and two circular buffers in shared memory, one to pass application requests, and one to return pool storage (see Figure 1).

The coroutines within the Streams server process yield control when blocked on a semaphore *(cr_p)* and by an explicit instruction *(cr_switch)*. Streams primitives always call *cr_switch*, ensuring a high degree of concurrency. Some parallel extensions were made to the GNU debugger [FSF87a] to provide per-coroutine stack traces.

[†] See *Queue-pair safe* modules in [Kle92a].

The simulator-based TCP/IP protocol stack may run side-by-side with the in-kernel TCP/IP, ensuring stable networking during development. This is done by appropriating unused Ethernet types for IP and ARP, so that both stacks may multiplex onto the same Ethernet driver. A daemon transfers packets between the Ethernet driver and the Streams server. An IP router with a Streams conversion module swaps Ethernet types and permits communication between the simulator TCP/IP and a conventional TCP/IP stack. The simulator has been ported to Interactive System V/386 Release 3.2 on Intel 80386 and 80486, ICL DRS/NX on a Motorola 68000, and RiscOS 4.52 on a MIPS RS2030.

Sequent Symmetry

Streams was ported to a multithreaded user process on an 18-processor Sequent Symmetry 2000/700 running DYNIX 3.1.4 on 80386s, and an 8-processor Sequent Symmetry 2000/750 running DYNIX/PTX 3.2.0 on 80486s. Several threads are dedicated to handling message from applications, while the rest execute Streams *service* procedures and callbacks. The *service* procedure threads block on reading from a pipe, and are woken up by writing data to the pipe when a queue is first scheduled and on *bufcall* callback. Client applications and the server communicate via System V message queues. The architecture for interfacing to the Sequent Ethernet driver is identical to that used in the simulator. A parallel version of **dbx** supplied with the Sequent Symmetry was used for debugging.

Debugging

Parallel debugging proved to be easier than anticipated. The user space environment provides a stable platform and fast build route. It allows use of familiar source level debuggers, breakpointing, easy process termination and examination of core dumps. The ability to run the simulator on most of our workstations greatly speeded development.

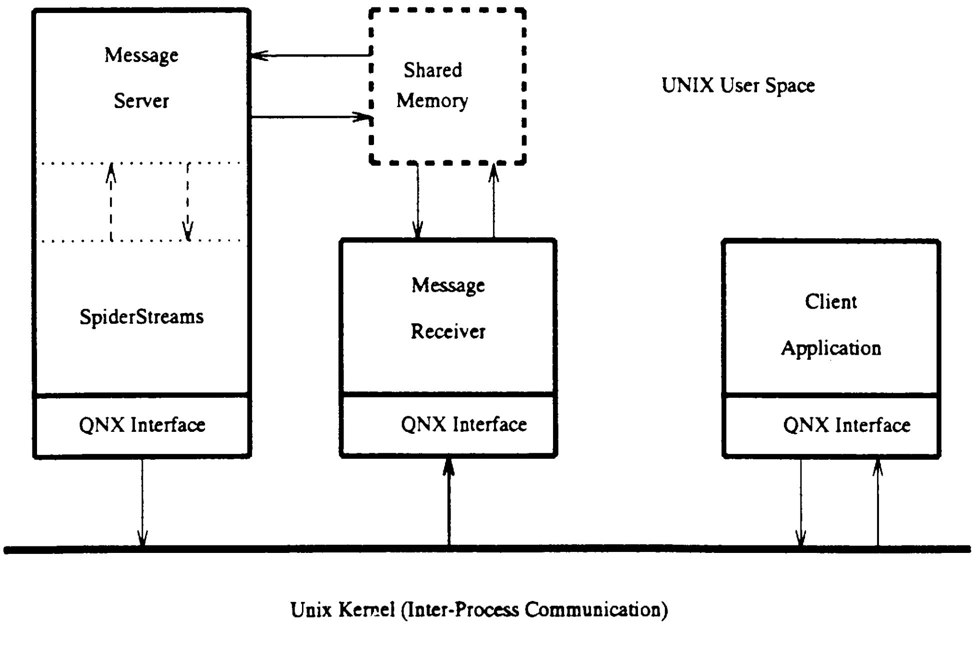


Figure 1: *Architecture of Multiprocessor Simulator*

Deadlock detection was helped by the inclusion of lock-specific trace information, notably the file name and line number where the lock was last acquired. A deadlocked thread hangs the server process in the simulator, or spins indefinitely on the Symmetry; it can be aborted and the problem rapidly isolated by inspection of stack trace and file name and line number associated with the lock. Additional runtime checking, static analysis and production code diagnostics are described in [Pea92a, Pac91a, Eyk92a], but are probably unnecessary for debugging in a standalone Streams environment; almost all deadlocks were detected in the simulator.

Experiences in porting to Windows/NT revealed some weaknesses in the development environment. The arrival of network interrupts is not adequately simulated. Currently these are treated identically to application requests. Interrupt levels have no meaning in a user context, although some detection of incorrect interrupt level setting is possible. The scheduling policy, two FCFS queues for application messages and *service* procedures, is inflexible. A configurable policy that permitted priority scheduling of user level threads, network interrupts or Streams *service* procedures would better reflect that adopted in the target architecture.

4. Design of Parallel Drivers

Optimising Single Processor Performance

The performance of algorithms with a high overhead on a single processor degrades in a multiprocessor environment. The machine is likely to be more heavily loaded, lists and tables are larger, and linear searches take longer. Critical sections are serialised, so that contention adds a further penalty. The UDP control block chain was the only such structure utilising a linear search, and has been converted to a hashed lookup (TCB chain,† routing table and ARP table lookups were already hashed). Replacement of timers by timestamps reduces contention, especially for removing expired table entries. If the table entry is allocated from dynamic memory, garbage collection can be carried out infrequently. New features have been designed with inherent support for parallelism; for instance, always using a single function to read or modify a shared datum allows the lock to be defined locally to the function (the *object oriented* approach).

Well Behaved Streams Drivers

There are some rules that well-behaved Streams drivers should follow, which are especially important in a multiprocessor environment. They include not accessing internal Streams structures directly (particularly the *queue*), assuming concurrency of *put* and *service* procedures, and awareness of race conditions in flow control and stream closure; see [Gar90a] for further details. All the TCP/IP modules and drivers conform to these rules.

† The *TCP Control Block* (TCB) holds the state for each TCP connection. TCBs are linked together to demultiplex incoming segments; there is a similar linked list of *UDP Control Blocks*. The advantages of hashing TCB lookups are described in [McK92a].

Put Procedures, *Service* Procedures, and Synchronisation

For purposes of synchronisation and contention, it is assumed that all protocol processing on both input and output may be executed within one *service* procedure. Alternatively, there may be arbitrary rescheduling at each layer. The choice depends on the relative costs of rescheduling messages versus potential advantages thereby for cache affinity scheduling, and is target dependent.

Spin locks provide all synchronisation in the protocol drivers. The use of *sleep* and *wakeup* is avoided in SpiderStreams, and so semaphores are not required. As communication within Streams is message based, data structures are local to a module, and locks need not be held across different modules. Any locks acquired within a *put* or *service* procedure are released before the next *put* procedure (*putnext, qreply, putctl,* and *putctl1*). This simplifies the design, bounds the time for which a spin lock can be held, and avoids deadlocks through attempts to reacquire the same lock. This deadlock is particularly dangerous in Streams because later *qreply* calls may reenter the driver (see [Kle92a]). Locks may be held across *putq* calls, as these simply enable the *service* procedure for scheduling. Following the above precepts ensures that critical sections are small and bounded, so that spin locks are adequate.

The TCP driver contains the longest critical sections, during segment transmission and reception. Connection state is altered during these critical sections. There is contention between two outgoing threads (the Stream head write *put* procedure, and the TCP upper write *service* procedure), some incoming threads limited to the number of segments in a TCP window, and any threads gathering statistics. Critical sections accessing global shared data are shorter, although contention is increased: an outgoing thread for each transport connection, incoming packets and threads gathering statistics and configuring tables. The number of such threads can reach the number of processors in a system under heavy load.

Locking Strategy

Coarse grained locks were used unless performance requirements warranted otherwise. Much global data is concurrently read by many threads but rarely modified. Contention for these was reduced by *readers/writer* locks, which allow concurrent reads but serialise writes. **Circular Wait** deadlock is prevented by enforcing a strict lock hierarchy. As all locks are local to a particular protocol driver, only a small number can be held concurrently, and it is unnecessary to use a more complex scheme such as conditional locking [Pac91a]. Another danger is deadlock through an interrupt service routine (ISR) attempting to acquire a lock held by the interrupted processor. To avoid this, lock acquisition for any datum that may be locked within an network ISR must disable network interrupts. Interrupt related issues are discussed in [Kel89a] and [Eyk92a].

Locking Implementation

Synchronisation around critical sections is coded using *cpp(1)* macros, and disappears when compiling for a uniprocessor. There are two levels of macros. A separate set of macros is defined for each shared datum; this level allows the locking scheme for that datum to be easily modified, unless the scope of the lock is being widened or narrowed.

These macros are defined as system-wide macros implementing mutual exclusion, readers/writer locks and other simple schemes such as incrementing a shared counter.[†] This second level permits efficient tuning of the synchronisation primitives for each target. Macros also permit statements to be embedded for debugging and, redefined as functions, allow profiling for performance analysis.

5. Protocol Driver Implementation

To illustrate issues in adding parallel support to the communications drivers, examples are taken from the TCP and IP drivers. Fine grained parallelism was also added to the NetBIOS, telnet, rlogin, UDP, loopback, ARP, SNAP, LLC1 and Ethernet drivers.

5.1. The TCP Driver

The TCP Driver is an upper multiplexor, where each TCP connection has an entry in a *device array* indexed by minor device number, and a **Transport Control Block** (TCB), allocated from Streams dynamic memory, holding state information for the connection. TCBs bound to an [address, port] pair are added to the **TCB chain**. This is a linked list of TCBs, and is used for demultiplexing incoming segments, traversed to carry out retransmission and other timer-related events, and to collect statistics. A hash table speeds TCB lookup for incoming segments. Figure 2 shows the relationships between these structures.

There are two main requirements for mutual exclusion in the TCP driver;[‡] protecting access to the TCB holding the state for each TCP connection, and to the TCB chain.

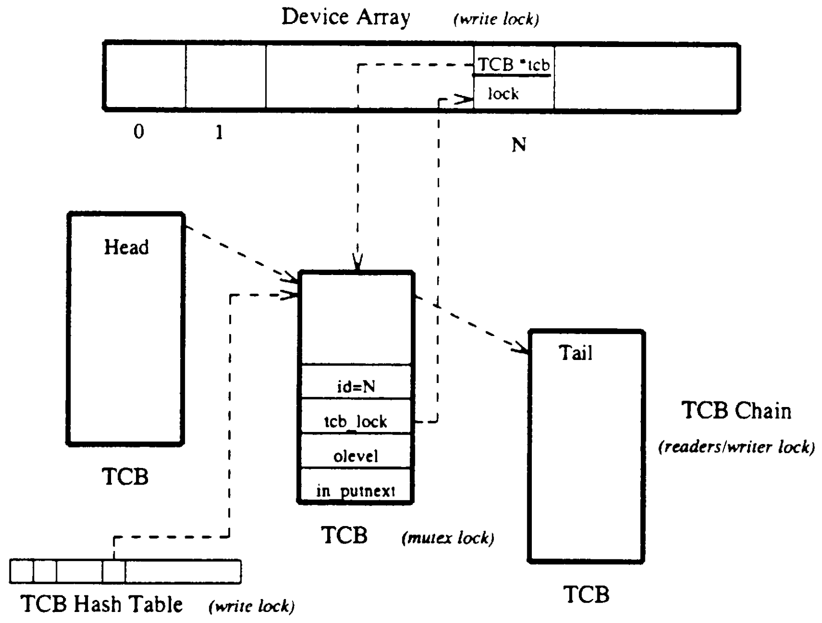


Figure 2: *The TCB Chain, TCBs, device entry table, and TCB hash table*

[†] On some architectures, this operation can be made atomic without recourse to a spin lock.

[‡] The design is similar to that briefly described in [Boy89a].

The TCB Lock

The TCB requires locking, as both write and read side threads modify connection state. The latter includes TCP sequence numbers, offered window size, and data buffered internally because of flow control or for potential retransmission. A mutex lock protects almost the entire TCB. Some fields, such as local and foreign addresses and the TCP state, are not protected for reading. Threads that scan the TCB chain do not need to acquire and release the lock for each TCB. Instead, when the desired TCB is found and locked, it is verified that the state is unchanged; if it has altered, the search is repeated.

Most of the incoming segment processing occurs before the TCB lock is acquired (lower layer processing, segment validation, which constitutes most of the processing for data segments,† and demultiplexing), so a mutex lock permits some concurrency for processing outgoing and incoming segments on a particular connection. Increased throughput for a single connection is not the primary goal, so finer granularity would introduce unnecessary complexity for little gain.

As TCP segments outgoing data, it is possible to make several *putnext* calls within one thread. The TCB lock must be released before and reacquired after the *putnext* call to adhere to the principle of never holding locks across a *put* procedure. However, this opens a window for incoming segments to acquire the TCB lock, so that it must be released with the TCB in a state consistent with incoming segment reception. This issue was sidestepped for paths with minimal influence on performance (connection establishment, connection termination, and delayed ACK generation) by calling *putq* to queue the message for later handling by the TCP lower write *service* procedure. Threads transmitting data and ACK segments, being performance critical, call *putnext* immediately, with the TCB in a consistent state. A similar approach was used when sending data and closing indications up to the application. The old interrupt level is always stored as a field (*olevel*) in the TCB on acquisition of the TCB lock, ensuring correct interrupt level restoration when the lock is released before *putnext*.

Message Reordering

Inherent parallelism carries the potential for message reordering on both input and output. This can happen because of flow control, lock schemes that do not queue requests, and ISRs.

Flow control can reorder messages because of the concurrency of *put* and *service* procedures. Spinning threads acquire the TCB lock in random order, unless the mutual exclusion primitives queue requests. The write *put* and *service* procedures are the only threads that acquire the TCB lock on output, and there can only be one of each for a particular TCB. These problems reduce to that of ensuring that all pending messages for the *service* procedure are processed before a subsequent *put* procedure. A flag in the TCB enforces this.

Neither of the above problems occur on input. Flow controlled data for upstream transmission is queued internally in TCP, rather than on the service queue. The TCP protocol reorders out-of-sequence segments, so incoming threads may acquire the TCB lock in any order. Reordering is a possibility when TCP input processing may be carried out within an ISR. An incoming thread, not called from an ISR (for exam-

† Unless the checksum calculation is implemented in hardware.

ple, as a result of upstream flow control), acquires the TCB lock, packages pending data into a Streams message, releases the lock and calls *putnext* to pass the message up. An ISR is scheduled on the same processor between the lock release and *putnext* call. The ISR repeats the above operation, calls *putnext* and exits. *Putnext* is then called by the interrupted thread, and the two messages have been reordered. Note that this is erroneous behaviour if the second thread is attempting to send up either a data segment or a close indication. To detect this, a flag (*in_putnext*) is set across the *putnext*; the second thread checks this, and if set, queues the data internally in the TCB and enables the *service* procedure for later handling. The flag ensures that only one thread is allowed to execute upward *putnext* calls at a time. Figure 3 shows a code fragment to implement this. *P_LOCK* and *V_LOCK* are the system wide macros for mutual exclusion, which take the address of the lock and the interrupt level at which it is held (the second level macros are omitted).

The TCB Chain

The TCB chain is protected by a readers/writer lock. Hashing reduces contention when demultiplexing incoming data segments. However, the entire chain may be traversed during connection establishment, statistics gathering, retransmission, and binding to a unique address, so a mutex lock is inappropriate. Modification is frequent, when opening and closing connections, so the lock is implemented to give writers priority.

The ordering of the individual TCB locks and TCB chain lock presents a problem. For operations such as retransmission, it is natural to acquire the TCB chain lock first, and then the TCB lock; for other operations, such as opening and closing connections, the reverse is true. The first case is the obvious ordering; otherwise, the TCB chain lock must be released and reacquired to enforce the lock hierarchy, creating a window during which the current TCB may be removed from the TCB chain. To circumvent this, the TCB chain lock is acquired before the TCB lock when a connection is opened, well before it is needed in the thread. The alternative, unwinding the locks (acquiring the TCB lock, releasing it, acquiring the TCB chain lock and then reacquiring the TCB

```
TCB *tcb;         /* TCP Control Block: tcb_lock points to mutex,
                     olevel holds interrupt level */
mblk_t *mp;       /* message block containing data in TCP segments */

#define PASS_DATA_UP(mp, tcb) {          /* macro to ensure no reordering */
        tcb->in_putnext = 1;             /* set flag */
        V_LOCK(tcb->tcb_lock, tcb->olevel);     /* unlock TCB, using TCB field as
                                                   pointer to lock */
        putnext(tcb->tcb_qptr, mp);      /* tcb_qptr is read queue:put message up */
        tcb->olevel = P_LOCK(tcb->tcb_lock, SPLSTR);    /* lock TCB */
        tcb->in_putnext = 0;                            /* clear flag */
}

                                         /* check on input */
if (tcb->in_putnext) {                   /* another thread is putnext'ing a message up */
        queue_data_internally(mp, tcb); /* queue message internally */
        qenable(tcb->tcb_qptr);     /* enable read service procedure, held in tcb_qptr */
} else
        PASS_DATA_UP(mp, tcb);           /* send message up */
```

Figure 3: *Avoiding data reordering on input*

lock), is dangerous for a networking driver, as an incoming segment may acquire the TCB lock during the window.

Contention for the TCB chain when closing connections was reduced in the following manner. Regular events in TCP are handled by a timer, which enables a *service* procedure used for retransmission, keep-alive generation and other purposes. This ensures that critical sections do not execute at timer interrupt level. Many TCP connections are closed in this *service* procedure, on expiry of the TIME_WAIT timer. It is straightforward to require that all TCBs be removed in the same way; those that should be removed immediately merely wait for the next invocation of the retransmission *service* procedure, meanwhile marking themselves as expired. This is the only thread that acquires the TCB chain lock for writing during TCB removal,almost halving the number of threads contending for this lock. Note that this *service* procedure must acquire the TCB chain lock for reading anyway.

Vanishing TCBs

The TCB is held in a Streams message block, dynamically allocated when the connection is opened. When the *service* procedure described in the previous section removes the TCB, it acquires the TCB lock, frees the dynamic storage used to hold the TCB, and releases the lock. Several events may occur while this in progress; a *timeout* routine, *bufcall* callback, or the demultiplexing of an incoming segment onto this TCB. Any of these spin on the TCB lock; on acquisition, the TCB has disappeared.† SVR4/MP Streams provides a *disable_procs* call that does not return until all messages queued on the read and write side of the stream have been processed, but it does not deal with these three issues.

It is insufficient to cancel the *timeout* or *bufcall*, as they may fire between the acquisition of the TCB lock by the closing thread, and calling *untimeout* or *unbufcall* (in which case they spin on the TCB lock until the closing thread releases it). To avoid this, a consistency check is required. At this point, the contents of the TCB are unreliable, so that it is important that neither the TCB lock nor the entry point used to find the TCB should be held in dynamic memory.

The following solution was implemented. The TCB lock and a sequence number are stored in the device array, along with a pointer to the TCB. The index into the array is logically *ORed* with the current sequence number,‡ and this is passed to the *timeout* routine or *bufcall* callback. Before TCB deletion the sequence number is incremented while holding the TCB lock. A consistency check (using a field in the TCB, *id*, that holds the index into the device array for that TCB), shows that the TCB has expired, and the routine returns without corrupting a freed or reused message block. Figure 4 illustrates the problem by showing data structures and a code fragment based on the timer that implements delayed ACKs.

As incoming segments can only access the device array through the index held in the TCB (without scanning the entire array), the sequence number check cannot be used. Instead, a consistency check is carried out on the addresses used to demultiplex, which are zeroed during closedown. If these have changed once the TCB lock has been acquired, the segment is dropped. The TCP protocol prevents immedi-

† SunOS 5.0 extends the Streams architecture to solve this problem, which is avoided by calling the *enterq*, *leaveq*, *entrstr* and *leavestr* routines [Sun92a].

‡ The combined array and sequence number spaces cannot exceed the size of the parameter passed to *timeout* or *bufcall*, i.e. the word size.

```
struct tcbarray {       /* device array, one for each open TCP connection */
        TCB *tcb;               /* pointer to TCP Control Block */
        lock_t lock;            /* mutex lock for TCB */
        unsigned char seq;      /* 8-bit sequence number */
} tcentry[NTCPCON];             /* NTCPCON < 2**(sizeof(int)-sizeof(unsigned char)) */

#define DEVICE_ARRAY_SPACE      24              /* 24 bits for device array */
#define DEVICE_MASK             0x00ffffff      /* mask off top 8 bits */

#define DEV_OR_SEQ(tcb) ((tcentry[tcb->dev].seq << DEVICE_ARRAY_SPACE) | tcb->dev)
#define SEQ(seqdev)             (seqdev >> DEVICE_ARRAY_SPACE)  /* current sequence
                                                                   number */
#define DEV(seqdev)             (seqdev & DEVICE_MASK)          /* used to find TCB */

tcb->tickid = timeout(acktick, DEV_OR_SEQ(tcb), HZ/5);          /* delayed ACK timer,
                                                                   200 ms */
void acktick(seqdev)            /* ACK timer fires */
caddr_t seqdev;
{
        int olevel, dev = DEV((int)seqdev);     /* calculate index into tcentry array */
        TCB *tcb = tcentry[dev].tcb;            /* calculate pointer to TCB */

        olevel = P_LOCK(&tcentry[dev].lock, SPLSTR);    /* acquire TCB lock,
                                                           using global address,
                                                           as TCB may have gone */

        if (SEQ(seqdev) == tcentry[DEV(seqdev)].seq) {  /* check sequence numbers */
                /* ....usual processing.... */
        } /* else: do nothing, connection has vanished */

        V_LOCK(&tcentry[dev].lock, olevel); /* release TCB lock */
}
```

Figure 4: *Detecting Vanishing TCBs*

ate reuse of addresses, so there is no possibility of a new connection with the same addresses immediately reusing the message and masquerading as the same connection. However, it is unsatisfactory to carry out a consistency check on a freed message. The solution is to extend the Streams multiplexor architecture as described in [Sun92a].

Other Areas

The device entry and hash lookup tables, packet identifier and flow control status flag are protected by write locks. A mutex lock on the TCB chain is required when scanning it to check unique use of [address, port] pairs. SNMP and other statistics updates require synchronisation. There is a potential race condition between *timeout* and *untimeout* being called on the same identifier, resulting in the timer continuing to fire. This is avoided by a mutex lock.

5.2. The IP Driver

Introduction

The IP driver is both an upper and lower multiplexor. A protocol table multiplexes incoming datagrams up to the TCP driver, UDP driver or *ping* utilities.† A network table multiplexes outgoing datagrams onto network interfaces. There is a routing table for outgoing route determination, and a double chain of fragment chains for receiving fragmented

† ICMP is also implemented within the IP driver.

datagrams. Incoming datagrams contend for the network table, the protocol table and, if fragmented, the fragment chains. Outgoing datagrams, statistics gathering routines, and SNMP management requests contend for the network table and the routing table. Figure 5 shows the data structures.

Upper and Lower Multiplexing Tables

The upper multiplexing table is protected by a mutex lock. Although this uses a linear search, and there is contention for this by all incoming threads, the critical section is short, as most threads are TCP segments or UDP datagrams, which are matched in the first two entries; later entries are used for demultiplexing ICMP Echo Replies to the appropriate *ping* application, which is rarely used.

The lower multiplexing table is protected by a readers/writer lock. SNMP commands to disable or enable interfaces can alter entries. In addition, the table is filled when the protocol stack is linked together at startup, and cleared when it is dismantled. The entries are rarely modified, so the lock can be implemented to give readers priority.

Routing Table

The routing table is implemented as an array of hash buckets, and protects the entire array by a readers/writer lock. As change occurs only during routing instability, this scheme suffers less contention than a mutex lock per hash bucket. Many outgoing IP datagrams routed to the same IP network require access to the same routing table entry, and would contend for a mutex lock on the entry.

Fragmentation and Other Areas

Locks must be released before *putnext* of fragmented datagrams, as in the TCP driver. Rather than risk changes to the network table between fragment transmissions, lock release and reacquisition is avoided. All the fragments of an outgoing datagram are constructed and queued

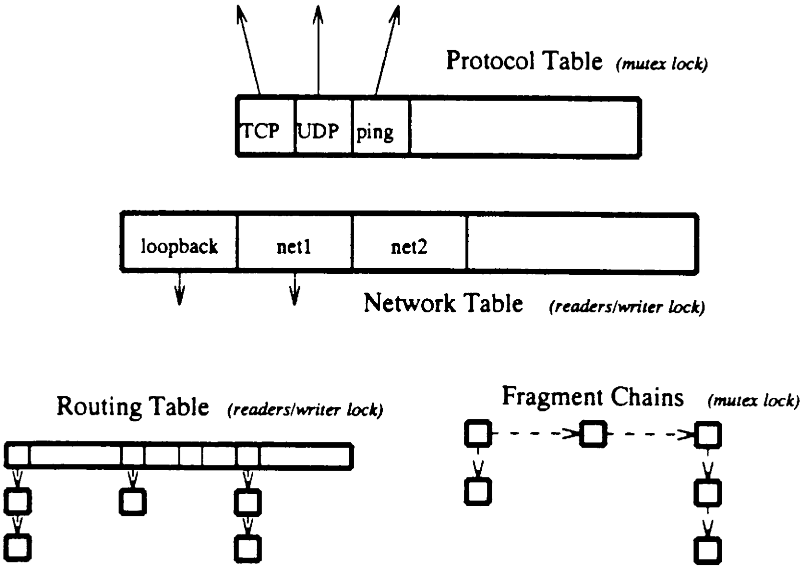

Protocol Table (mutex lock)
TCP
UDP
ping
loopback
net1
net2
Network Table (readers/writer lock)
Routing Table (readers/writer lock)
Fragment Chains (mutex lock)

Figure 5: *IP Multiplexing Tables, Routing Table, and Fragment Chain*

internally, the locks are released, and *putnext* is called for each queued fragment. The chain of received fragment queues is protected by a mutex lock. Fragmented datagrams are rarely received, and carry a performance penalty themselves, so that contention may be acceptable.[†] Other locks include write locks for the IP device entry table and ICMP packet identifiers.

5.3. Performance Measurement and Analysis

Performance Tuning

It is important to tune the synchronisation primitives (mutex locks, readers/writer, and atomic increments) for the target architecture. The implementation of these mechanisms varies depending on the coordination primitives that are available on a particular architecture [Gle91a, Mel91a, Fre91a, Bae91a]. For example, implementations of readers and writers that utilise *compare&swap*, *load-linked/store-conditional*, and *fetch&add* offer higher performance than those based on mutual exclusion primitives such as *test&set* [Got81a, Joh92a, Mel91b, Hsi91a].

The Intel 80486 is a popular architecture for shared bus multiprocessors. The development environment, *Windows/NT* and SVR4/MP all run on 80486-based multiprocessors. The 80486 *xadd* instruction, when combined with a bus lock, provides an atomic *fetch&add*. This allows us to implement *fetch&add* based optimistic writer priority algorithms for readers-writers coordination, supplied by Eric Freudenthal of the research staff at the NYU Ultracomputer laboratory.[‡] Appendix C shows the C source for the algorithms. In the absence of writers, this algorithm requires a single *fetch&add* for reader lock acquisition, which is no more expensive than a mutex lock. Mutex locks can be implemented using *test&set* with exponential backoff, which yielded good results in [Mel91a]. Shared counter increments avoid the penalty of a mutex lock by prefixing the increment with the 80486 bus lock instruction. Appendix C includes the inline assembler source for this on the Sequent Symmetry. It would be fruitful to investigate queuing locks and other synchronisation primitives.

Performance Measurement

Scalability can be measured by using the TCP/IP performance benchmark program *ttcp*. This measures memory-memory transfer rates for a variety of buffering strategies. Parallel benchmarks may be measured by increasing the number of connections simultaneously running *ttcp*. The time taken to complete one transfer indicates scalability, as long as the other transfers do not complete beforehand.

Currently, the only platform available to us for measurement is the development environment running on the Symmetry. This architecture is optimised for ease of development, not performance. The Streams server and all applications run as user processes. Packets to and from the network incur large IPC and context switch overheads. The architecture could be improved by multithreading data source and sink applications, and the Streams server, as the same user process. Loopback transfer would then incur no IPC overhead. The server is suited

[†] In some configurations, NFS iragments UDP datagrams. The lock granularity may be reviewed in the light of this behaviour.

[‡] The algorithms will be included in a forthcoming technical report by the NYU Ultracomputer Research laboratory.

for a real time operating system or microkernel architecture, and it would be interesting to measure performance in such an environment. Performance in *Windows/NT* and SVR4/MP will be measured in the future.

Performance Analysis

A tool has been developed to collect lock contention statistics (others include [Cam91a] and [Pea92a]). The Streams trace logger is used to collect output for later analysis. Figure 6 shows the statistics collected. In addition, a configuration structure allows fine control over collection of statistics, which may be turned on or off while the system is running, or by triggering certain thresholds.

Profiling of the message server process using *gprof*[Gra82a] can provide useful information about bottlenecks. Redefining the locking macros as functions provides an indication of contention for those locks. A specialised parallel program performance tool like IPS-2 [Mil90a] would be more flexible, as pointed out in [Hol92a]. The combination of more sophisticated profiling techniques and the lock spin statistics package should allow fine grain monitoring and tuning of the parallel protocol stack.

6. Conclusions

In this paper we have investigated some aspects of the implementation of TCP/IP in Parallel Streams. These include the design of the TCP and IP drivers, the development environment, implementation on an Intel 80486 architecture, and techniques for performance analysis and measurement.

The lower layers, and important areas such as statistics, have not been described. Scalability measurement and analysis of overheads on various platforms remains to be done. Traffic analysis for WANs and LANs could be used to model realistic workloads for both small and large numbers of processors.

The effect of varying parameters such as lock implementations and scheduling policies could also be measured.

```
struct cr_lock {        /* structure holding the lock, debugging and statistics */
        lock_t  slock;  /* the lock itself */
        char    *name;  /* debugging information */
        char    *file;  /* debugging information */
        int     line;   /* debugging information */
        int     held;   /* debugging information */
#ifdef PERFORMANCE_TRACE
        int report;     /* sequence number indicating when to report */
        int reset;      /* sequence number indicating when to reset */
        unsigned long assertions;       /* number of times lock is acquired */
        unsigned long assert_no_spins;  /* times lock acquired w/o spinning */
        unsigned long total_spins;      /* total number of spins */
        unsigned long max_spins;        /* maximum number of spins */
        char *maxspins_file;            /* file where max spins occurred */
        int maxspins_line;              /* line where max spins occurred */
        unsigned long min_spins;        /* minimum number of spins */
#endif
};
```

Figure 6: *Spin Lock Statistics and Debugging Information*

7. Acknowledgements

Nick Felisiak of Spider Software was the architect of our implementation of Parallel Streams and the development environments described in Sections 2 and 3. Gavin Shearer, Richard Edmonstone, Ben Durruti, Ian Steele, and Siangjie Jin worked on the parallel TCP/IP project. Nick Oakins implemented the 80486-based readers/writer locks. We are grateful to the NYU Ultracomputer laboratory for providing us with the algorithms.

Thanks to Edinburgh University Computing Service for their cooperation and use of their machines. We've enjoyed working with Mike Massa, Henry Sanders, Dave Thompson and the others at Microsoft.

Many thanks to my colleagues and others who read various drafts of the paper; in particular, to Eric Freudenthal for his ideas. Lastly, thanks to Stuart McRobert for his patience.

Appendix A: Parallel Streams Terminology

Parallel Streams is discussed in [Gar90a, Pea92a, Cam91b, Kle92a]. A Streams-based protocol stack is composed of *modules* and *drivers*, which are used interchangeably in this paper except when specifically referring to a *multiplexing driver*. Entry points to modules or drivers are *queues*, and all intermodule communication is via *messages* placed on the queues. A *put procedure* is a function call to another Streams module, passing the queue and the message as parameters. A *service procedure* is a thread of execution within Parallel Streams, which has its own non-preemptive FIFO multithreaded scheduler. When the Streams scheduler runs, the *service procedure* for each enabled queue is executed; it calls (possibly zero) *put* procedures, until the message is queued for another *service* procedure, when it returns, terminating the thread of execution. This means that *put* procedures on different queues also execute in parallel.

Appendix B: Overview of the SpiderStreams Architecture

SpiderStreams provides an environment in which modules and drivers written to run in a UNIX V.3 or V.4 kernel may be used on other systems not supporting Streams. It consists of a library of functions which provide a UNIX-like interface, and a server process in which the Streams "kernel" code runs. SpiderStreams uses a message passing mechanism between applications and the server process in which the Streams drivers execute. This is very suitable for microkernel-based operating systems like Mach or QNX. For operating systems with a UNIX-like system call interface, the server may be converted to a procedural interface. This model is strongly process-oriented, and assumes that a process will handle a limited number of streams, and that concurrent activity on those streams will not be the normal mode of operation.

SpiderStreams may also provide a bare protocol stack on a front end card. Note that Streams itself does not require a multitasking environment; the Streams scheduler is single threaded and *service procedures* (Streams threads) do not have a context. A *Cross Bus Driver* has been implemented to extend the Streams interface across a bus, for PC (ISA), Multibus II and VME architectures, with SpiderStreams running on multiple front end cards, possibly with Streams also on the host, allowing Streams protocols to run in both environments.

Streams requires basic services from the underlying operating system; store allocation, a source of real time, and an interface to the I/O hardware. If applications are required, there must be some kind of interprocess communication. The Streams server communicates with applications via a synchronous message passing interface, which can be implemented as a library based on IPC facilities; this has been ported to UNIX System V message queues and shared memory, QNX RPC primitives, pSOS message queues and Vrtx message queues. In particular, SpiderStreams has been implemented as a UNIX user process. This forms a powerful development environment; the kernel is stable, many instances of SpiderStreams may be running on one machine, source code debuggers are available, and the *edit-build-debug* cycle is very short, deriving all the advantages of user level versus kernel programming.

Appendix C: Lock Implementations on the Intel 80486

```
/*
 * inline assembler macro for atomic counter increment
 */
asm void INCREMENT(laddr)
{
%reg laddr;
/PEEPOFF
        incl laddr
/PEEPON
%mem laddr;
/PEEPOFF
        lock
        incl laddr
/PEEPON
}
/*
 * NYU Ultracomputer Laboratory readers/writer algorithms provided by Eric Freudenthal
 * readerPrologue, readerEpilogue, writerPrologue, writerEpilogue
 */

/* C versions  - faa() is an atomic fetch&add */

#define BIG 65536

readerPrologue(lock) {
        t = faa(lock, BIG);             /* request lock */
        while (t % BIG) {               /* is there a writer? */
                t = faa(lock, -BIG);    /* release request */
                while (t % BIG)         /* wait till other writer drains... */
                        t = lock;
                t = faa(lock, BIG);     /* request lock */
        }
}

readerEpilogue(lock) {
        faa(lock, -BIG);
}
```

```
writerPrologue(lock) {
top:     t = faa(lock, 1);               /* request lock, lock out readers */
         if (t % BIG != 0) {             /* is there another writer? */
                 t = faa(lock, -1) - 1;  /* undo request */
                 while (t % BIG != 0)    /* wait for other writers */
                         t = *lock;
                 goto top;
         }
         while (t / BIG)                 /* wait for readers to drain */
                 t = *lock;
}

writerEpilogue(lock) {
         faa(lock, -1);
}
```

References

[And91a] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska, "The Interaction of Architecture and Operating System Design," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, University of Washington, Santa Clara, California (April 1991).

[AT&86a] AT&T, *UNIX System V Release 3 STREAMS Primer*, 1986.

[AT&89a] AT&T, *UNIX System V Release 4.0 Device Driver Interface/Driver-Kernel Interface (DDI/DKI) Reference Manual*, 1989.

[Bae91a] Jean-Loup Baer and Richard N. Zucker, "On Synchronization Patterns in Parallel Programs," in *Technical Report No. 91-04-01*, Department of Computer Science, University of Washington (April 1991).

[BBN90a] BBN, *Inside the TC2000™ Computer,* BBN Advanced Computers, Inc. (February 1990).

[Boy89a] Joseph Boykin and Alan Langerman, "The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis," in *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS I)*, Encore Computer Corporation (Oct 1989).

[Cam91a] Mark Campbell, Russ Holt, and John Slice, "Lock Granularity Tuning Mechanisms in SVR4/MP," in *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II)*, NCR Corporation (1991).

[Cam91b] Mark Campbell, Richard Barton, Jim Browning, Dennis Cervenka, Ben Curry, Todd Davis, Tracy Edmonds, Russ Holt, John Slice, Tucker Smith, and Rich Wescott, "The Parallelization of UNIX System V Release 4.0," in *USENIX*, NCR Corporation, Dallas, Texas (Winter 1991).

[Cla91a] David D. Clark and David. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *SIGCOMM 91*, Laboratory for Computer Science, MIT, Zurich, Switzerland (September 1991).

[Cus92a] Helen Custer, "Inside Windows NT™," in *ISBN 1-55615-481-X* (1992).

[Dov90a] Ken Dove, "A High Capacity TCP/IP in Parallel Streams," in *UKUUG*, Sequent Computer Systems (Summer 1990).

[Eyk92a] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, D. Stein, M. Smith, A. Shivalingiah, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing: Multithreading the SunOS Kernel," in *USENIX Summer 1992*, SunSoft, Inc., San Antonio, Texas (June 1992).

[Fre91a] Eric Freudenthal and Allan Gottlieb, "Process Coordination with Fetch-and-Increment," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, UltraComputer Research Laboratory, New York University, Santa Clara, California (April 1991).

[FSF87a] FSF, *GDB Manual: The GNU source-level Debugger,* Free Software Foundation (1987).

[Gar90a] Arun Garg, "Parallel STREAMS: a Multi-Processor Implementation," in *USENIX*, Sequent Computer Systems, Washington, DC (Winter 1990).

[Gle91a] Andy Glew and Wen-mei Hwu, *A Feature Taxonomy and Survey of Synchronization Primitive Implementations,* Center for Reliable and High-Performance Computing, University of Illinois (February 1991).

[Got81a] Allan Gottlieb, B.D. Lubachevsky, and Larry Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," in *Ultracomputer Note 16*, New York University (December 1981).

[Got83a] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Lawrence Rudolph, and Marc Snir, "The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer," in *IEEE Trans. Comp.* (February 1983).

[Gra82a] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A Call Graph Execution Profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (June 1982).

[Gui90a] Roger Peel, Department of Electronic and Electrical Engineering, University of Surrey, Guildford, U.K., *TCP/IP Networking using Transputers,* Transputer Research and Applications 3, Amsterdam (1990).

[Hil92a] Dan Hildebrand, "An Architectural Overview of QNX," in *Proceedings of the USENIX Workshop on Microkernels and other Architectures*, Quantum, Seattle (April 1992).

[Hol92a] Jeffrey K. Hollingsworth and Barton P. Miller, "Parallel Program Performance Metrics: A Comparison and Validation," in *Supercomputing '92*, Computer Sciences Department, University of Wisconsin (1992).

[Hsi91a] Wilson C. Hsieh and William E. Weihl, *Scalable Reader-Writer Locks for Parallel Systems,* Massachusetts Institute of Technology (November 1991).

[Jai90a] Niraj Jain, Mischa Schwartz, and Theodore R. Bashkow, "Transport Protocol Processing at GBPS Rates," in *ACM SIGCOMM 90*, Columbia University, New York, Philadelphia, Pennsylvania (September 1990).

[Jen88a] M. N. Jensen and M. Skov, "Multi-Processor Based High-Speed Communication Systems," in *6th European Fibre Optic Communications and Local Area Networks Exposition*, Technical University of Denmark, Lyngby, Amsterdam, Netherlands (June-July 1988).

[Joh92a] Ted Johnson, "Approximate Analysis of Reader/Writer Queues," in *IEEE Transactions on Software Engineering*, Dept. of CIS, University of Florida (In Submission 1992).

[Kan88a] Hemant Kanakia and David R. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors," in *ACM SIGCOMM 88*, Computer Systems Laboratory, Stanford University, Stanford, California (August 1988).

[Kel89a] Michael Kelley, "Multiprocessor aspects of the DG/UX kernel," in *USENIX Winter 1989*, Data General Corporation, San Diego, California (February 1989).

[Kle92a] S. Kleiman, "Symmetric Multiprocessing in Solaris 2.0," in *COMPCON*, San Francisco, California (Spring 1992).

[Kol89a] Frank Kolnick, *The QNX Operating System,* Basis Computer Systems (1989).

[Len92a] Daniel Lenoski, James Landon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam, "The Stanford Dash Multiprocessor," in *IEEE Computer*, Stanford University (March 1992).

[Mar92a] Evangelos P. Markatos and Thomas J. LeBlanc, "Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance," in *Technical Report 420*, University of Rochester, NY (May 1992).

[McK92a] Paul E. McKenney and Ken F. Dove, "Efficient Demultiplexing of Incoming TCP Packets," in *SIGCOMM 92*, Sequent Computer Systems, Baltimore, Maryland (August 1992).

[Mel91a] John M. Mellor-Crummy and Michael L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Transactions on Computer Systems, 9(1):21-65*, Centre for Research on Parallel Computation, Rice University (1991).

[Mel91b] John M. Mellor-Crummy and Michael L. Scott, "Synchronization Without Contention," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Centre for Research on Parallel Computation, Rice University, Santa Clara, California (April 1991).

[Mil90a] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," in *IEEE Transactions on Parallel and Distributed Systems* (April 1990).

[Nuc91a] Neal Nuckolls, *Multithreading your STREAMS Device Driver in SunOS 5.0,* Internet Engineering, Sun Microsystems (December 1991).

[Pac91a] Noemi Paciorek, Susan Loverso, and Alan Langerman, "Debugging Multiprocessor Operating Systems Kernels," in *Symposium on Experiences with Distributed and Multi-*

processor Systems (SEDMS II), Encore Computer Corporation (1991).

[Pea92a] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang, and Wilfred Yu, "Experiences from Multithreading System V Release 4," in *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, Intel Multiprocessor Consortium, Newport Beach, California (March 1992).

[Pik92a] Rob Pike, "Implementation of Coroutines for Plan9," in *Message-ID on comp.os.research* (July 8, 1992).

[Pre90a] David Leo Presotto, "Multiprocessor Streams for Plan9," in *UKUUG Summer Conference* (Summer 1990).

[Rea88a] Ready Systems, *VTRX32/86 User's Guide*, 1988.

[Rit84a] D. M. Ritchie, "A Stream Input-Output System," *AT&T Bell Labs Technical Journal* **63**(8) (October 1984).

[Rud90a] Harry Rudin and Van Jacobson, "Protocols for High-Speed Networks," in *ACM SIGCOMM 90 Symposium*, Philadelphia PA (September 1990).

[SCG90a] SCG, *pSOS+/68K User's Manual,* Software Components Group (1990).

[Seq87a] Sequent, *Symmetry Technical Summary, 1003-47396-00 Rev. A,* Sequent Computer Systems (1987).

[Sun92a] SunSoft, *SunOS 5.0 STREAMS Programmer's Guide, Chapter 13, Multi-Threaded STREAMS*, 1992.

[The92a] C. Thekkath, D. Eager, E. Lazowska, and H. Levy, "A Performance Analysis of Network I/O in Shared Memory Multiprocessors," in *Dept. of Computer Science and Engineering FR-35*, University of Washington, Seattle, WA (July 1992).

[Zit89a] Martina Zitterbat, "High Speed Protocol Implementations Based On A Multiprocessor Architecture," in *Proceedings of the IFIP WG 6.1/WG 6.4 International Workshop on Protocols for High Speed Networks*, University of Karlsruhe, Zurich, Switzerland (May 1989).

A Shared Memory Programming Environment for Distributed Systems

Umesh Bellur
Sumana Srinivasan

Syracuse University
Syracuse, NY, USA
{ bumesh | sumana }@cat.syr.edu

Abstract

Although there has been a significant increase in the use of distributed systems, we are yet to see them being commonly employed to solve a problem concurrently. One of the obstacles to programming distributed systems is the absence of the tight coupling that characterizes a parallel processing MIMD machine. Programming with distributed memory can be extremely difficult, owing to the fact that the programmer needs to deal with the complexities of communication across machine boundaries and that, this is the only form of communication possible. Problems with communication include network protocols, asynchrony inherent in such an environment and the incompatibilities of various architectures that go to form any heterogeneous distributed platform. What is therefore required is, an environment that can present to the developer an illusion of shared memory atop a platform of distributed memory machines. This is distributed shared memory, an idea which makes possible shared memory parallel programming on distributed networks. This paper discusses the design and implementation of such a system under development at Syracuse University.

1. Background

Networks of powerful workstations are becoming increasingly common because of the enormous computing power they offer at low cost. Recent advances in workstation technology coupled with the availability of high speed networks (FDDI) has not only made these architectures viable alternatives to massively parallel ones, but their low cost has made them attractive as well. Some of the other reasons that have attributed to the rising popularity of distributed systems include high availability (fault tolerance) and the scalability afforded by these architectures.

Despite all their advantages we are yet to witness the collective use of these workstations as a single parallel machine to solve computationally intensive problems, since they lack the ease of programmability that shared memory multiprocessors have to offer. To program the required communication and synchronization into a distributed application is no easy task, primarily due to the reasons of complexities and cost of network communication and the asynchrony of events in distributed systems. The availability of a shared memory abstraction on top of such an architecture is therefore likely to provide the necessary incentive to users to utilize networks to their fullest extent possible. We present here, an effort to design and implement such an abstraction on a platform consisting of a heterogeneous network of workstations.

The rest of this paper is organized as follows. We first describe the goals of this project. This is followed by the motivation for such an effort and a discussion of issues important to the design of such a system. We then describe in detail, the design and implementation of our system.

1.1. Motivation

There exist two basic models of communication for co-operating processes – the message passing model and the shared memory model. In the latter paradigm processes share a common address space and this tight coupling allows processes to communicate through shared data. Shared memory provides a simple, yet powerful way of structuring systems. But as we slowly approach the physical limits of processor and memory speeds, it is becoming necessary to extend this paradigm to non-shared memory (distributed memory) architectures as well.

However, for a loosely coupled distributed system, there is no physically shared memory to support such a model and communication is commonly achieved by passing messages. Programming using this model is often tedious as it is the responsibility of the programmer to move data back and forth between processes as well as handle the inconsistencies arising out of the different representations of data on different architectures. These reasons make it attractive to have available, an abstraction of shared memory on a loosely coupled distributed system. Therefore the primary advantage of distributed shared memory over the message passing model is the simpler abstraction provided to the application programmer. In spite of structuring such a DSM system over a message passing one it sometimes delivers better performance as compared to the latter because of the following reasons:

- If the application exhibits a sufficient degree of locality, then the overhead of moving data by message passing get's amortized.
- We can exploit concurrency better by spreading out the data exchange phase of the application.
- There is a corresponding decrease in the paging and swapping activity due to the net increase in memory available.

Sharing can be provided either at the level of pages and segments or at the level of user-defined data structures. Most of the systems that we have seen use the *page* or *segment* as the unit of sharing. But from the programmer's point of view sharing is in terms of the data structures that he/she declares and manipulates. We intend to provide sharing directly at this level – the level of data structures (objects). The other important point to note is that support for DSM is built into the operating systems kernel in most of these systems. This implies that porting the DSM portion to a new architecture really means porting the entire operating system kernel. By providing DSM as a separate service on top of the operating system kernel we reduce the effort required to port this to a variety of architectures thereby facilitating portability and heterogeneity.

1.2. Goals

The primary aim of this project is to provide an *easy to use* environment for shared memory parallel programming atop a distributed memory architecture such as a network of workstations. The main goals arising out of this are to:

1. Provide support for sharing at the level of user defined data structures including types that can be defined in the context of an object oriented language such as C++.
2. Provide this abstraction as a software layer on top of the operating system for reasons of portability.
3. Make the environment machine-independent and thereby support heterogeneity.
4. Provide a structured interface to the environment's capabilities.
5. Provide language/compiler support for this abstraction thereby automating the required interaction of the user with our environment.
6. Make available a larger total virtual address space than is individually available on any one machine in the network.
7. Make all of this possible with the lowest possible run time overhead.
8. Serve as a testbed for evaluating the performance of shared memory applications on a network.

2. Related Work

There has been a significant amount of research done on DSM systems over the last decade [Tam90a, Kes89a, Stu90a, Tan90a]. We now discuss some of the systems that were developed and then provide the motivation for our design.

IVY [Li86a] is a coherent, shared virtual memory system on a loosely-coupled multiprocessor – the Appolo Domain System. Shared data is paged between processors, some of which have copies of the virtual address space pages. The model assumes ownership of pages can vary from processor to processor either statically or dynamically. The last writer to a page becomes the new owner. Unless the local processor owns the page a managing site must be inquired before a write can occur.

Clouds [Das88a] is an object oriented distributed operating system which supports DSM through data mobility and replication. Segments [Ram89a] are the units of sharing and are classified upon movement into *none*, *weak-read*, *read-only*, and *read-write*. The *none* mode guarantees exclusive access, but the segment in this mode can be taken away at any time and data consistency is not guaranteed. The *read-only* mode provides that guarantee until the reading process has explicitly unlocked the segment. The *read-write* mode provides exclusive access and a guarantee that the segment will not be taken away till it is explicitly unlocked. The segment can be acquired by a process in a weak-read mode, but data consistency is not guaranteed. Extensive work has been carried out for coherence and synchronization protocols. The main contribution of Clouds is in recognizing that the combination of memory coherence protocol and process synchronization can improve the performance of DSM [Ram89b]

Mach [Acc86a] supports a shared memory server. Memory objects are managed either by the Kernel or by the user programs through a message interface. Sharing of memory is provided between the tasks running on the same machine or across machines. An external memory paging task handles the paging duties and is responsible for the memory object. Mach attempts to deal with multiple page sizes and some aspects of heterogeneity.

An object oriented language and run time environment supporting DSM indirectly through object mobility is **Emerald** [Jul88a]. All movement must be explicitly specified by the developer through movement primitives built into the language.

Munin [Ben90a] is an object-based DSM system that investigates type specific coherence protocols. It does not support heterogeneity and provides synchronized objects for synchronization. It allows for dynamic system decisions like Replication vs. Remote load store. The user can specify either of the two decisions based on the program semantics. Munin requires the programmer to specify all semantic requirements that are required by the run-time system.

Linda [Nit91a] is a shared associative object memory system which can be layered atop many languages and machines. It uses the notion of a globally accessible tuple space into which live data can be cast. This tuple space is essentially shared memory. It's coherence semantics does not allow any mutable data.

Mermaid [Nit91a] is a DSM system for heterogeneous systems, where the compiler forces shared pages to contain only variables of a single type. Type conversion is performed on reference.

Mirage [Fle89a] provides a kernel level implementation of DSM and reduces thrashing by prohibiting pages from being invalidated before a certain minimum amount of time has elapsed.

3. Design Issues

We will take a look at some of the issues that need to be considered in the design of any DSM system.

- **Granularity**: This refers to the unit of sharing in any shared memory system. Most systems previously built consider a line, a page or a segment to be the unit of sharing. The two issues under consideration are the way shared data is laid out in memory and the size of the unit of sharing. Large granularities are useful only if the degree of locality is high. One must however put up with the high cost of transferring a page across the network. Fine granularities may result in a lot of movement and is suited to systems with a lower degree of locality. However, in order that the applications using this model be architecture independent it is necessary that the sharing should be done at program level rather than at kernel level. The Munin system is one such system that follows this approach.
- **Coherence Protocol**: This guarantees the consistency of shared data across machine boundaries. Strict semantics automatically enforce consistency and make sure the most recently written value is returned on a read. Weak semantics on the other hand leave the task of consistency maintenance to the programmer and provide some synchronization that can be used for sequentialization of accesses to shared data. Guaranteeing memory coherence is one of the principal goals of any DSM system. In systems

where shared data is replicated, guaranteeing coherence is not a trivial task. The two main types of protocols that are used in practice to achieve this are the *write-invalidate* and *write-update* protocols. The *write-invalidate* protocol makes sure that all copies of the shared data are invalidated before a write operation. The *write-update* protocol updates all the copies during the write operation.

- **Synchronization**: Any situation that involves multiple processes and shared data amongst these processes needs to have some form of synchronization. This exists mostly in the form of constructs that guarantee sequentiality of accesses. Common synchronization mechanisms include semaphores, monitors etc. Locks are commonly used by parallel applications for achieving synchronization. As these applications are computation intensive, the demand on the lock is usually light and a simple algorithm such as the central server algorithm is sufficient. However if the lock is highly contended for, then a centralized server poses to be a performance bottleneck.
- **Scalability**: One of the advantages of distributed architectures is their ability to be scalable. This is because they are independent units on a network and it is only the bandwidth of the network that constrains the number of machines. Any DSM system atop this network therefore must be capable of scaling with an increase in the capacity of the system. Central control is likely to become a bottleneck with increased capacity and is to be avoided. On the other hand too much decentralization can lead an increased cost of operation.
- **Heterogeneity**: This is one of the most difficult issues to deal with in any DSM system design. It is almost certain that two machines do not use the same representation for even basic data types. So sharing at the level of pages or segments presents a multitude of problems including those of fragmentation, representation conversions and so on. It becomes possible to accommodate heterogeneity easily when the DSM system is structured around the data structures of the language, for then, the compiler and the DSM system can insert the necessary conversion code on transfer.

4. Requirements of a DSM Model

A typical DSM model should be able to support the features mentioned in the previous section. Some of the requirements of such a DSM model are:

1. A single large virtual address space, distributed over many machines with overall memory coherence similar to that provided by cache coherence protocols in a multiprocessor environment.
2. All data movement necessary to achieve coherence.
3. A way to ensure partial ordering of events for synchronization purposes.
4. Semaphore-like mechanisms for locking shared address spaces which may be required by the application to ensure consistency of shared data during program execution.

5. As Cheriton points out [Che86a], if the underlying system implements DSM using a communication paradigm then it must support the identification of the recipients, proper data delivery and interpretation of data to ensure coherence of state of the shared object.

5. Design Overview

5.1. The Environment

A bus based network of workstations is used as the distributed environment. The system has been designed to function on a heterogeneous network made up of workstations such as SUN Sparcs, IBM RS6000s etc on Ethernet. We also use ISIS [Bir89a, Bir85a] – a network toolkit based on the notion of process groups, as the primary means of communication and so we require ISIS to be available on each of the architectures.

5.2. The Model

The fundamental component of sharing in the proposed model is an *object*. An object is nothing but the instantiation of a user defined data type. In our scheme the ownership of an object is allowed to vary dynamically from processor to processor. The last writer to an object becomes its current owner. This requires objects to be migrated from the current owner to the new owner. Hence, object mobility is supported insofar as the state of the object is concerned. Because of the high locality that a given process may exhibit, mobility is backed by replication of object state and there may be multiple valid copies of a shared object.

At the highest level the system consists of three types of entities:

- **DSM Manager**: This is responsible for handling all requests to the shared objects and is the primary interface for the user program to it's shared data. One DSM manager exists per node in the system. It is responsible for maintaining the consistency of the shared objects. The DSMM keeps the the information regarding each shared object in a hashed table. Each entry of the hashed table stores the attributes related to the shared object apart from its name. The attributes include the type, status and the current value of the shared object. Whenever there is a request for an access of a shared object by a client at a processor, the DSMM checks the hash table and if the mode is nonconflicting with the current access request then the request is satisfied locally. Otherwise the DSMM communicates with all the other DSMMs to satisfy the request.
- **Semaphore Manager**: Mutual exclusion is now achieved using the notion of a *distributed semaphore*. This is nothing but a semaphore that can be locked or unlocked from any process in the system with access to it. The semaphore manager is centralized and may be replicated only for the purposes of fault tolerance.
- **Synchronization Manager**: Barrier synchronization is an integral part of parallel processing. Hence for any DSM to be useful it should provide a mechanism to synchronize processes executing in parallel. The synchronization manager is used to achieve

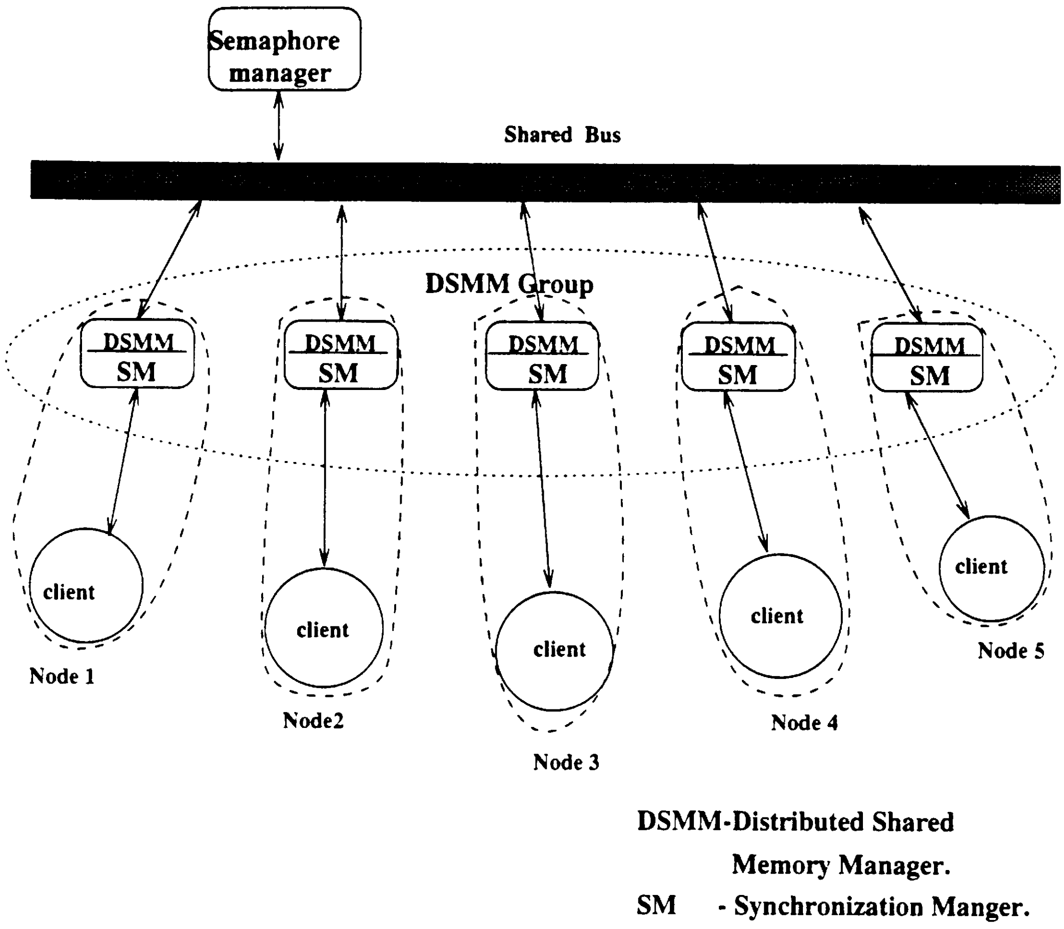


Figure 1: *The DSMM model*

barrier synchronization of a group of communicating processes. This entity is also distributed and one of these exists per node in the system.

5.3. Coherence

The permissible modes an object can be in, and the actions that are taken if an object is in any specific mode are as follows:

- ***read-only***: If the object is in the read-only mode at a given processor, then it will remain in this mode until it is invalidated. At any give time, there can exist more than one read-only copy of an object in the system. The mode essentially permits extracting the state of an object but does not permit mutating it's state.
- ***read-write***: When an object is acquired by a processor in a read-write more to update the object, all the read-only copies of the object are invalidated and then the object is allowed to be updated. At any given time, there can exist only one copy of an object in the read-write mode. The processor having an object in read-write mode is the current owner of the object.
- ***invalid***: This mode forbids any operations on the object. The latest copy of the object's state has to be got from it's current owner before invoking any operations on the object.

These actions of the DSMM are summarized in Table 1.

Current Mode	Request	DSMM Action
read-only	read	Satisfy the request
	write	1. Suspend the requesting process 2. Invalidate all other copies 3. Grab ownership of object 4. Write and restart the suspended process
read-write	read	Satisfy the request
	write	1. Suspend the requesting process 2. Invalidate all other copies 3. Write and restart the suspended process
invalid	read	1. Suspend the requesting process 2. Get the latest state from the owner 3. Make mode *read-only* 4. Return value and restart
	write	1. Suspend the requesting process 2. Invalidate all other copies 3. Grab ownership of the object 4. Write and restart the suspended process

Table 1: *DSMM Actions*

5.4. The Interface

The primary interface to the DSM system is through the three managers which export the following sets of operations:

- **DSM Manager**:
    - ◆ ***register***: This allows a user to inform the world of a newly declared shared object.
    - ◆ ***extract***: This helps the user extract the current state of the shared object. In case the local copy has been invalidated the latest copy of the state is got from the current owner.
    - ◆ ***mutate***: Used to mutate the state of the object, this operation requires that the node invoking the operation is the owner of the target of the invocation. This also implies invalidating all other copies of the object being mutated.
- **Semaphore Manager**:
    - ◆ ***Declare a semaphore***: This allows a user process to declare a system wide binary semaphore.
    - ◆ ***Lock***: This is used to "lock" the mutex under consideration. Protection against possible deadlocks include generation of error messages on trying to lock an already locked semaphore.
    - ◆ ***Unlock***: Simply releases the lock held on the semaphore.

- **Synchronization Manager**:
    - *Barrier*: This is used for declaring a barrier which can be used to synchronize *n* processes.
    - *Synchronize*: This is used for actual synchronization. The calling process essentially blocks till "n" processes have reached the barrier.

The developer now treats the entire distributed system as a shared memory parallel machine. The additional work required is in registering shared objects with the DSM manager. Semaphores can now be declared and manipulated just as on any other shared memory machine. In fact the synchronization manager provides additional functionality that is normally not available on a multiprocessor. The first time the object is registered with the manager as a shared object it's initial value is written through to all the managers involved in the application. Subsequent updates take place only when required. One of the objectives of this project is to provide *network transparency*. This means that the applications programmer need not be concerned about the details of communication and he/she views the entire network as a single shared memory multiprocessor. This transparency is provided by DSM manager at each processor. Requests to manipulate shared objects at any given processor are handled by the DSM manager at that site. Hence, the user program does not have to locate the shared object and maintain its consistency in the distributed environment. In order to be able to support complex user defined data structures we intend to provide language/compiler support so that the composition of the data type being manipulated becomes visible to our system. Figure 1 gives a pictorial representation of the proposed system.

6. Implementation Notes

As we have mentioned in earlier sections we have used ISIS to implement all the managers. Each of the managers has been structured as an ISIS process with the corresponding interface being equivalent to the tasks (entry points) of an ISIS process. The client and the manager service essentially bind by joining a process group and then communicate by means of point to point messages. The managers themselves talk with each other through the use of broadcasts to the entry point under consideration. Here are some points which try to briefly describe the implementation and design features relevant to the managers:

- Every client joins a group with it's manager while all the managers form another group not accessible to the client.
- Each DSM manager maintains a hashed map which maps each object identifier to it's attributes (like type, status etc). This enables quick access to any shared variable at the time of an update(write) or a read.
- The "register" function allocates space for the shared object at the local node and informs other DSMMs of the existence of the shared object.
- The current implementation requires an explicit call to the register entry point of the DSMM at the site.
- The coherence mechanism is built into the manager and is not available to the client to control. In case of "writes", the manager at which the write is requested broadcasts an invalidate and

completes the write only after an acknowledgment is received from each of the others owning a valid copy of the variable.

- An encapsulation of UNIX semaphores is available through the **mutex** class. An interesting feature of the UNIX semaphore is that it outlives the life of the process which created it and hence has to be explicitly destroyed using system calls. Encapsulating it in a C++ class implies that this is automatically taken care of since the destructor of an object is called upon exit and the destructor can contain the appropriate system call.
- The barrier synchronization functions are built in as a set of entry points in the DSMM. Barrier synchronization is mainly to be used to synchronize a set of asynchronous concurrent processes. The way we accomplish this is by initially declaring that a "barrier" which takes as an argument the number of processes to be synchronized. At the time synchronize is called form any process, it increments a local count which keeps track of how many processes have already reached this barrier. We then broadcast to other managers to inform them that another process has reached it's barrier. Each manager, on receiving this message increments it's count. If this count is equal to the number of number of processes to be synchronized then the call to synchronize returns and the process which requested synchronization is allowed to continue.

7. Conclusions

In summary, we feel that it has been a really useful experience trying to translate theory into something useful and practical. C++ and ISIS proved to be good choices, the former because of the clean-style of programming it promotes and the latter for extremely reliable communication that it allows us to provide. As of now, we have not utilized the "fault-tolerance programming facilities" offered by ISIS in our system. However we believe that it is possible to make our DSM managers "reliable" without significant changes to the design by writing the appropriate monitoring routines for group changes. For example a situation in which a manager either goes down or get's isolated from other managers because of a network break can be handled if the group change operation alerts other members of a change in group. One of the managers can then start up the dead manager and bring it up to date on the state of the various shared variables. We can simply invalidate all variables or get a valid copy of each and write it's value in.

We feel that this effort has indeed provided a proof of concept for distributed shared memory at the level of user-defined data structures. What we feel, is essentially required is to be able to integrate this into the compiler so that sharing can be identified without the user ever bothering to make calls to the manager functions. The main advantage of language/compiler support would be static identification of sharing and dependencies. The other logical way would be to add keywords to the language like **shared** so that the compiler recognizes the shared objects and inserts the necessary calls to the DSMM. Compiler support would be the only way to support object mobility where objects are complex entities like those in C++ primarily because we need access to the internal structure of the object for it's mobility. A third area of improvement is in handling heterogeneity. What is essentially required is ISIS on each of the target architectures. Since all object movement is done by sending ISIS messages just having ISIS on target architectures would take care of all of the associated problems.

In retrospective, we hope that this effort will provide a much needed incentive for increased use of distributed systems in parallel processing.

References

[Acc86a] M. Accetta, R. Baron, W. Bolosky, and D. Golub, "A New Kernel Foundation for UNIX Development," *Proceedings of the USENIX 1986 Summer Conference, Atlanta* (1986).

[Ben90a] J. K. Bennett, J. B. Carter, and W. Zwaenepole, "Munin: Distributed Shared Memory Based on Type-specific Memory Coherence," *Proceedings of the 2nd ACM SIGPLAN on Principles and Practice of Parallel Programming*, pp. 168-175 (1990).

[Bir89a] Ken Birman and Keith Marzullo, "ISIS and the META Project," *Sun Technology* **2** (1989).

[Bir85a] Kenneth P. Birman, "Replication and Fault Tolerance in the ISIS System," *Proc. of the 10th ACM Symposium on Operating System Principles, Operating Systems Review* **19**, pp. 79-86, ACM (December 1985).

[Che86a] David R. Cheriton, "Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems Design," *Proceedings of the 6th International Conference on Distributed Computing Syatems*, IEEE Computer Society (May 1986).

[Das88a] P. Dasgupta, R. LeBalnc, and W. Appelbe, "The Clouds Distributed Operating System: Function Description, Implementation Details and Related Work," *Proceedings of the IEEE International Conference on Distributed Computing Systems* (1988).

[Fle89a] B. D. Fleisch and G. J. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proceedings of the 12th ACM Symposium on Operating System Principles*, pp. 211-223 (1989).

[Jul88a] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black, "Fine Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems* **6**, pp. 109-133 (February 1988).

[Kes89a] R. E. Kessler and Miron Livny, "An Analysis of Distributed Shared Memory Algorithms," *Proceedings of the 9th Distributed Computing Symposium*, IEEE (1989).

[Li86a] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," *Doctoral Dissertation*, Yale University, Department of Computer Science (1986).

[Nit91a] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, pp. 52-60 (August 1991).

[Ram89b] Umakishore Ramachandran, M. Yousef Khalidi, and Mustaque Ahamad, "Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer," *Pro-*

ceedings of the International Conference on Parallel Processing (1989).

[Ram89a] U. Ramachandran and Y. Khalidi, "An Implementation of Distributed Shared Memory," *Proceedings of the Distributed and Multiprocessor Workshop*, pp. 21-38 (1989).

[Stu90a] M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, pp. 54-64 (May 1990).

[Tam90a] Ming-Chit Tam, Jonathan Smith, and David J. Farber, "A Taxonomy Based Comparison of Several Distributed Shared Memory Systems," *Operating Systems Review* (July 1990).

[Tan90a] A. S. Tanenbaum and R. van Renesse, "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems," *Operating Systems Review* **24**, pp. 40-67 (July 1990).

Specification and Implementation of Cooperation Paradigms for Distributed Applications

Martin Zimmermann

J. W. Goethe University
Frankfurt, Main Germany
zimmerma@informatik.uni-frankfurt

Abstract

In this paper, we present an integrated approach for the specification, implementation and management of distributed applications. Driven by the basic characteristics of distributed applications, we introduce a distributed application model enabling the integration of different aspects during the development phase of a distributed application. Based on this model, we explain our specification technique for interfaces, components, and application configurations.

In order to support reusability of distributed applications, we propose some research directions for the design of generic distributed applications. For this purpose, we introduce the concept of templates, which serves as a framework that provides a skeleton for developing distributed applications with a specific cooperation pattern. Templates are a technique for making application specifications as general and flexible as possible. It allows interfaces, components and application configurations to have generic parameters. During application engineering a generic distributed application is reused to construct specific applications, i.e. a concrete running application through an instantiation process. Based on practical examples, we demonstrate the usefulness of our approach by specification of templates for client server applications and different types of distributed group work applications.

Moreover, we will illustrate an object-oriented realization of the specification technique and discuss several tools supporting the specification and implementation steps.

1. Introduction

A distributed application is composed of a set of cooperative application components. The name results from the fact that such applications typically execute on distributed systems, such as local area networks. There is a growing demand for distributed applications, especially in the areas of computer integrated manufacturing, office automation and cooperative work. Although the advantages of distributed applications are well known and widely recognized many questions are still open, especially adequate abstraction mechanisms which provide appropriate and user-friendly specification and implementation techniques.

New models such as Open Distributed Processing (ODP) [Gei90a] intend to address distributed applications from the application point of view by masking distribution aspects. Meanwhile, there are numerous activities in standardization bodies, in research projects and in the commercial sector in order to define a general framework for distributed application development, such as DAF [CCI90a], SE-ODP [ECM90a], ANSA [ANS89a], REX [Mag90a] and OSF-DCE [OSF90a].

From the application viewpoint, major goals to be achieved by distributed applications are the decentralization of functionality and data, the shared use of distributed and expensive resources and the realization of parallelism between loosely coupled components.

From the management viewpoint, distributed applications are long live applications. This implies that new components can be inserted or deleted during runtime. This requires mechanisms which allow to express consistency properties, e.g. configuration properties like the availability of components with certain characteristics. Moreover, in order to support a consistent integration and termination of components, we need appropriate language support for the specification of configuration steps to be performed on integration or termination of components. As a consequence, from the management viewpoint mechanisms for monitoring and controlling a distributed application configuration have to be provided.

Specification of distributed applications is a quite complex activity. In order to support reusability at specification level, a designer should be able to describe generic distributed applications. Distributed applications are called generic when they are created from specific ones through an abstraction process. A generic application is represented in our specification technique as a set of interface-, component- and application configuration templates.

The paper is organized as follows: As an integration effort Section 2 formulates a model for distributed applications, which meets the requirements resulting from the basic characteristics of distributed applications. The model is then used as a base for an integrated specification technique for distributed applications. In Section 3 we introduce the basic concepts of our interface-, component- and application configuration language. In order to support reusability of specifications, Section 4 presents an extension which allows the specification of generic distributed applications based on templates. Section 5 discusses how the specification technique can be mapped onto an object oriented architecture using C++. Finally, Section 6 gives the current status of our research and concludes the paper.

2. Distributed Application Model

Distributed applications are built from a set of interacting application components which form the basic building blocks of a distributed application (see Figure 1). Each application component could be defined context independent, i.e. there are no references to other components. Components are related to each other by the specification of bindings between interfaces. The application behaviour of an application component is defined by the interaction behaviour at its application interfaces.

Distributed applications are long living applications which have to be monitored and controlled during runtiome. In order to enable management activities, each application component must provide management interfaces which define operations for the control, maintenance and monitoring. Distributed application management involves establishing and terminating of a distributed application within a given computer network. In the establishment phase, the application components have to be located on the nodes, initialized properly and finally the bindings have to be established. During runtime, we have to provide mechanisms for monitoring and controlling the behaviour of the application components. Management is performed by management components, which are associated with the application components via management interfaces.

The communication-oriented aspects are integrated by the concept of communication contexts, which describe the communication requirements of an application component, like the communication relation (connection-oriented or connectionless), the interaction type (message-oriented or operation-oriented) and the properties of a transport service such as time contraints and the desired throughput.

A communication context can be assigned to each interface of an application component to deal with their varying communication needs. Alternatively, a default communication context can be defined, which enables the specification of a single communication context for an application component as a whole. Furthermore, we are able to assign more than one communication context to an interface to reflect that this interface is accessible via different communication services.

In the paper we concentrate on the application and management oriented aspects. The concept of communication contexts is introduced in [Fel91a].

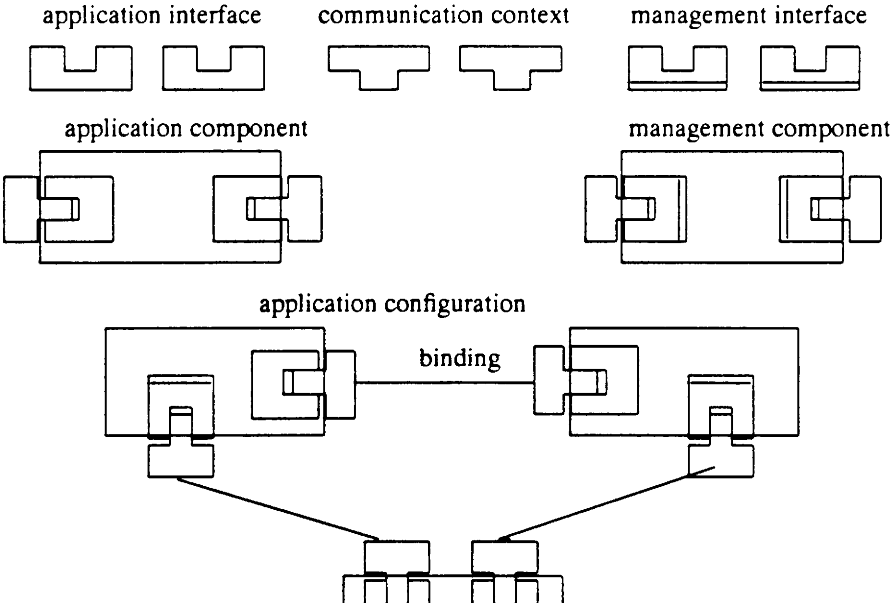


Figure 1: *Distributed Application Model*

3. Specification Support

3.1. Interface Specification Language (ISL)

Our interface concept is based on a bidirectional interaction specification and describes what operation each of a pair of components could request the other to perform. The specific behaviour at an interface, i.e. the determination whether an application component is acting as a consumer and/or a supplier, is defined when we describe how a component is constructed from a set of interfaces. Moreover, each operation specification could be decorated with attributes, such as an attribute declaring an operation to be performed atomically.

In contrast to other approaches, our ISL allows in addition to operation specifications, the integration of a cooperation protocol, which is an optional part of an interface specification. A cooperation protocol allows the specification of regulations, i.e. which operation should be executed on which interface and by whom. This allows the specification of the dynamic cooperation behaviour at an interface, i.e. the behaviour can be restricted by the definition of a cooperation protocol in terms of sequencing rules, synchronisation and responsibility.

Sequencing rules are useful for the specification of ordering constraints for operation invocations and can be expressed by extended path expressions. Synchronisation is necessary if the interactions of two or more components have to be coordinated. Responsibility aspects are needed if we want to express that the initiation of an interaction has to be performed by a consumer requiring certain properties.

In our approach, roles are used to express synchronisation properties and responsibility aspects, i.e. the initiation of an operation is only allowed if the invoking component has a specific role (Figure 2). Moreover, roles can be assigned statically to a component or requested dynamically.

Figure 2 illustrates an example from the office procedure domain. In our context, an office procedure is represented as a component specification. The related interface in Figure 2 defines the operations separated in a consumer and supplier part. The cooperation protocol defines the required roles for initiating the operations and a path expression defining the sequence rules.

The separation of operations and cooperation protocol within an interface specification supports reusability (i.e. the cooperation protocol and the operations could be extended or changed separately). This modu-

```
officeProcedureInt INTERFACE
        CONSUMER INVOKES {
                fillin ExpenseData
                fillinManagementData
                ...}
        SUPPLIER INVOKES {
                approved }
        COOPERATION PROTOCOL {
                ROLES { Manager, .. }
                fillinManagementData REQUIRES ROLE Manager
                fillinExpenseData < fillinManagementData < .....
                ......}
```

Figure 2: *Example of an interface specification*

larity is very important for the design of an interface ("What is to be done?" is not mixed with "When does it have to be done?"), as well as for readability.

3.2. Component Specification Language (CSL)

From the application viewpoint, a component is defined by its interfaces and a related behaviour (supplier and/or consumer). Regarding the cooperation at an interface, we could associate each component with a set of potential roles according to the interface specification. This allows the specification of restrictions in respect to its behaviour, e.g., a client component may only be allowed to act as a reader when interacting with a file server. Moreover, for each potential role we could specifiy a request policy. We distinguish between implicit and explicit role requests. In the first case, a role is requested automatically when an operation is invoked.

A binding specification enables the definition of the allowed binding topology (single/multiple binding) and the component responsible for establishing the binding. Moreover, the time for the establishment and termination of a binding can be specified. For example, binding could be established at instantiation time or interaction time. In the second case, binding is only established when an interaction takes place at an interface. A component could be associated with a scheduling policy if it provides an interface which is decorated with the attribute multiple binding. There are some predefined scheduling policies, such as FCFS. However, sometimes an application dependent policy is required, e.g. a disc driver needs an internal scheduling policy to manage a disk.

We distinguish between interactive and not-interactive distributed applications. The attribute interactive supports the requirements of CSCW and office procedure applications. Interactive applications are composed of interactive components and require a set of users to work properly. This attribute has impact on the application configuration process, e.g., an interactive distributed application may not be established until the required users are available personally. In this case, establishment of an application configuration has to be delayed until all the required users are available.

Composition of components is provided by different composition schemes. A hierachical composition enables components to be constructed of subcomponents and related bindings. Moreover, collections of components, i.e. aggregates of same type components, can be defined. For this purpose, we support the notation of component groups.

A component or component group could be enriched with constraints, which describe properties valid at each time of a running distributed application. Constraints can be described in terms of configuration restrictions. For a component group we could specify a constraint which is valid independent of the current amount of members. Constraints are defined by predicates which have to be valid either for all components or part of them (e.g. all components having a specific role).

Distributed applications are characterized by evolutionary changes during runtime. Components could join or leave a distributed application. Dependent on the state of the components of a running distributed application, a new component have to be initialized properly. Moreover, on termination of components, the resulting distributed application must be kept in a consistent state. However, the merge activities

```
officeProcedureComp COMPONENT
APPLICATION PROPERTIES {
    INTERFACES
        SUPPLIER AT
            OfficeProcedureInt
    BINDING PROPERTIES
        MULTIPLE BINDING RESTRICTED TO n
        IMPLICIT ESTABLISHMENT {
            AT INTERACTION TIME }
```

Figure 3: *Example of a component specification*

and their ordering are often application dependent. To reflect this properties we have developed a method for specifying instantiation and termination rules. This concept enables a consistent integration of newly created application components into a running distributed application and also a correct termination of components. Moreover, they allow the encapsulation of all application dependent actions which have to be performed on configuration changes. For example, in a distributed application representing a set of dining philosophers, instantiation of a new philosopher means that dependent on the application state of its neighbours, a new fork has to be created.

Figure 3 illustrates a component specification representing an office procedure.

3.3. Application Configuration Specification Language (ACSL)

Our configuration language follows the concepts introduced in [Kra90a]. A complete configuration of a distributed application describes the types of application components from which the distributed application is to be constructed, the instances, how these instances are interconnected and optionally where they are located. Additionally, in case of interactive applications we need a specification of the required users which are associated with the components.

A binding between two interfaces of different components is only possible if the interfaces and the related communication contexts can be matched. This means that either two related interfaces must both be symmetric or one of them must act in a supplier role and the other one must act in a consumer role.

Moreover, time restrictions can be defined for the establishment of distributed applications. For example, if not all the members from the critical member set are available the distributed application can not be established. An application configuration could be decorated similar to a composite component with a set of instantiation/termination rules and a set of constraints.

4. Templates

If we analyze distributed applications, we could find typical cooperation paradigms, independent of the specific functionality of an application. Each cooperation paradigm is a model of an interaction pattern between the components of a distributed application.

From the application designer's point of view, it would be helpful to provide a set of predefined generic distributed applications representing specific cooperation paradigms. Breaking down distributed applications into generic distributed applications enables us to structure them in advance, enabling the designer to be guided and understood more efficiently in their applications. For this purpose, we have extended our specification technique by providing the concept of templates. Templates are a technique for making application specifications as general and flexible as possible. They serve as a framework that provides a skeleton for developing distributed applications with a specific cooperation pattern. Templates can be regarded as black boxes by completing partially defined behaviour. It allows interfaces, components and application configurations to have generic parameters. As a consequence, a cooperation paradigm could be expressed by the specification of a set of related interface-, component- and application configuration templates containing generic parameters. During application engineering a generic distributed application is reused to construct a specific application, i.e. a concrete running application through an instantiation process.

Figure 4 illustrates as an example the basic structure of an application configuration template graphically. It is composed of a generic configuration specification and a set of properties: The application is interactive, has a set of related constraints, and instantiation and termination rules. In the following two cooperation paradigms, client server and group cooperation are discussed in more detail.

4.1. Client Server Cooperation

Client server applications are asymmetric applications, consisting of a set of client components and a server component, i.e. regarding the configuration, we have a n:1 binding topology, each client component is bound to the server component. A client is a triggering component acting as a consumer at an interface. Clients make requests that trigger reactions from server components. A server is a reactive component supplying operations at an interface. Moreover, a server allows multiple binding at its interfaces. Regarding the lifetime, a server is usually a nonterminating component. Its main purpose is to manage a collection of resources and to service requests from any client who wants to access those resources. Service requests are performed according to a scheduling policy. In order to coordinate concurrent access to the resources we need appropriate synchronisation mechanisms between the client components.

A generic specification for a client server application consists of the following parts: An interface template which defines an asymmetric behaviour. The operations are generic parameters, which have to be

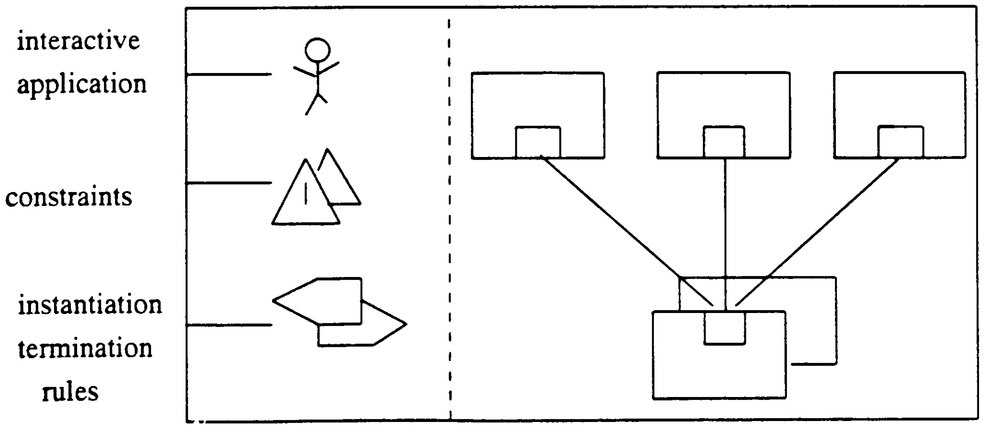


Figure 4: *Example of an application configuration template*

provided when defining a specific client server application. Synchronization between client components is expressed by a role template providing roles Reader and Writer. When using an interface template, a designer has to define which operations require the Reader and which require the Writer role.

At component level, we have a server and client component template. For example, a server component template is a component acting as a supplier at an interface, allowing multiple bindings (see Figure 5). The job control policy determines the mechanism for performing the incoming requests. We support some basic scheduling policies.

The configuration template contains a generic specification of a n:1 configuration.

4.2. Group Cooperation

In a distributed group work application we have a set of same type components which cooperate in order to provide an application functionality. We distinguish between *explicit* and *implicit* group cooperation. In the first case, interaction is performed directly between the group members of a group whereas in the second case cooperation is performed via one or more shared components.

4.2.1. Implicit group work

Applications following this cooperation paradigm are composed of a set of same type components representing the group members of a group (consumers) and one or more components representing shared objects. Cooperation takes place implicitly by interaction of the group members with their shared components. These kinds of applications have in common that one or more actions on a shared component are to be carried out by the members of the group. Moreover, certain group members may be responsible for certain actions and the actions may be dependent on each other, e.g. they have to be performed in a specific order.

One application which follows this cooperation type is a group of replicated workers which are responsible to perform a bag of tasks (represented as shared components), e.g. distributed calculation of the adaptive quadrature problem [And91a]. Initially, there is one task corresponding to the entire problem to be solved and a set of worker components responsible for solving the problem. In order to solve a task a worker component has to establish a binding to a task, often generating new tasks corresponding to subproblems. The cooperation terminates when all tasks have been processed.

Another type of application based on implicit group work are applications from the office procedure domain. A common example of an

```
<< server >> COMPONENT
        APPLICATION PROPERTIES
                SUPPLIER AT << interfaces >>
                        MULTIPLE BINDING RESTRICTED TO n
                        SCHEDULING << policy >>
        COMMUNICATION PROPERTIES
                << communication contexts >>
        MANAGEMENT PROPERTIES
                REPLICATION << policy >>
                MIGRATION << policy >>
```

Figure 5: *Template for a server component*

office procedure application is the processing of a travel expense form requiring interactions of an initiator (employee), of his manager and secretary, and of the travel expense office. An office procedure typically consists of a set of execution steps which are handled by actor components which represent the group members of a related group [Sch91a]. In this application, the cooperation protocol can be regarded as a routing specification which describes all the required steps (operations) to execute the office procedure and the responsible components.

Using our specification technique, implicit group cooperation is supported by the following templates: An asymmetric interface template is used to represent the cooperation between a group member and the shared component. There are three generic parameters: The operations which have to be initiated by the worker components in order to perform the different steps of the cooperation protocol, the required roles to initiate the operations, and finally a path expression in order to express the specific sequence ordering. A role template defines the required roles and their relationships, i.e. who is responsible for each single execution step. Component templates are provided for both the shared component and the worker components.

The shared component template is defined by a supplier behaviour at its interface and a binding policy which defines that binding has to be established at interaction time and terminated when the cooperation protocol has been completed. This means, after performing each single step of the office procedure a new worker component responsible to perform the next step has to be selected and an appropriate binding must be established. Selection of a worker component is driven according to the role requirements of the actual operation to be performed.

Finally, an application configuration template defines the overall application structure. This involves the specification of the application configuration, its constraints and instantiation/termination rules for integration of new component instances. Associated with the application configuration is a constraint about the required worker components which must be available. It defines that each required role to perform the cooperation protocol must be available by at least one worker component. Informally, this means that at any given time enough worker components must exist to perform the cooperation protocol completely. This also implies, that we have to check the constraints whenever a configuration change takes place.

```
<< implicitGroupWork >> DISTRIBUTED APPLICATION
        COMPONENTS
                << shared >> [ ]: << SharedComponent >>
                << worker >> [ ]: << WorkerComponent >>
                        INSTANTIATION BY Management
                        INITIAL MEMBERS << n >>
                        INITIAL ROLES
                                << worker >> [ 1 ] ROLE << role1 >>
                                ...

        BINDINGS
                FOR ALL i: MEMBER OF << shared >>
                        << shared >> [ i ] - << worker >> [ 1 ]
                        ...
        CONSTRAINTS
                FOR ALL r: ROLES OF << impGrWorkRoles >>
                        EXISTS j: << worker >> [ j ] HAS ROLE r
```

Figure 6: *Configuration template for implicit group work*

The constraint concept also provides means to express placement restrictions for components. For example, one may wish to express that a shared component has to be located at the actual consumer. This constraint implies, that office procedure components are mobile components which are moved dynamically between the worker components.

Normally, distributed applications cooperating according to implicit group work are long duration activities [Sch91a], which have to be monitored and controlled during livetime. For this purpose, an interface should support generic operations for monitoring and control. Monitoring involves support for status inquiries to retrieve the current status, e.g., in case of mobile components we need operations to return the node where the shared component is currently located or to return the worker which represents the actual consumer. Operations supporting control include status management operations which allows to cancel a cooperation or freeze their actual status.

4.2.2. Explicit group work

Applications following this cooperation paradigm are composed of a set of peer components each providing one or more symmetric interfaces. Each component contains one or more subcomponents representing resources which could be accessed and manipulated by the other group members.

This category involves applications like replicated servers, date planning applications or applications from the CSCW domain. For example, a replicated server is a group of server components that each do the same thing. Replication serves one of two purposes: It can either be used to speed up finding a solution to a problem by dividing the problem into independent subproblems that are solved concurrently by multiple worker components (implicit group work) or it can increase accessibility of resources. For example, a group of n server components could manage n copies of a data file. Each group member provides an identical client interface. However, the server components themselves have to cooperate to present clients with the illusion that there is only a single copy of the file. As a consequence, the members of the server group need to synchronise with each other in order to solve the file consistency problem. For this purpose, we need appropriate group cooperation mechanisms between the members in order to guarantee that at the same time a writing client has exclusive access to a replicated resource.

Using our specification technique, explicit group cooperation is supported by the following templates: A group interface template is used to represent interactions between the group members of a component group. Synchronisation properties are integrated by the specification of a related role template which defines conditions for initiating group interactions at a group interface. For example, a role template could define synchronisation properties in order to provide mechanisms like floor control for synchronous interactive applications. Component templates which represent the members of a group are composed of symmetric interfaces. Each interface of a component is associated with a role request policy, which is a generic parameter. For example, a server component may request implicitly a role Writer before initiating a write interaction. In the application configuration template the bindings between group members are specified: Each member is connected with all the group members as a star.

Regarding the amount of members, we distinguish between static groups and groups of an indefinite amount of members (open group). In the second case, a group can be decorated additionally with attributes, such as an initial and critical member set specification. Moreover, in order to enable the integration of new members during runtime, a configuration template could be decorated with a set of instantiation and termination rules.

Using the concept of instantiation rules, we could specify all the activities for merging a new member. For example, a new server *S* wishes to join a server group *SGroup*. The join requires the new member *S* to be initialized properly. Initialization is typically performed by aquiring a consistent state from the other group members. In case of a replicated server this implies, that after joining a group, a new server component should aquire valid resources (e.g. files) from the other members to merge into the group's activities. Moreover, all the required bindings have to be established. Termination rules determine how to leave a group. Typically, leaving of a member from a group is performed by releasing all roles that it holds. Moreover, the binding information has to be modified to get a consistent configuration.

5. Implementation Support

For the implemention of our approach we have mapped our abstract component model onto a set of interacting C++ objects. Based on the formal specification, the implementation of a distributed application is supported by a set of tools which facilitates automatic derivation of object-oriented implementations.

An interface specification is mapped onto a set of class definitions. Instances of interfaces are represented as objects. For each interface we can define a related application/management class, which realizes the interface specification. Additionally, in order to enable a remote access, we need stub objects at the consumer and supplier side. They function as local representatives of remote objects.

The cooperation protocol feature of an interface is implemented by a base class realizing the userdefined specification. In our approach, we model path expressions of an cooperation protocol using petri nets, roles are mapped onto related role objects which provide operations for requesting and releasing roles.

Management objects are responsible for providing management operations. For this purpose, we provide a class library realizing management properties. We have developed a classification scheme which allows to gain systematically the management facilities for a distributed application.

Built in management facilities are provided for each application component and component group independent of its functionality. They reflect monitoring properties like monitoring the configuration of a component (static and dynamic information about interfaces, subcomponents, communication contexts and bindings), test of liveness of a component, status of a component and its memory/processor usage.

Derived management facilities can be directly obtained by analyzing the specification of the application properties and communication contexts of a component. For this purpose, we have to evaluate attributes such as the behaviour and binding characteristics of components and component groups. For instance, regarding the behaviour at an interface, monitoring facilities, such as work load monitoring, response time monitoring, throughput monitoring and queue monitoring have to be supported in case of a server behaviour. If a component specification is decorated with a constraint definition, a management object is responsible for monitoring and control the constraints after performing configuration changes in order to preserve consistency of a component. This category also involves monitoring and controlling the critical member set of a group before joining or leaving of members.

Optional management properties have to be declared explicitly by the designer of an application component. They involve configuration management facilities for performing changes during runtime at component and application level. At component level, the declaration of management properties such as replacement, replication, migration and checkpointing of components are supported by an extended CSL.

As a result, from the implementation viewpoint, the construction of application components is supported by configuration of a set of C++ objects (Figure 7).

A management component is responsible for the establishment of a distributed application. Establishment includes the determination of suitable nodes for each component (dependent on the component properties and the node facilities), distribution of the related source text, compilation, creation of component instances, their initialization, and setting up the defined bindings. Moreover, a management component enables monitoring and initiating dynamic changes (dependent on the management characteristics of each application component) during runtime.

An application configuration specification has to be transformed into a computational representation. This is achieved by mapping a specification onto a related object, which can be generated automatically from a formal specification. The public interface supports the creation of components and bindings as well as the establishment and termination of the related distributed application. The properties of each application component are represented as member objects. From the management point of view, this object architecture allows a fast access to an application configuration and its consistent computational representation. For the administration of configuration objects, i.e. creation, removal and retrieval of application configuration objects, a dictionary object provides appropriate operations.

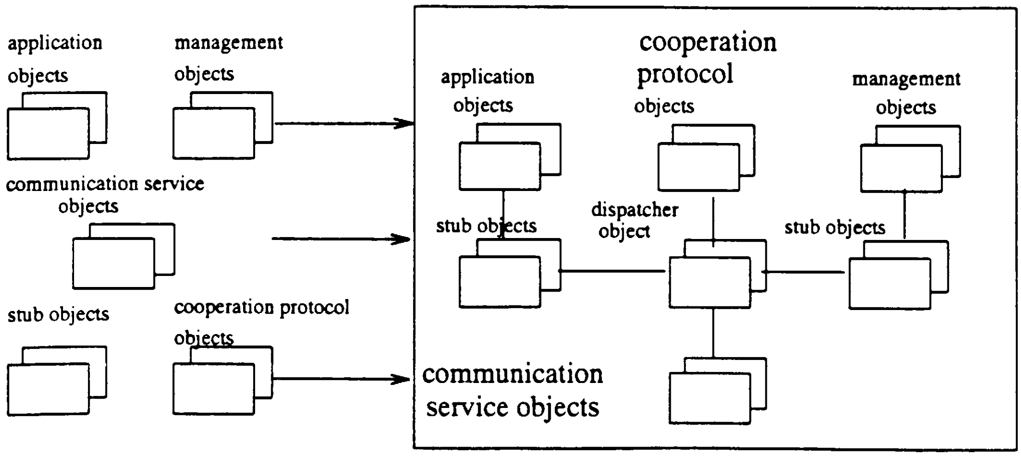


Figure 7: *Configuration of an application component*

This approach has several advantages. First of all, a declarative configuration description can be mapped onto the creation of a computational application configuration object which are a representative of the distributed application. Furthermore, a "normal" application component can be responsible for the configuration, which means, to define and initiate a dynamic extension of a distributed application without the need of a separate configuration description. Consequently, providing the concept of configuration objects there is no need to extend a programming language in order to support a dynamic configuration modification during runtime initiated by an application component. This way interactive and programmed configuration and reconfiguration of distributed applications are possible.

6. Conclusion

In this paper we have developed a new approach for the specification, implementation and management of distributed applications. Our specification technique supports the specification of quite different aspects of a distributed application, namely application behaviour, management facilities and communication contexts. Reusablity is supported by the concept of templates which can be used as a base for the development of generic distributed applications. We have demonstrated the usefulness of templates by some well known cooperation paradigms, client server applications and different types of group work applications.

The clear separation of application-, management- and communication-oriented aspects of our concept is a solid base for a constructive approach for building of distributed applications. Moreover, the consecutive steps of specification and implementation allow the validation at specification level and the automatic generation of the implementation. At the implementation level, the different aspects could be integrated into a general object-oriented architecture. As a consequence, modularity and reuse of software is improved.

We are currently working on a rapid prototype of the described approach based on the C++ programming language and X windows. This includes compilers for the specification languages and management tools for the representation and interactive control of distributed applications. Some basic concepts of our approach have been validated by a prototype implemenation of the User Agent-Message Store protocol (MHS P7) as part of a joint research project with Digital Equipment Corporation [Doe91a].

Future work will focus on the development of a library of generic distributed applications and composition schemes for combining generic distributed applications. A graphical support environment for the development and management of distributed applications will be realized. Furthermore, we have to analyze how to derive automatically management and communication properties from the application specification.

References

[And91a] G. R. Andrews, "Paradigms for Process Interaction in Distributed Programs," *ACM Computing Surveys* (1991).

[ANS89a] ANSA, *Reference Manual*, APM Ltd., 24 Hills Road, Cambridge CB2 IJP, UK, March 1989.

[CCI90a] CCITT, *Study Group 7: Support Framework for Distributed Applications (DAF)*, 1990.

[Doe91a] P. Doemel et al., "Concepts for the Reuse of Communication Software," *Technical Report 5/91, University of Frankfurt/M* (May 1991).

[ECM90a] ECMA, "Support Environment for Open Distributed Processing (SE-ODP)," *ECMA TR/49* (January 1990).

[Fel91a] M. Feldhoffer, "Communication Support for Distributed Applications," *Int. Workshop on ODP*, Berlin (October 1991).

[Gei90a] K. Geihs, *The Road to Open Distributed Processing*, Technical Report No. 43.9002, IBM European Networking Center (ENC), 1990.

[Kra90a] J. Kramer, J. Magee, and A. Finkelstein, *A Constructive Approach to the Design of Distributed Systems*, 10th International Conference on Distributed Computing Systems, May 1990.

[Mag90a] J. Magee, J. Kramer, M. Sloman, and N. Dulay, *An Overview of the REX Software Architecture*, Second IEEE Workshop on Future Trends of Distributed Computing Systems in the 1990s, 1990.

[OSF90a] OSF, *Distributed Computing Environment Request for Technology, DCE Framework – Preliminary Position Paper*, Open Software Foundation, January 1990.

[Sch91a] A. Schill, "Distributed system and execution model for office enviroments," *Computer Communications* **14**(8) (October 1991).

Implementing a Modular Object Oriented Operating System on Top of CHORUS

Paulo Amaral   Rodger Lea
Christian Jacquemot

Chorus systèmes
France
rodger@chorus.com

Abstract

Building distributed operating systems benefits from the micro-kernel approach by allowing better support for modularization. However, we believe that that we need to take this support a step further. A more modular, or object oriented approach is needed if we wish to cross that barrier of complexity that is holding back distributed operating system development. The Chorus Object Oriented Layer (COOL) is a layer built above the Chorus micro-kernel designed to extend the micro-kernel abstractions with support for object oriented systems. COOL v2, the second iteration of this layer provides generic support for clusters of objects, in a distributed virtual memory model. This approach allows us to build operating systems as collections of objects. It is built as a layered system where the lowest layer support only clusters and the upper layers support objects.

1. Introduction

Building distributed systems is difficult simply because the complexity of interactions among entities scattered on a collection of machines is enormous. The distributed systems community has long been wrestling with this complexity and has developed methods such as RPC, group communications, distributed shared memory etc. in an attempt to provide mechanisms that abstract over some of this complexity.

However, in attempting to build systems that actively use these mechanisms we have run into two major problems, performance and integration. Performance because we have tried to add these mechanisms to existing systems, and integration because we have tried to do so in an ad-hoc manner without fully considering how these tools should interact, or how applications will use these services.

Work in the operating system community has tried to deal with these issues by re-visiting our existing operating systems and looking at the minimum abstractions necessary to build distributed operating systems.

By combining these with a system building architecture that stresses modularity, we can begin to address the performance and complexity issues. This approach, often called the *micro-kernel* approach, allows us to provide a minimum set of abstractions that can be used to build operating systems themselves.

We feel however, that while this is the correct approach, it is only one step in the right direction. We need to augment our basic mechanisms with a framework that allows system builders to glue functional components together in a coherent and performant way. In effect, we need to provide a system building environment that supports a programming model, tools and services needed to work within that framework.

The object oriented paradigm offers a solution to this problem by offering a framework for building large complex applications, such as OS's in a way that is amenable to distribution. However, we must not repeat the mistakes of early distributed system builders by trying to impose a model on a set of mechanisms, rather, we must actively support the model at the lowest layers in our system, by making sure that our abstractions are suitable for supporting objects [Bla92a].

In this paper we discuss how the COOL system has been designed to exploit the unique features of the Chorus operating system model to provide an efficient set of abstractions that are well suited to support the object oriented metaphor. We stress that this approach not only facilitates building distributed OS's, but any distributed object oriented application, because it reduces the mismatch between our OO services and the model we use to build distributed applications.

Our goal is to provide a framework that will allow operating system builders to develop their applications, the operating system, in a well structured, flexible and coherent environment.

We will introduce the basic COOL v2 architecture, and then concentrate on the Persistent Context Space model that we have developed to allow us to efficiently support distributed, shared objects at the lowest layer in the system.

2. COOL v2

The COOL project is now in its second iteration, our first platform, COOL v1,[†] was designed as a testbed for initial ideas and implemented in late '88 [Hab90a, Des89a, Lea91a].

Our early work with COOL (COOL v1) consisted of experimentation in the way that systems could be built using the object oriented model, and how this supported distributed applications. In an attempt to move the COOL platform from a testbed towards a full object oriented operating system we began a redesign of the COOL abstractions in 1990. This work was carried out in conjunction with two European research projects, both building distributed object based systems, the Esprit ISA project and the Esprit Comandos project [Cah91a].

The result of this work has been the specification of the COOL v2 system and its initial implementation in late '91, [Lea92a, Ama92a].

[†] COOL v1 was a joint project between Chorus Systemes, the SEPT (Service d'Etudes des Postes et Telecommunications), and INRIA (Institut National de Recherche en Informatique et en Automatique)

3. The COOL v2 Architecture

COOL v2 is composed of three functionally separate layers, the COOL-base layer, the COOL generic run-time and the COOL language specific run-time layer.

Our goal when designing this architecture was twofold, efficiency and flexibility. We wanted to support distributed interactions using a number of base mechanisms.

To allow objects to interact we can;

- Support a communications mechanisms that will allow transparent invocation.
- Allow objects to migrate between contexts by unmapping from one context and mapping into another, relocating internal pointers on the fly.
- Use a distributed shared memory mechanisms that ensures object level faulting.

Each of these mechanisms have their advantages and their drawbacks, and each will be used in different circumstances. A key element of our work is that the three levels of the architecture, interact to provide all three mechanisms, allowing policy to decide which mechanisms to be used at which particular time.

In the following sections we briefly outline the functionality of the three levels and then return to the base level and explain further its support for a distributed virtual memory model and its implementation as a distributed system support layer.

3.1. The COOL Base

The COOL-base is the system level layer. It has the interface of a set of system calls and encapsulates the CHORUS micro-kernel. It acts itself as a micro-kernel for object-oriented systems, on the top of which the generic run-time layer can be built. The abstractions implemented in this layer have a close relationship with CHORUS itself and they are

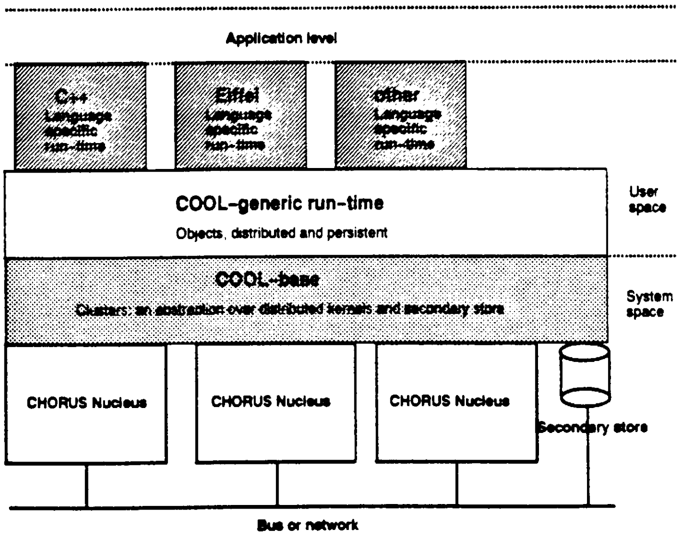

Application level
C++ Language specific run-time
Eiffel Language specific run-time
other Language specific run-time
COOL-generic run-time
Objects, distributed and persistent
User space
COOL-base
Clusters: an abstraction over distributed kernels and secondary store
System space
CHORUS Nucleus
CHORUS Nucleus
CHORUS Nucleus
Secondary store
Bus or network

Figure 1: *COOL v2 architecture*

intended to benefit from the performance of a highly mature microkernel.

The COOL-base provides memory abstractions where objects can exist, support for object sharing through distributed shared memory *and* message passing, an execution model based on threads and a single level persistent store that abstracts over a collection of loosely coupled nodes and associated secondary storage.

In our initial work with COOL our base level supported a simple generic notion of objects. This proved to be too expensive in terms of system overhead so that in COOL v2 we have moved the notion of objects out of our base layer and replaced it with a more generic set of abstractions which we term the *Persistent Context Space* model (PCS).

The persistent context space supports a basic abstraction, the *cluster* which is a set of virtual memory regions and provide a repository for objects. Clusters, being persistent, are represented on secondary storage using the CHORUS abstraction of a segment, and are represented in memory using the CHORUS abstraction of regions.

Clusters are grouped together into *containers* which represent collections of objects whose references are completely contained, i.e. all references within clusters point into clusters within the same container.

A *context* abstracts the notion of an address space, and provides a place into which containers can be mapped for execution. To support distributed shared memory we define the *context group* which is a collection of contexts, on one or more sites, that map identical containers.

We will return to the PCS model and its implementation in Section 4.

3.2. The COOL Generic Run-Time

The generic run-time implements a notion of objects. Objects are the fundamental abstraction in the system for building applications. An object is a combination of state and a set of methods. An object is an instance of a class which defines an implementation of the methods. The generic run-time has a sub-component, the virtual object memory that supports object management including: creation, dynamic link/load, fully transparent invocation including location on secondary storage and mapping into context spaces.

Two types of object identifiers are offered by the generic run-time: domain wide references and language references. A domain wide reference is a globally unique, persistent identifier. It may be used to refer to an object regardless of its location. A language reference is a pointer in C++ and is valid in the context in which the object is presently mapped.

The generic run-time defines the primitives to convert one type of reference to the other one. When a domain wide reference to a remote object is converted to language reference a proxy associated to the object is created [Sha86a]. This proxy is used to transparently invoke the remote object.

Objects are always created in clusters. Each cluster's address space is divided into three parts: the first one is used to store all the structures associated with the cluster used by the generic run-time, the second one is used to store the applications objects, and the last one is used to store the proxies. A different allocator is associated to each part, this allocator is used to allocate and free space.

The classes are structured in modules (set of classes, unit of code). The generic run-time allows the code to be dynamically linked. The generic run-time offers a primitive to link a module. Each class contained in the module are store at the context level. When an instance of a class is created in a cluster, the class descriptor is saved in the cluster. This class descriptor is used to retrieve the appropriate module and therefore the appropriate class when a cluster is remapped in another address space.

The generic runtime provides an execution model based on the notion of *activities* which are mapped onto CHORUS kernel supported threads and *jobs* which models distributed execution of activities. Each cluster can support multiple activities, with more than one activity capable of running within the same object at any particular time.†

One of the main problems with trying to use a single generic base to support multiple language level models is that of semantics. Most languages, and systems, have their own semantics, each of which are subtly different. To enable the building of sophisticated mechanism that support multiple models we have defined a generic run-time to language interface based on upcalls.

The generic runtime maintains for each object a link between the object and its class. This link is used to find the upcall information associated with each object.

The upcall information, and associated functions is used for a variety of purposes, including support for persistence, invocation and re-mapping between address spaces. In fact, any time where a functionality of the generic run-time needs access to information about objects that only the language specific environment will know.

For example to support clusters persistence, and hence object persistence, we need access to the layout of objects to locate references held in the objects data. When a cluster is mapped into an address space all the objects are scanned by using the appropriate upcall function to locate the internal references (to external objects) and performing a mapping from the domain wide references (used when an object is located on secondary storage) to address space specific references, this technique if often called pointer swizzeling.

Another example is for object invocation. Invocations between objects in the same cluster is based on the standard method invocation of the language (C++ method). Invocations between objects in different address space use the model offered by the COOL-base layer (CHORUS communication primitives). The proxy is used to trap the normal function invocation and replace it by an remote invocation which marshalls the parameters, issues an remote procedure call, and unmarshall the results. At the receiver, a dispatch procedure, which is part of the upcall function associated with an object, is used to call the appropriate method on the appropriate object.

Invocation may also use the underlying cluster management mechanisms to map clusters into local address spaces for efficiency reasons, or locally to allow light weight RPC and maintain protection boundaries, again the upcall functions are used to support this.

3.3. The Language Specific Run-Time

The language specific run-time maps a particular language object model to the generic run-time model. This may be achieved through the use of pre-processors to generate the correct stub code and the use of the upcall table.

† Subject to language level constraints.

As discussed above, the GRT will, in the process of operations such as mapping or unmapping an object from an address space, upcall into the language specific run time responsible for that object by using the upcall table associated with the object and generated by the language specific run-time. This requires that the language run-time, usually the compiler, generates enough information to interface to the generic run-time. Currently we use pre-processor techniques to generate this information so that at run time objects can be managed by the underlying COOL system.

4. The Base Level Revisited

In Section 3.1 we briefly outlined the abstractions that the base level provides, however the container/cluster mechanism is designed to support more than a simple grouping of objects.

Our goals when designing the base abstractions where:

- Support distributed, shared virtual memory so that we could efficiently support languages based on virtual memory references.
- Provide a form of memory persistence including the mechanisms for a single level store so that higher levels would not see a multi-tiered storage hierarchy.
- Provide a means to structure the distributed virtual memory space so that system builders can control their use of the distributed virtual memory.

The mechanisms that form part of individual language run-times and the GRT support distributed programming, however, in all cases they are costly. Object relocation requires pointer swizzling when clusters are mapped and unmapped; invocation using a message passing mechanisms need parameter marshalling and often break the semantics of object invocation.

Supporting a distributed, shared virtual memory is one solution that allows efficient transparent programming within a distributed environment. Although there are many costs and restrictions to a distributed virtual memory model, when combined with a complete system that supports other mechanism, such as mapping and remote invocation, it offers a powerful tool. A key difference between our work and others is that we offer a range of mechanisms to support distribution, not just one.

Our basic unit of distribution at the base level is the container, which, as described in Section 3.1, is made up of several clusters.

Containers are lazily mapped from secondary storage, by the base level into a virtual address space, or context. This mapping my involve relocation, as the form held on secondary storage may store pointers in a global format.†

Each container is ultimately mapped to one or more CHORUS segments, the unit of secondary storage. When mapped, a container is said to have a *view.* The view represents this mapping from secondary storage segments, to primary storage regions.. More than one view may be managed by a context at one time, allowing multiple containers to be

† A vanilla language mapped onto the GRT, without language run time support will not be able to support relocation and will be constrained to always be located at a particular set of addresses. An extended language, that was designed to exploit the GRT would allow addresses to be relocated, thus allowing the system to relocate containers as required.

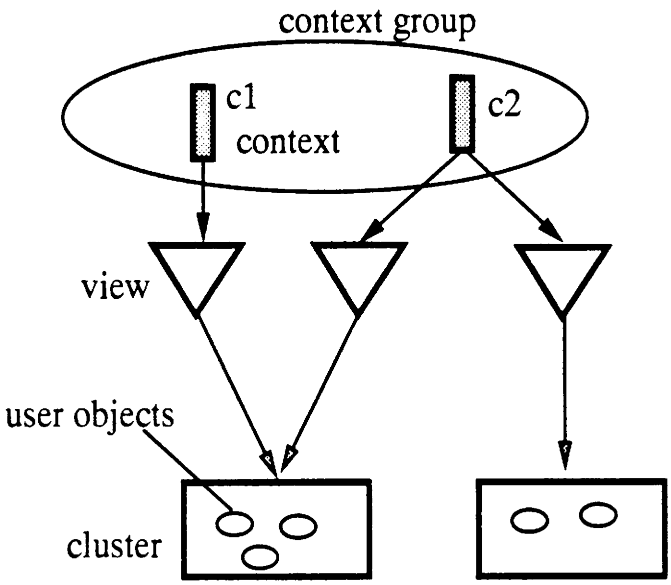


Figure 2: *COOL-base fragmented objects*

mapped into a single context; see Figure 2. The management of distributed views of a container is carried out by the base level.

A container, once created will remain in the system as long as there are references to that container. This persistence is managed by the base level.

Once mapped, objects within the container can carry out invocation using virtual memory references. If that activity wishes to diffuse to other sites, for example to allow physical parallel activity, then we create a context group. A context group is a set of contexts that support one or more containers. Each container is mapped at exactly the same set of addresses in each context in the group.

It is possible, and indeed likely, that a context will support more than one container at a time. Hence, contexts may belong to multiple groups at any one time with parts of their address space "allocated" to different groups. A group of contexts that map a particular container as said to support a Persistent Context Space, a distributed, persistent address space from the containers point of view.

The management of these Persistent Context Spaces requires some form of distributed control. There are several aspects to this. Containers which wish to diffuse to new contexts need to know if that context is capable of supporting the container, e.g. if addresses used by the container are already allocated then the diffusion can not be carried out.† When new virtual memory is added to a container, then allocation must be carried out across all containers in the group, this is managed by the distributed view control mechanisms.

4.1. Implementation Structure

To manage these distributed entities, the COOL-base is composed of several objects, or fragments‡ represented on each site and each using

† It may be possible to remap the contain... to a new set of addresses compatible with the new context.

‡ We use the term fragment, because each local representative, is a part of a global distributed, or *fragmented*, object.

the underlying CHORUS mechanisms to implement a distributed algorithm. These objects are grouped into three major components:

- Base Object: it contains all state informations about the local site;
- Base Proxy: transparently addresses the correct base object whenever a request for some system action must be re-directed to another site;
- Base Server: transparently forwards incoming remote requests to the local fragment responsible for managing that resource.

Each COOL-base server implements three protocols (with one CHORUS thread per protocol):

- Distributed group management: creates and deletes *groups,* attaches and detaches *contexts* from groups and controls address space allocation;
- Distributed view management: attaches and detaches *views* to and from *clusters* and informs the COOL-base to raise an upcall whenever these operations can influence the use of the data stored inside the *cluster*;
- Distributed cluster management creates, deletes, activates and de-activates *clusters*; it is also responsible for adding and deleting segments to/from *clusters*.

The protocols are mapped directly on to CHORUS IPC. Some operations have to upcall the generic run time to update upper layer state information. This also uses the CHORUS IPC, allowing us to upcall both locally and remotely.

4.2. Persistency Support

Persistent memory is organized in *containers* as explained above. Each container is further subdivided in *clusters;* a cluster being a set of persistent segments.

Each entity managed by the system layer is named using a CHORUS capability, which uniquely names it in the distributed system. Capabili-

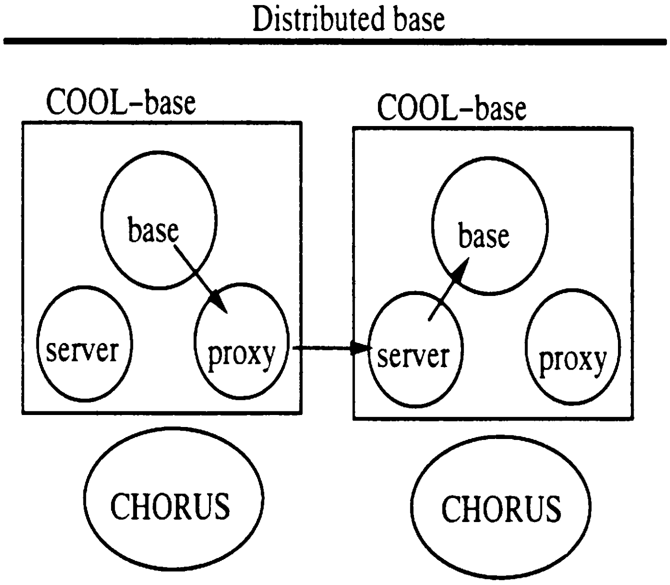


Figure 3: *COOL-base fragmented objects*

ties are the means to manage system entities and are passed between servers. In the case of clusters and containers; both virtual memory based entities, capabilities are managed by *mappers*. Each mapper is designed to manage the relationship between secondary storage and main store. When a request to use a cluster is generated, the COOL base system hands of the request for an unmapped cluster to the mapper managing that cluster. The mapper is responsible for locating the secondary storage representation of the cluster, and will understand enough of this format to allow it to map the cluster into primary store.

Persistency of clusters is also managed by the mapper. In conjunction with higher level tools such as garbage collectors, the mappers decide which clusters are referenced and will always ensure that such clusters are mapped out into secondary store where they will remain until referenced again.

The capability assigned to the cluster, and managed by the mapper is guaranteed to remain unique during the lifetime of the system. This guarantee is made by the underlying CHORUS micro-kernel which is responsible for generating capabilities.

4.3. Cluster Mapping

An application starts to run within a single context. Initially a single container will be associated with this application, with a minimum of one cluster mapped into the context.†

An exception mechanism exist that uses memory faults to map the correct memory at the right place. The first container mapping can be considered the highest level fault. It makes visible the next level in the structure, that is, clusters.

The second exception level is the segment fault: upon an access to some unmapped memory address, the exception handler verifies if there is some existing cluster, part of the current complete memory space, that contains a segment with the needed address when mapped. It then maps the right cluster. Finally, in a page-based architecture, each segment is divided in pages, so it is only effectively read to in-core memory if it is really accessed (in a third level fault).

After mapping, memory may need to be relocated. This is dependent on the semantics of memory contents and only application levels are aware of it. The relocation itself is based on symbolic information and only known symbols can be relocated. The problem is that there may be pointers that have no correspondent symbols generated normally by the compilation chain, so, special high level run-time code has to exist in order to access intrinsic semantic information of memory contents at user-level in a transparent manner. The base level, causes the run-time level to carry out any relocation required when the cluster is mapped into a context for the first time.

4.4. Cluster Unmapping

The upcall mechanism can also be considered an exception (but a distributed one). As we saw, cluster mapping and relocation is done automatically by the base and the run-time system when a memory fault occurs. In the simple case, mapping is carried out from secondary storage on an inactive cluster. However, it is likely that a cluster is already in use as part of another persistent context space. Hence it is necessary

† Remember that a container is made up of one or more clusters. Clusters are the unit of mapping.

to force that cluster, and its container, to be unmapped from one persistent context into the faulting one.

An upcall has to be issued from the kernel to the run-time system in order to unmap the cluster transparently from the application contexts. This upcall is performed using the CHORUS communication mechanisms allowing the upcall to work in the distributed system.

4.5. Mutual Exclusion

With the exception and upcall mechanisms in place it is straight forward to assure mutual exclusion of clusters that need to be mapped at different addresses, i.e., that belong to different active persistent contexts spaces.

During memory fault handling, if the system sees that a cluster is being used by another persistent context space it upcalls all contexts in that context space to force the unmapping, and proceeds. Later on, if any one of the other contexts needs that cluster again, it will do exactly the same in the inverted sense.

After remapping a cluster, the system has to verify if the container information about that cluster is still valid. The container may have a different set of clusters or belong to another persistent context. This has to be done immediately after cluster mapping because it may now directly reference another cluster after being changed in the persistent context space where it was previously mapped.

4.6. Shared Memory Coherency

Memory mapped in a context space needs to be assured single-writer multiple reader coherence between all distributed contexts that have it mapped into its address space. A distributed shared memory system such as proposed by Li [App91a] is used. This is a strict coherency algorithm but is well suited to the semantics of languages such as C++. We are currently investigating weak coherency support.

5. Conclusion and Current Status

The CHORUS micro-kernel is a set of low level functionality on which higher level systems can be built. After four years of experience using it to build object oriented operating systems we are convinced that micro-kernels are a sensible approach to reduce system complexity and the development cycle.

The COOL project is building an object oriented kernel above the CHORUS micro-kernel. Its aims are to provide a generic set of abstractions that will better support the current and future object oriented languages, operating systems and applications.

Our experience showed that much of the work in implementing a distributed system goes into the maintenance of distributed state. We used an object-based system to describe distributed state with fragmented objects. The use of the CHORUS micro-kernel allowed the implementation of these fragmented objects in a natural manner using a set of protocols over CHORUS IPC based on a distributed capability-based naming scheme that CHORUS supports.

We currently have a limited COOL platform running above the CHORUS micro-kernel, running native on networked 386 based machine. This platform implements the basic cluster level including the distributed virtual memory support. The COOL GRT offers full support for object distribution and for persistence. In addition we have built a pre-processor environment that allows us to generate pre-processor tools that can be used to extend existing languages such as C++ to take full advantage of the COOL v2 operating system interface.

Acknowledgments

We would like to thank our colleague at CHORUS systems for their valuable input to this work, in particular, Peter Strarup Jensen and Adam Mirowski.

References

[Ama92a] Paulo Amaral, Rodger Lea, and Christian Jacquemot, "A model for persistent shared memory addressing in distributed systems," in *Proceedings of the International Workshop on on object orientation in operating systems*, IEEE Computer Society, Dourdon, France (September 1992).

[App91a] Andrew W. Appel and Kai Li, "Virtual Memory Primitives for User Programs," pp. 96-107 in *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA USA (April 1991).

[Bla92a] Gordon Blair and Rodger Lea, "The impact of distribution on the object-oriented approach to software development," *IEE Software Engineering Journal* 7(2) (March 1992).

[Cah91a] Vinny Cahill, Chris Horn, Gradamir Starovic, Rodger Lea, and Pedro Sousa, "Supporting Object Oriented Languages on the Comandos Platform," in *Proceedings of ESPRIT'91 Conference*, Brussels, Belgium (November 1991).

[Des89a] Jean-Marc Deshayes, Vadim Abrossimov, and Rodger Lea, "The CIDRE Distributed Object System Based on CHORUS," pp. 8 in *Proceedings of TOOLS'89* (1989).

[Hab90a] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov, "COOL: Kernel support for object-oriented environments," *SIGPLAN Notices* **25**, pp. 269-277 (1990).

[Lea91a] Rodger Lea and James Weightman, "Supporting object oriented languages in a distributed environment: The COOL approach," pp. 37-47 in *Proceedings of the 5th TOOLS Conference*, Prentice Hall publishing, Santa Barbara, USA (October 1991).

[Lea92a] Rodger Lea and Christian Jacquemot, "The COOL architecture and abstractions for object oriented distributed operating systems," in *Proceedings of the 5th ACM European SIGOPS*, Mont Saint-Michel, France (September 1992).

[Sha86a] Marc Shapiro, "Structure and Encapsulation in Distributed Systems: the Proxy Principle," in *Proceedings of the 6th ICDS conference* (May 1986).

Programming Distributed Applications Transparently in C++: Myth or Reality?

Graham D. Parrington

Computing Laboratory
University of Newcastle upon Tyne
England
Graham.Parrington@newcastle.ac.uk

Abstract

Modern computing and networking hardware make the physical interconnection of many machines simple. However, programming an application to take even limited advantage of the interconnection is notoriously difficult due to the complexity of the protocols involved. Furthermore, real world demands *insist* that such applications need to be programmed in an existing, preferably widely available, language. One approach aimed at easing this difficulty is based upon the concept of *transparency*. By making the underlying distribution of the system transparent to the programmer it is hoped that the programming task becomes comparable with that of programming centralised applications. This paper describes mechanisms and tools that enable the various facets of transparency can be accomplished for the language C++ noting what level of transparency can be realistically attained.

1. Introduction

The programming of any large application is notoriously complex, requiring discipline on the part of both designer and programmer. Object-oriented design and programming techniques show great potential for alleviating many of the problems due to the inherent modularisation and encapsulation properties these techniques possess. Currently, most *widely* available object-oriented languages and systems (that is, available in the *commercial* arena) have little or no support for the programming of distributed applications despite the ease with which the computer hardware can be connected to construct a physically distributed system. Although much notable research effort has been concentrated upon ways of hiding, to greater or lesser degrees, the underlying distribution of the system, this has generally been achieved by creating entirely new distributed programming languages or systems (for example: *Emerald* [Bla87a], *Clouds* [Das85a], *Avalon* [Det88a], *Argus* [Lis88a], and *Camelot* [Spe88a]).

The primary reason for this focus is that existing languages have been developed without any consideration of the problems introduced by distribution and thus contain one or more features that are either impossible, or impractical, to distribute for a variety of reasons. A classic example of such a feature is the assumption that the application will execute within a single address space.

However, for distributed computing to truly gain importance to a wider audience does require the ability to program distributed applications in *existing* languages – preferably with a *minimum* of additional effort. Thus, although total transparency is typically impossible to achieve, partial transparency is both achievable and highly desirable. Computing platforms supporting this philosophy are starting to emerge (for example: Integrated Systems Architecture (ISA) [APM91a], Open Network Computing (ONC) [Sun88a], the Open Software Foundation Distributed Computing Environment (OSF/DCE) [OSF91a], and the Object Management Group Common Object Request Broker Architecture (CORBA) [OMG91a]) from both manufacturers and international standards bodies. While generally welcomed by users from a wide background (as repeatedly stated during a workshop at the ESPRIT Conference in 1991 [Par91a]) these platforms are not without their flaws, in particular they still typically do not provide much transparency to the applications programmer who is forced to write substantially different code to that written for a normal non-distributed application.

With this philosophy in mind, this paper examines the extent to which transparently distributed applications can be written in C++ [Str86a] (an increasingly popular commercially available object-oriented language) using only features available in the language itself, in combination with some other auxiliary tools. Furthermore, the implementation is aimed to be as portable as possible so as to promote the maximum potential reusability in a real environment.

The driving force behind this work is the *Arjuna* [Shr91a] project at the University of Newcastle upon Tyne. *Arjuna* is an object-oriented programming system for the construction of reliable distributed applications, which provides flexible and integrated mechanisms for the management of concurrency, recovery, naming, and persistence of C++ objects. These facilities are implemented solely using the inheritance capabilities of C++ thus gaining considerable flexibility. One basic facility required by *Arjuna* is the ability to transparently distribute C++ applications. This paper principally concentrates on the general purpose tool developed as part of the project to address this problem. The remainder of the paper concentrates on several of the other separate facets of distribution transparency and examines whether they can be achieved within a C++ application.

2. Distributing Applications Transparently

The term distribution transparency encompasses many varied attributes including:

- *Location.* The current location of an object is hidden from all other objects, enabling it to be located anywhere within the distributed system.
- *Access.* The syntax and the semantics of the invocation of operations upon objects is identical for both local and remote objects.
- *Failure.* The effects of any partially completed operation invocations are hidden in the event of failure.

- *Concurrency.* The existence of concurrent users of an object are hidden. That is, the effects produced by any concurrent use of an object are not observable.
- *Replication.* The existence of multiple copies (for availability purposes, for example) of an object is hidden.
- *Migration.* This is a dynamic form of location transparency. If an object moves from one location to another while in use, any users is unaware of the migration.

Furthermore, an application can be transparent in different ways to different users. For example, an application that is fully transparent to an end user may not be so to the application programmer. Most of the platforms mentioned earlier allow the creation of applications that are transparent to the end user. Their level of transparency to the programmer, however, is typically quite limited as will be outlined briefly in later sections.

Due to the encapsulation inherent in the object-oriented model, object-oriented systems should provide a natural framework in which the above transparency attributes can be expressed. For example, since object interaction is via well-defined interfaces it should be irrelevant where the actual objects reside, both to the user and the programmer; all that should be required is a means by which operation invocations are delivered to the correct object regardless of its current location.

2.1. RPC Systems and Stub Generators

A popular technique employed in the construction of distributed applications is based upon the concepts of *Remote Procedure Call* (RPC) [Bir84a, Ber87a] and *Stub Generation* [Jon85a, Gib87a, Sun88b]. Conceptually, a distributed application consists of several fragments split between the client (caller) and the server (callee). These fragments are: the client, the client stubs, RPC transport, the server stubs, and the server. Both the client and server are typically designed and implemented as if the application was to execute in a traditional centralised environment. It is the function of the client and server stubs to hide the underlying distribution to as great a degree as possible. Since production of these stubs can be tedious and complicated the process can be automated through the use of a *Stub Generator.* This parses a description of the interface between the client and the server, written in some *Interface Definition Language,* and produces the required stub code in a language compatible with both. In many systems this interface description language has a different syntax and semantics to the language in which the application is programmed. This requires that the programmer map the original interface description from the host language to the IDL before the stub generator can operate. Furthermore, many platforms require the programmer to write the server routines in a special way, by either explicitly massaging the names of the server routines (for example, by appending some number that represents a unique code for that routine), or by the inclusion of extra arguments to each routine, or even a combination of both.

3. Distributing C++ Applications

As outlined above, distribution transparency encompasses several distinct attributes. The following sections describe to what extent an ordinary C++ application can be made to exhibit the required transparency attributes.

3.1. Location and Access Transparency

Location transparency effectively requires that the name of an object as known by the user or programmer does not reveal any information about its true location. Exhibiting access transparency requires that all operation invocations performed by a client application upon an object, such as creation, destruction and normal operation invocation, can be suitably caught and redirected to the actual instance of the object on whatever node the object actually resides upon. This access transparency can be achieved simply in C++ by replacing the programmer's class declarations with new class declarations such that these new classes have the same *external* interface as the original but with new *implementations* of all of the operations (as RPC's, for example). This approach should produce full access transparency to the programmer (and, by default, the end user) providing that only the publically available operations of an object are used.

Arjuna uses stub generation techniques to achieve just these effects. To further enhance programmer transparency, the *Arjuna* stub generator does not have a separate interface definition language, instead its input is the original C++ class header files that would normally have been processed by the standard C++ compiler. Since typical C++ header files both contain things that are a hindrance to this technique (inline function definitions, macro definitions, etc.), and may also fail to provide sufficient information in some cases (particularly with respect to pointers), some programmer assistance may be required. Therefore, the operation of the stub generator can be controlled in two ways. Firstly, through flags passed on the command line, and secondly, through stub generation specific commands inserted into the actual header files themselves. In order that these header files remain acceptable to a standard compiler these commands are hidden inside comments that precede the syntactic entities to which they apply (in terms of the C++ grammar they are classed as *declaration-specifiers* just like storage class specifiers, for example).

3.1.1. Client and Server Classes

For any class definition presented as input, the *Arjuna* stub generator produces two additional classes that represent

- The replacement class for use by the programmer in the client program
- The server stub class responsible for decoding an RPC request, unmarshalling any incoming arguments, invoking the correct operation on the real class, and marshalling any output arguments and any returned value before returning to the caller.

For example, the class definition shown below in Program 1, which represents a simple interface to a distributed diary system, when processed by the *Arjuna* stub generator would cause the generation of the client class shown in Program 2. Simple renaming tricks played using the standard preprocessor enable this class to be transparently used under its original name in the programmer's application code. The generated client and server stub code is independent (deliberately) of the particulars of the underlying RPC mechanism since it is accessed through a class interface (implemented by the class `RpcControl`) which has a proscribed minimum set of operations. This class can

```
#include "AppointMent.h"

// The following stub specific commands are actually the default
// @Remote, @NoMarshall
class Diary : public LockManager
{
public:
    Diary(ArjunaName AN);
    ~Diary();

    String WhereIs(time_t now, String user);

    AnAppointment GetNextAppointment(time_t now);
    int AddAppointment(AnAppointment entry);
    int DelAppointment(time_t when);

    virtual Boolean save_state(ObjectState&, ObjectType);
    virtual Boolean restore_state(ObjectState&, ObjectType);
    virtual const TypeName type() const;

private:
    String      user_name;
    Appointment *appts;
};
```

Program 1: *Sample input class*

implement these operations in any manner applicable to the actual RPC transport mechanism being used. However, since the actual implementation of the RPC classes is irrelevant to the generated output it will not be described further.

This generated client stub class has the same set of *public operations* as the original (although any constructors have had an extra argument added to them, this is effectively invisible and the code written to use instances of the original class will still compile). Public instance variables, however, are deliberately not included in the generated class for reasons that will be explained in a later sub-section. Internally the implementation of the class is totally different. Firstly, only variables pertinent to the establishment and maintenance of the RPC connection

```
class RemoteDiary : public RemoteLockManager
{
public:
    RemoteDiary (ArjunaName , RpcCcontrol *crpc = 0);
    ~RemoteDiary ();

    String WhereIs (time_t , String );
    AnAppointment GetNextAppointment (time_t );
    int AddAppointment (AnAppointment );
    int DelAppointment (time_t );
    virtual Boolean save_state (ObjectState & , ObjectType );
    virtual Boolean restore_state (ObjectState & , ObjectType );
    virtual const TypeName type () const ;

protected:
    RemoteDiary(RpcControl *, const RpcBuffer&, char);

private:
    virtual RpcControl *_get_handle () const;

    RpcControl *_client_handle;
    ...
};
```

Program 2: *Generated client class*

are present. Secondly, all of the operations are reimplemented to perform the appropriate argument (un)marshalling and RPC invocation. Thirdly, some additional operations are introduced including an additional protected constructor which is used to ensure that certain information pertinent to the RPC system is correctly propagated to the stub generated versions of all base classes (if any).

Similarly, the generated server class (Program 3) has operations that primarily correspond to those of the original `Diary` class except that each is responsible for parameter (un)marshalling and calling the equivalent operation on the real object. In addition this server class has operations for server initialisation and two operations that implement the code that determines from the incoming call which server operation to actually call (the so-called operation dispatch code).

Each routine in the server class effectively has the same set of arguments. The first is a pointer to the object to be manipulated which is passed to ensure that the semantics of multiple inheritance are obeyed. The second is an `RpcBuffer` that contains all of the call information (incoming parameters, for example), and the third is an `RpcBuffer` into which the results (if any) can be placed. All operation names in this class are generated by combining the original name with a hash value computed from the original full operation signature (class name, operation name, and types of all parameters). This scheme ensures that operations overloaded in the original class can be correctly resolved in the server (otherwise the standard overloading mechanism would not be able to tell them apart). This computed hash value is also used in the server dispatch code when determining which operation in the server to actually call.

```
class ServerDiary : public ServerLockManager
{

public:
    ServerDiary ();
    ~ServerDiary ();

    void Server (int, char **);
    long DispatchToClass (LocalDiary*, long, RpcBuffer&, RpcBuffer&);

private:
    // Main server dispatch operation
    long DispatchToOper (LocalDiary *, long, RpcBuffer&, RpcBuffer&);

    // Operations corresponding to those callable in the client
    long Diary164312325(LocalDiary *, RpcBuffer&, RpcBuffer&);
    long Diary262355078(LocalDiary *, RpcBuffer&, RpcBuffer&);
    long WhereIs186673735(LocalDiary *, RpcBuffer&, RpcBuffer&);
    long GetNextAppointment31096804(LocalDiary *, RpcBuffer&, RpcBuffer&);
    long AddAppointment101964452(LocalDiary *, RpcBuffer&, RpcBuffer&);
    long DelAppointment222961300(LocalDiary *, RpcBuffer&, RpcBuffer&);
    long save_state140478901(LocalDiary *, RpcBuffer&, RpcBuffer&);
    long restore_state9807781(LocalDiary *, RpcBuffer&, RpcBuffer&);
    long type117319830(LocalDiary *, RpcBuffer&, RpcBuffer&);

    // Pointer to real object
    LocalDiary *therealobject;

};
```

Program 3: *Generated server class*

The stub generator preserves the encapsulation properties of the input classes in its output classes. That is, the server dispatch code (implemented by the generated routine `ServerDiary::DispatchToOper` in this example) will only directly invoke the operations of the `Diary` class – not any operations from any class from which `Diary` might have been derived (for example, `LockManager`). If an operation inherited from some base class needs to be invoked the request is passed to the appropriate base class by the routine `ServerDiary::DispatchToClass` (in this example). This ensures that the stub code for each class can be compiled independently from any of its parents and that a change in a base class need not necessarily force a recompilation of the stub code for any derived class.

The `DispatchToClass` routine is also responsible for resolving the potential ambiguities on which routine to call in the server that can arise when multiple inheritance is used (the ambiguity cannot exist in the client otherwise the compiler would have rejected the code). It does this using information built in the client when the client object was constructed and which is transmitted as part of the call.

The client stub code produced exploits the C++ constructor and destructor notions to ensure that the real (user) objects in the server have lifetimes that match the lifetime of the (stub) objects in the client. At the point that the stub object enters scope in the client (and thus the constructor operation of the object is automatically executed) then binding of client to server is accomplished using the supplied `ArjunaName` (this is part of the standard *Arjuna* naming scheme for persistent objects and the mechanism via which location transparency is achieved). Furthermore, the first RPC sent to the newly created server, corresponds to the invocation of the constructor for the real object and is passed the arguments presented by the client application. Similarly, when the stub object is destroyed in the client, the generated destructor causes an RPC request to be sent to the server causing the execution of the remote object destructor before the server is itself destroyed.

3.1.2. Parameter Marshalling

C++ operator overloading is used to simplify considerably the code required to marshall (encode) and unmarshall (decode) arguments to and from the underlying RPC buffers. In particular, the operators >> and << have been adopted for this purpose (similar to their use in the C++ I/O system). Thus << is used to marshall arguments into the buffers used by the RPC mechanism, and >> to unmarshall arguments from the buffers regardless of the actual type of the argument. The RPC buffer class (`RpcBuffer`) provides a set of operations that permit the marshalling and unmarshalling of all of the basic types of C++ (int, char, etc.). The marshalling of more complex structures is simply achieved by breaking the structure up into its component parts and marshalling each independently. The actual encoding scheme used is the same as that used by the persistence mechanisms that enable a C++ object to be stored on disk.

Since all C++ objects are treated as encapsulated entities, the stub generator ensures that suitable definitions exist for these marshalling operators for all objects passed as arguments – even class objects which must have their public operation set augmented by the inclusion of the operations for (un)marshalling. Thus in the above example the marshalling code generated for the class `AnAppointment` is similar to that shown in Program 4.

The generated client stub code for each operation follows a standard pattern: marshall arguments, send invocation, await reply, unmarshall results, and return to caller. This pattern is illustrated in Program 5, which has the corresponding server code of Program 6.

Arguments passed by pointer or reference require special handling. By default these are treated as in/out parameters and are both sent in the call and assumed to be returned as part of the result. This behaviour can be modified in two ways. Firstly, if the argument is declared to be *const* then it is automatically treated as input only. Secondly, the programmer can augment the declaration of an argument with stub generation specific commands (`@In`, `@Out` and `@InOut`) to guide the process explicitly.

So that complicated data structures (such as lists and trees) can be (un)marshalled automatically the actual routines in `RpcBuffer` that do the real work of encoding the data attempt to keep track of whether a parameter has been packed into the buffer already, in which case it is not packed again. Instead a special flag is inserted that the unpacking routines can recognise and can thus compensate appropriately. This helps to ensure that arguments only get encoded and decoded once. In addition, the programmer can suppress the automatic generation of marshalling code and provide alternative implementations if required

```
class AnAppointment : public StateManager
{
public:
    AnAppointment ();
    ~AnAppointMent ();

    // Ignore other operations here for clarity
    ...
    // These are the added marshalling operations

    virtual void marshall (RpcBuffer&);
    virtual void unmarshall (RpcBuffer&);

private:
    time_t start;
    time_t end;
    String description;
    Boolean confirmed;
};

// Overload << to marshall instance into buffer
inline RpcBuffer& operator<< ( RpcBuffer& rpcbuff, AnAppointment topack)
{
    topack.marshall(rpcbuff);
    return rpcbuff;
}

// Marshall each variable in turn
void AnAppointment::marshall ( RpcBuffer& rpc_buff )
{
    rpc_buff << start;
    rpc_buff << end;
    rpc_buff << description;
    rpc_buff << confirmed;

}

// Unmarshalling operations are similar only using >>
    ...
```

Program 4: *Sample marshalling code*

```
AnAppointment RemoteDiary::GetNextAppointment (time_t now)
{
    RpcBuffer cbuffer, rbuffer;     /* call and return buffers */
    RPC_Status rpc_status = OPER_UNKNOWN;
    long server_status = 0;

    class AnAppointment returned_value;

    /* marshall parameter */
    cbuffer << now;

    /* do call */
    rpc_status = _client_handle->Call(31096804, cbuffer,
                                      server_status, rbuffer);

    if (rpc_status == OPER_DONE && server_status == 0)
    {
        /* unpack result */
        rbuffer >> returned_value;
    }
    else
        rpc_abort();

    return (returned_value);
}
```

Program 5: *Sample generated client code*

through a command (`@UserMarshall`) embedded in the header file describing the class.

3.1.3. Some Potential Problems

Stub generation is not without its problems caused primarily by the lack of a shared address space between the client and the server. For example, the semantics of procedure call may be different (stub generation usually utilises a copy-in, copy-out process for arguments which may have a different effect upon application execution). Furthermore, certain types of parameters may be disallowed altogether (procedure type parameters, for example). Such problems are not particular to the stub generation system described here but are inherent in the stub generation process and affect all conventional languages distributed this way. Additionally, since C++ was not designed for distributed programming some of its constructs are not amenable to stub generation techniques and have to be disallowed. Examples of such constructs include:

```
long
ServerDiary::GetNextAppointment31096804
    (Diary *theobject, RpcBuffer& work, RpcBuffer& result)
{
    /* unpack incoming argument */
    time_t now = 0;
    work >> now;

    /* perform the real call */
    AnAppointment returned_value = theobject->GetNextAppointment(now);

    /* send back result */
    result << returned_value;
    return OPER_DONE;
}
```

Program 6: *Sample generated server code*

- Variable length argument lists. These cannot be marshalled automatically since the stub generator cannot determine at the time it processes the header file how many arguments will need to be marshalled on any given call.
- Public variables and friends. These break the assumed encapsulation model and allow potentially unconstrained access to the internal state of an object. Since that object may now be remote from the client application such variables will typically not exist or at least not be accessible in the same address space.
- Static class members. C++ semantics state that only a single copy of a static class variable exists regardless of the number of instances of the class in existence. These semantics cannot be enforced in a distributed environment since there is no obvious location to site the single instance, nor any way to provide access to it.

All of these problems have the effect of lowering the overall access transparency to the programmer, however, and this is the important gain, not completely to zero. With care applications can be written that are fully location and access transparent, while others require only minimal additional programmer assistance. However, the point remains that stub generation relieves the programmer of a significant proportion of the burden involved in the distribution of applications.

3.2. Failure Transparency

There are many potential sources of failure even in a conventional non-distributed system. Distribution only complicates matters by adding communication systems failures and the possibility that only parts of the system fail while other parts continue working. Handling failure transparently essentially requires a means by which operation invocation appears to be atomic. *Arjuna*, uses the notion of (nested) atomic actions [Gra78a] familiar to the database community for this purpose. Atomic actions have the useful properties of:

- *Failure Atomicity*. All of the operations that comprise the action complete successfully or none of them do.
- *Serialisability*. The concurrent execution of actions is equivalent to some unspecified serial order of execution.
- *Permanence of Effect*. Once completed, any new system states produced by the atomic actions are not lost.

Programmer defined groups of operations can be executed under the control of an atomic action and the *Arjuna* system ensures that the effect is as if they all complete or none of them do. Programmers must currently declare and use atomic actions themselves (by explicitly declaring instances of the class `AtomicAction` and invoking the `Begin`, `End` or `Abort` operations it provides). However, the stub generator could also make each remote operation execute under the control of an (potentially nested) atomic action if required which would result in a system that had similar semantics to *Argus*.

A simple implementation of the `GetNextAppointment` operation for the `Diary` class might then be as shown in Program 7.

Supporting atomic actions requires that objects be *recoverable* so that a prior state of an object can be (re)established in the case of failure. Using currently available C++ compilers, the *Arjuna* classes implementing recovery cannot determine the structure of an object at runtime without programmer assistance (although techniques akin to stub gen-

```
AnAppointment Diary::GetNextAppointment (time_t now)
{
    AtomicAction A;
    AnAppointment retV;

    // Start action
    A.Begin();
    // Set an appropriate lock
    if (setlock(new Lock(READ), RETRIES) == GRANTED)
    {
        // now perform the real operation on the list of appointments
        retV = appts->apptAtTime(now);

        // Success - commit action
        A.End()
    }
    else
    {
        // I failed - undo all my changes
        A.Abort();
    }
    return retV;
}
```

Program 7: *Example application code*

eration could reduce the amount of assistance required). Hence, *Arjuna* requires that the programmer provides implementations of the operations used for recovery (called `save_state`, and `restore_state`) explicitly, although the *Arjuna* runtime system controls exactly when these are called. This requirement does appear to break the failure transparency attribute, however, since *Arjuna* utilises the same operations to provide object persistence, recovery essentially comes about for free.

The most likely additional causes of failure in a distributed system over those found in a non-distributed one will be caused by failure of the RPC system for some reason. RPC failure typically comes from two sources. Firstly the RPC itself fails for some reason (that is, the server does not respond to the client request for a variety of reasons including, crashed server machine or process, network partition, or server overload). Secondly, the RPC succeeds (in the sense that the call is delivered) but the server process rejects it as invalid. This latter case can be caused by mismatches between the client and server interfaces for example. In either case an exception is raised by the stub code (until the proposed C++ exception handling mechanism is available this is simulated using UNIX signals). This exception can be caught and handled by the programmer and in addition will be caught and processed by other *Arjuna* components to ensure orderly cleanup (that is, outstanding atomic actions are aborted, recoverable and persistent objects restored to earlier states, etc.).

3.3. Concurrency

In a centralised application, concurrency is likely to be limited unless the programmer makes explicit use of any system provided facilities or is accessing some potentially shared resource such as a file. In such cases programmers will typically have to use the facilities of the system to avoid any potential interference problems when multiple clients access any shared object concurrently.

In a distributed system, concurrent use of an object is much more likely to occur. Furthermore, such usage is likely to be unforeseen and potentially uncontrolled. Hence more heavyweight concurrency control requirements may have to be imposed. To provide strong consistency *Arjuna* uses the heavyweight concurrency control required by the serialisability property of atomic actions [Ber87b]. This concurrency controller is implemented by the class `LockManager` which provides sensible default behaviour while allowing the programmer to override it if deemed necessary [Par88a]. The programmer interface to the concurrency controller is via the `setlock` operation (as illustrated in the code fragment of Program 7). By default, the *Arjuna* runtime system enforces strict two-phase locking following a multiple reader, single writer policy on a per object basis. Lock release is under control of the system and requires no intervention by the programmer. This ensures that the two-phase property is correctly maintained. However, applying concurrency control (setting locks) to an objects is deemed to be the responsibility of the programmer of the class who should be in a position to know the concurrency requirements of each operation.

The introduction of the locking calls could also, however, be handled automatically by the stub generation system (providing that the default behaviour is acceptable) since it can determine what type of lock to apply (read or write) based upon the "constness" of each operation as determined by its declaration.

3.4. Replication and Migration

Support for transparent object replication requires that the replicas of an object behave as if they were a single logical object. Migration requires firstly that the access mechanisms detect that an object has moved to maintain access transparency, together with a mechanism by which the state of the object can be moved. This latter problem is akin to that encountered in passing objects as parameters, or to making them persistent, and can be solved using similar techniques.

To date, the publically released version of *Arjuna* contains no mechanisms to handle these transparency attributes. However, a design for several replication strategies exists [Lit92a] and an experimental implementation of one of those strategies is currently under test. This implementation allows the state of an object to be replicated on several machines and ensures that all replicas remain consistent and is furthermore has been implemented using precisely the same stub generation techniques described in the earlier sections of this paper. Support for object migration will naturally follow in due course.

4. Conclusions

Producing transparently distributed applications in C++ is possible but requires care since the language was not designed with distribution in mind. At Newcastle we have built a distributed diary system, a simple theatre reservation system, and a moderately sophisticated distributed bibliographic database system [Buz92a], all using the mechanisms described here. Certain C++ constructs do cause problems but can usually be avoided or programmed around. While these restrictions lower programmer transparency it is a compromise that works well with existing tools and provides a practical solution to the problem of distributing C++ applications. In addition, since C++ is rapidly becoming a "de-facto" standard object-oriented language, the approach adopted here provides a route for others to exploit the capabilities of distributed systems without too many of the potential headaches.

Finally, the capabilities of *Arjuna* and the stub generation system can be mixed with other existing C++ class libraries (of which many exist already). For example, several applications written at Newcastle use Interviews for their User Interface, *Arjuna* for Persistence and Recovery, and the Stub Generation system for distribution.

For those whose interest has been piqued and who wish to experiment further, the source code for *Arjuna*, together with some other papers and documentation, is available via anonymous FTP from *arjuna.newcastle.ac.uk* (128.240.150.1)

Acknowledgements

The *Arjuna* project has been and continues to be a team effort. Many colleagues have contributed to it and have improved this paper by their diligence. The work reported here has been supported in parts by grants from the UK Science and Engineering Research Council and ESPRIT project 2267 (Integrated Systems Architecture).

References

[APM91a] APM, *ANSA Reference Manual*, (Available from APM Ltd., Poseiden House, Cambridge, UK.), 1991.

[Ber87b] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley (1987).

[Ber87a] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz, "A Remote Procedure Call Facility for Interconnecting Heterogenous Computer Systems," *IEEE Transactions on Software Engineering* **SE-13**(8), pp. 880-894 (August 1987).

[Bir84a] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* **2**(1), pp. 39-59 (January 1984).

[Bla87a] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering* **SE-13**(1), pp. 65-76 (January 1987).

[Buz92a] L. E. Buzato and A. Calsavara, "Stabilis: A Case Study in Writing Fault-Tolerant Distributed Applications Using Persistent Objects," *Proceedings of the Fifth Int. Workshop on Persistent Objects*, San Miniato, Italy (September 1-4, 1992).

[Das85a] P. Dasgupta, R. J. LeBlanc, and E. Spafford, "The Clouds Project: Designing and Implementing a Fault Tolerant Distributed Operating System," Technical Report GIT-ICS-85/29, Georgia Institute of Technology (1985).

[Det88a] D. Detlefs, M. P. Herlihy, and J. M. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++," *IEEE Computer* **21**(12), pp. 57-69 (December 1988).

[Gib87a] P. B. Gibbons, "A Stub Generator for Multilanguage RPC in Heterogeneous Environments," *IEEE Transactions on Software Engineering* **SE-13**(1), pp. 77-87 (January 1987).

[Gra78a] J. N. Gray, "Notes on Data Base Operating Systems," pp. 393-481 in *Operating Systems: An Advanced Course*, ed. R. Bayer, R. M. Graham and G. Seegmueller, Springer (1978).

[Jon85a] M. B. Jones, R. F. Rashid, and M. R. Thompson, "Matchmaker: An Interface Specification Language for Distributed Processing," *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pp. 225-235 (January 1985).

[Lis88a] B. Liskov, "Distributed Programming in Argus," *Communications of the ACM* **31**(3), pp. 300-312 (March 1988).

[Lit92a] M. C. Little, "Object Replication in a Distributed System," PhD Thesis, Technical Report No. 376, University of Newcastle upon Tyne (February 1992).

[OMG91a] OMG, *The Common Object Request Broker: Architecture and Specification*, Object Management Group, Cambridge, Mass (December 1991).

[OSF91a] OSF, *OSF DCE V1.x Requirements,* OSF DCE Reliable Computing Group (1991).

[Par88a] G. D. Parrington and S.K. Shrivastava, "Implementing Concurrency Control for Robust Object-Oriented Systems," *Proceedings of the Second European Conference on Object-Oriented Programming, ECOOP88*, Oslo, Norway, pp. 233-249 (August 1988).

[Par91a] G. D. Parrington, "Report on the Workshop on Distributed Enterprise Computing," *Esprit Technical Conference*, Brussels (November 1991).

[Shr91a] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of Arjuna: A Programming System for Reliable Distributed Computing," *IEEE Software* **8**(1), pp. 63-73 (January 1991).

[Spe88a] A. Z. Spector, R. Pausch, and G. Bruell, "Camelot: A Flexible, Distributed Transaction Processing System," *Proceedings of CompCon 88*, pp. 432-439 (February 1988).

[Str86a] B. Stroustrup, *The C++ Programming Language,* Addison-Wesley (1986).

[Sun88b] Sun, "Rpcgen Programming Guide," in *Network Programming Guide*, Sun Microsystems Inc. (1988).

[Sun88a] Sun, "Network Services," in *Network Programming Guide*, Sun Microsystems Inc. (1988).

Microkernel Performance Evaluation using the JEWEL Distributed Measurement System

Martin Gergeleit
Frank Lange   Reinhold Kroeger

*German National Research Center
for Computer Science
Germany*
{ mfg | frank.lange | kroeger }@gmd.de

Abstract

The JEWEL performance evaluation tool, developed at GMD (German National Research Center for Computer Science), and its application in microkernel evaluation is presented. Some of the key-mechanisms of such systems have been observed with this tool. Since JEWEL is able to detect single events instead of simple mean values very detailed results are available.

1. Introduction

Performance evaluations of operating systems have always been of interest to the whole user community, because their results affect any application on top of a system. In the past, a lot of work has been carried out in the area of classical centralized monolithic operating system kernels. In the last years microkernel based operating systems appeared in the arena and became competitive in functionality to traditional operating systems [Ren89a, Ras88a, Roz88a]. A microkernel only offers the basic abstractions of memory, computing, and communication. It serves as a basis for constructing the system functionality out of several cooperating user-level servers. In comparison with monolithic operating systems microkernels promise a higher degree of flexibility and extensibility, better maintainability, and appropriate paradigms for distributed systems consisting of loosely coupled nodes and for other architectures, e.g. NUMA. Besides the necessity to assess the performance of the offered microkernel primitives, e.g. for interprocess communication, the use of microkernels raises additional questions concerning the performance penalties one has to pay for the advantages coming along with this change in operating system architecture. A lot of performance evaluation and tuning in the microkernel as well as in the servers stil, seems to be necessary in order to reach a comparable performance for providing standard system services.

In GMD's project RelaX a distributed measurement system called JEWEL ("Just a new evaluation tool") has been designed and implemented to evaluate the performance of distributed systems and applications and has been applied to these problems [Lan92a]. The fine grained observation facilities of JEWEL allow for an exact breakdown of the costs of the key mechanisms in microkernel operating systems. The measurements not just result in mean-value calculations of execution times. Since every single event can be observed, the measurements can provide minimum and maximum values, standard deviation and even empirical distributions for interesting performance measures reflecting detailed informations about caching effects, scheduling policies, or e.g. fairness related issues. This is in contrast to former approaches that normally consist of repeating interesting sequences of statements several thousand times, measuring the loop's total execution time, and computing the average execution time by dividing the measured execution time of the loop by the number of repetitions.

2. JEWEL – Overview

The overall goal of JEWEL is *flexibility* in order to avoid a limitation to specific application domains or specific classes of experiments. JEWEL has been designed to provide a flexible measurement system consisting of a set of generic components that can be reused without modifications and that have clear interfaces allowing for application-specific adaptations and extensions, to instantiate a measurement system that is then well-suited for the specific domain and the envisaged objectives. But JEWEL is also flexible with respect to the interface to the experimenter, which may be both, system developers or system managers. While the former probably will be more interested in performing off-line analysis, the latter often wishes to be able to do on-line monitoring. Graphical visualization of the system activity is very useful to this end because it eases the understanding and recognition of behavioural patterns.

The second important goal of the JEWEL measurement system is *preciseness* of the delivered quantitative results. For reasons of economy, the experimenter is interested in using adequate tools, i.e. the simplest or cheapest tools that ensure the accuracy of the measured values within certain bounds. Preciseness of results is very much dependent on the amount of interference between the measurement system and the system under test, on accurate timing facilities, and on the method of how data is extracted out of the system under test.

The *low-interference* property subsumes that the measurement process itself should have negligible or at least predictable influence on the measured quantities. The experimenter must have evidence that the observed behaviour reflects the real behaviour in the same non-instrumented system. The interference problem arises as soon as the observed system and the observing measurement system share common resources. The existence of a *global timebase* with sufficient resolution and accuracy is necessary to distinguish all relevant events, thus being able to deduct the correct global ordering of events, and to measure even time intervals, which are delineated by events on different nodes. Limitations concerning the quality of results also may be due to the inadequate extraction of measurement data. As argued above, performance measurement results are provided almost always as mean values for the interesting quantities, thus loosing important information and evidence of the measurements. The main reasons for this are that statistical sampling techniques for extracting measurement data (e.g.

the UNIX *profil*() system call) are applied instead of basing extraction on the occurrence of the relevant events in the system under test. The discussion above led to the identification of the following three tasks that have to be fulfilled by JEWEL. First, the measurement system must provide means to extract information from the system under test based on relevant events, and, because of the huge amounts of raw data that come to hand when monitoring computer systems, it must use data reduction techniques to make that data eligible to human users. Second, the collected and reduced data has to be presented to the user in a way that allows him to pickup the relevant information easily. Graphical presentation is very helpful here. Finally, an experimenter is interested in controlling the entire measurement system interactively, e.g. to change the focus of interest or the level of detail. While data collection has to be performed in a distributed fashion due to the distributed nature of the system under test, the experimenter is interested in observing and controlling the system from a central point.

Besides the *System under Test (SUT),* the JEWEL distributed measurement environment distinguishes the following three functional blocks: the *Data Collection and Reduction System (DCRS),* the *Graphical Presentation System (GPS),* and the *Experiment Control System (ECS).* Measurement data is extracted from the SUT, collected, filtered and processed by the DCRS, and then passed to the GPS for visualization to the experimenter. ECS supports an experimenter in managing his experiments efficiently by offering flexible mechanisms for experiment definition and maintenance and automated setup.

JEWEL supports the notions of *performance index, aspect* and *level (of detail)* to the experimenter for logically structuring the system under test from the measurement point of view. Performance indices are quantities that characterize the performance of the system under test. An aspect corresponds to a certain topic of interest inside the system under test (an aspect of an operating system may be, e.g., page usage or network communication) and is defined by a set of related performance indices. For further structuring of this set of performance indices, in order to limit the amount of information the experimenter has to perceive, different levels of detail can be used. The highest level of detail encompasses the entire set of performance indices for that aspect, each lower level defines a subset of the performance indices of the next higher level. During an experiment, the experimenter may select some aspects as currently being relevant at a certain level of detail, and it is the task of the measurement system to provide the results for the corresponding performance indices and to visualize them. The experimenter may specify a so-called *report mode* for each aspect which determines, whether the measurement system will present the results as soon as they are available, or whether it will buffer them. In the following subsections the design and implementation of the JEWEL subsystems are described to a certain level. More details are presented in [Lan92a].

2.1. Data Collection and Reduction System

Computing the performance indices is the general task of the DCRS subsystem. In general, this is a multi-stage process involving different kinds of DCRS components linked together to form a distributed measurement data processing network: *sensors* recognizing relevant events inside the system under test and extracting local measured quantities; *collectors* receiving these from sensors, perhaps residing on different machines, and combining them to provide global measured quantities; and *evaluators* computing performance indices based on both local and global measured quantities from sensors and collectors, respectively. Additionally, a fourth type of component called *mediators* allows for recording and replaying of measurement data into or from files. Mediators may be inserted at any stage of performance index computation. The information flow between DCRS components and the GPS is achieved by exchanging typed messages called *measurement data records (MDRs)* which have a generic structure suited for all measurement purposes.

The low-interference property requires that as many of the DCRS tasks as possible are done outside of the system under test. Therefore, the JEWEL measurement system is generally provided with its own physical resources attached to the system under test for filtering, collecting and evaluating of measurement data, and its own measurement LAN for exchanging MDRs. On the other hand most of the important events are only observable from inside the software with the knowledge of the internal logic of the observed code, so nearly each experiment has to have some influence on the measured quantities. JEWEL uses a hybrid approach to overcome these problems. Sensors are divided into a SUT-internal part and a SUT-external part which runs on resources dedicated solely to the measurement system. Event detection and measured value extraction are implemented by inserting so-called sensor statements (the internal part of the sensor) at appropriate locations in the code of the SUT. Sensor statements are really small pieces of code that writes an event record into a shared data structure, called event queue, where they are picked up by the SUT-external sensor part. If the instant when an event occurs is of interest, it has to be passed as an argument explicitly. After the event record has been placed in the event queue, the measurement data processing takes place concurrently to the operation of the SUT. This is done by the SUT-external part of the sensor.

The choice *where* to put the sensors depends heavily upon the constraints imposed by the actual system to be observed. In cases where the source code of the observed program is available, the most detailed and precise results will be achieved by instrumenting the program itself. Of course this requires knowledge of the logic of the observed code. This knowledge is typically given if JEWEL is used by a developer during software engineering. In cases where this approach seems to be too expensive or if only binaries are accessible there are three other, more generic possibilities for sensor placement with a decreasing level of observable details. If the code will be linked before execution, sensors may be placed inside the libraries. If the code is statically linked, it may be possible to modify the underlying operating system (e.g., each communication request or even every system call may execute at least one sensor). The last and most coarse grained possibility is to use separate observer processes, that report parts of the machine state via sensor macros. All these placement strategies may be used intermixed in a single JEWEL environment and they are independent of the actual JEWEL-adaptation.

Another question is, *how* to insert the macros. Currently, in JEWEL instrumentation of the SUT has to be done by hand. This requires knowledge of the program and causes additional coding work. On the other hand, such an instrumentation may be very problem-specific. A few sensors reporting the state known only in the context of the execution are often enough to give a very detailed view of the programs behaviour. A'ternatively sensor placement can be done automatically by the compiler or a precompiler (e.g., before and after each subroutine). This would cause no additional programming, but it leads to an unspecific instrumentation with a lot of useless, but time-consuming, sensor code. Such an instrumentation may be sensible to get a first, rough overview of what is going on, but it will reach its limits as soon as one wants to go into details.

2.2. Graphical Presentation System

The graphical presentation system (GPS) is the front end to the experimenter with respect to measurement data processing. The performance indices computed by the DCRS are usually sent to GPS for presentation to the experimenter. GPS supports the visualization of numerical values by providing a set of high-level graphical objects called *charts*. Based on X11 and OSF/Motif a completely graphical user interface to GPS has been designed and implemented. Each aspect corresponds to a so-called *view* and is graphically represented by a view window. In the same way that an aspect encompasses a number of performance indices, a view window consists of a number of charts, each of these displaying one or more performance indices of the corresponding aspect. Different levels of detail for an aspect result in different numbers and/or types of charts within the corresponding view. For presentation of performance indices GPS provides the following well-known chart types: pie chart, bar chart, curve, time series diagram, kiviatgraph, ganttdiagram and speedometer. The appearance of a view can be manipulated interactively according to the preferences of the experimenter. This process is called *customization* modifications made by an experimenter can be saved for reuse in following sessions. The GPS also has an integrated archiving mechanism that allows for recording and redisplaying of view snapshots.

2.3. Experiment Control System

The experiment control system (ECS) of JEWEL provides means for the experimenter to control the mode of operation of both the DCRS and the GPS. Like GPS, it provides a user interface based on the standards of X11 and OSF/Motif. According to the JEWEL architecture, ECS supports the notion of aspects and components. While aspects are abstractions used to logically structure the SUT from the measurement point of view, *components* are instances which collect, process, and present measurement data related to one or several aspects. ECS provides another abstraction, called an *experiment,* to refer to a real or planned configuration with respect to the entirety of defined aspects and components. Accordingly, the operations which are supported by ECS can be divided into three categories: operations on components, aspects, and experiments. With respect to the scope of ECS operations, an aspect can be viewed as affecting a set of components which support that aspect, while an experiment in turn can be viewed as providing a set of aspects. Thus, operations on experiments usually result in execution of a set of operations on aspects which in turn result in a set of operations on individual components.

2.4. Time Base

As pointed out before, an accurate global time base with high resolution has to be available for performance measurements in distributed systems. The solution to be applied depends on the granularity of the time intervals to be measured. If the granularity is coarse, a system clock providing a resolution of the order of 10 microseconds, combined with a software clock synchronisation protocol, may be sufficient, generally resulting in a total accuracy of the order of several tens of microseconds. An example of such a time base is the Digital DECdts time service (now part of OSF/DCE). A global time base with a resolution of the order of 10 microseconds requires, at least, measurement of time intervals of the order of seconds in order to ensure a reasonably low measurement error. However, for fine-grained measurements interesting distributed actions occur (e.g. reliable multicasts) which take even less than a single tick of such a clock. If one is interested in measuring actions with much shorter execution times, a fine-grained global time base becomes necessary.

For Jewel such a global time base has been designed and built in collaboration with GMD ASA project [Kle92a]. It consists of several high-resolution local clocks which are synchronised with each other by hardware. The first solution was based on clocks implemented as VME-bus boards which optionally can be synchronised with each other.

3. Applications

In the following, we will concentrate on two applications of JEWEL. Although main parts of JEWEL are generic and have been used in both applications without any modifications, the examples differ with respect to the implementation of the SUT-dependent part of the Data Collection and Reduction System. While one implementation uses a hybrid approach by deploying dedicated hardware to keep interference low, the other has been explicitly aimed at developing a pure software solution that does not require any special hardware. The first implementation of JEWEL was done in cooperation with the European Space Research and Technology Center (ESTEC), Noordwijk (NL), of the European Space Agency (ESA). This implementation is based on VMEbus-based hardware. To prove the viability of this approach, JEWEL was applied to assess the performance of the RPC mechanism of the Amoeba Distributed Operating System. Subsequently, an adaptation of JEWEL to the Mach 3.0 microkernel was done in collaboration with the Open Software Foundation (OSF) Research Institute at Grenoble (F). It was used to analyse the UNIX system call emulation of the OSF/1 server on top of Mach 3.0.

	Server Local			Server Remote		
Bytes	0	8192	30000	0	8192	30000
no. of RPCs	99761	100000	100000	100000	10000	10000
min	0.631	2.182	6.013	1.109	12.854	43.670
mean	0.661	2.217	6.046	1.126	12.891	43.775
max	2.662	3.645	7.472	102.610	114.215	144.994
std.-dev.	0.028	0.033	0.042	0.717	1.014	2.269
In Comparison to the Results of [Ren89a]:						
mean	0.8	2.5	7.1	1.4	13.1	44.0

Table 1: *Amoeba RPC Performance measured with JEWEL [microseconds]*

3.1. Amoeba RPC

The overall goal of the joint project DOSVAL (Distributed Operating Systems Validation Method) has been to provide methods and tools allowing ESA to identify distributed operating systems that are suited to be used in on-board data management system environments and to proceed with the actual validation of such identified candidates. The Amoeba Distributed Operating System [Ren89a] has been selected as the system under test.

The DOSVAL implementation of JEWEL makes extensive use of dedicated hardware resources in order to keep interference between the system under test and the measurement system low. The SUT consists of two (or more) VME crates running Amoeba 4.0 and a SUN 3/60 workstation acting as Amoeba file server, i.e. soap and bullet server. The VME crates contain pool processors according to the Amoeba terminology which are MC68030 based CPU boards. Pool processors and file server are connected by a dedicated 10 MBit/s Ethernet.† To each VME crate another MC68030 based processor board has been attached that belongs to the measurement system. The VxWorks real-time kernel is running on these boards. All VxWorks boards have been connected by a second Ethernet as measurement LAN with other workstations (SUN 4, DECstation) which host ECS and GPS. In order to solve the problem of global time the DOSVAL implementation of JEWEL uses several high-resolution clocks (640 – 5120 ns) which are synchronized with each other. SUT and measurement system communicate with each other via event queues that are located in physical shared memory on the measurement boards, and are accessible from the Amoeba boards via the VME-bus. Thus, interference between SUT and the measurement system is restricted to the overhead induced by executing the sensor statements. The execution time of a sensor statement can be determined exactly and depends on the number of measured values to be passed to the measurement system. The execution time is given by the following equation:

$$\text{execution time} = 17.4 + 1.1 * \text{number of arguments [microseconds]}$$

The constant part of the execution time is due to reading the clock, synchronising access to shared data structures, and writing an event record into this data structure. Any further processing of the measurement data is performed outside the SUT.

This configuration was used to assess the performance of the Amoeba RPC in detail. The primary goal of this performance evaluation was to demonstrate the feasibility of the JEWEL approach, secondary goal was to extend the knowledge about the Amoeba RPC mechanism itself.

Comparison of our results with those published by other researchers [Are89a, Lan90a, Ren89a] yielded close conformity and thus gave evidence for the feasibility of our approach to performance evaluation. For this validation of our JEWEL measurement environment we observed, e.g, the performance of a local and a remote Amoeba-RPC on our hardware platform. We have chosen the same RPC-sizes of 0, 8192 and 30000 bytes as van Renesse et al. in [Ren89a]. The results are shown in Table 1. Our figures are slightly better than those reported by van Renesse. This has to be explained by the different hardware (25 vs. 16 MHz M68020). But the comparison of the mean values is only one aspect our measurement. As JEWEL is able to observe the whole

† Because it is currently not possible to download the Amoeba kernel from the file server to the pool processors, another Sun 3/60 running a version of SunOS 4.1 which has been extended by the Amoeba communication protocols is used to this end.

population, and not only the mean values much more information is available. The small standard deviations, e.g., show, that nearly all RPCs execution times are closely distributed around the mean value. This might be expected in the local case but it is a notable feature in the distributed case where protocol processing has to be done. A packet loss seems to be seldom but when it happens the retransmission occurs after a timeout of 100 microseconds, as one can obtain from the maximum values in the remote case.

The second goal was also achieved, because some interesting results became apparent, which – to our knowledge – have not been reported before. Among others, we got results that indicate that the RPC mechanism of Amoeba is unfair with respect to clients that run together with the server on the same node. Table 2 presents these results that were obtained with a configuration of 1 server containing eight server-threads and eight clients either local or remote to the server. The chosen message sizes were a 0 byte call and an 8 Kbyte reply, which models the typical pattern for file server access. All clients tried to get as much RPCs done as possible. The striking fact is that in the local case all the time only one out of eight clients (the one we started last) was scheduled and was able to make RPC requests to the server (99.9 % of all RPCs made). The other seven clients could not sent more than one single RPC per second, while client number 8 reaches a mean value of 428.13 RPCs/second, which is not much worse that the results in a configuration with only one client (442.15 RPCs/second). We suspect a bug in hand-off scheduling between caller and callee that leads to this behaviour, but this has to be verified.

A more detailed description of our experiments and results can be found in [Kro92a]. As a global summary of the whole study we can state that the Amoeba RPC works as fast and stable as promised, that this very good performance scales well to very high workloads.

3.2. OSF/1 on Top of Mach 3.0

In collaboration with the OSF Research Institute, the JEWEL measurement system has been adapted to serve as a development tool for the OSF/1 operating system server on top of the MACH 3.0 microkernel in a multiprocessor environment. The primary goal of this activity is to assess the implications of a microkernel-based operating system architecture. The UNIX system call emulation done by the OSF/1 server was

Client	1	2	3	4	5	6	7	8	Totally
	Local Server								
min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	413.00	413.00
mean	0.08	0.08	0.08	0.08	0.08	0.08	0.08	428.13	428.69
max	1.00	1.00	1.00	1.00	1.00	1.00	1.00	432.00	432.00
std.-dev.	0.27	0.28	0.27	0.27	0.27	0.28	0.27	2.25	2.12
	Remote Server								
min	1.00	0.00	0.00	0.00	2.00	0.00	1.00	0.00	68.00
mean	15.21	15.86	15.19	15.63	15.77	15.56	15.59	15.61	124.42
max	21.00	21.00	21.00	21.00	22.00	23.00	21.00	21.00	127.00
std.-dev.	4.48	3.96	4.15	3.98	3.89	3.98	3.96	4.05	4.13

Table 2: *RPC frequency from 8 clients to 8 servers [RPCs/sec]*

the first aspect to be analysed. Further experiments focussed on the identification and evaluation of multiprocessor effects.

The current hardware platform is a Zenith Z1000 multiprocessor with four Intel386 CPUs (one 20 MHz, three 16 MHz) connected via a high-speed C(ache)-bus (64 MB/s) with 16 MB main memory and 64 KB write-back cache per CPU. Special emphasis has been put on the exhibition of multiprocessor effects on the system behaviour, which requires a fine granularity of the measurement time base. To achieve high portability this adaptation of the JEWEL system has been implemented purely in software, but we are dedicating two processors to the measurement system. One processor implements a counting clock with a resolution of 2 microseconds, while the second processor runs the so-called *measurement server*. The latter implements most parts of the data collection and reduction system and has been implemented as a user-level server on top of the Mach 3.0 kernel and has the regular network interface for transferring the measurement data out of the SUT. This sharing of resources between the measurement system and the observed system causes interference that has to be quantified to validate the relevance of the measured results. Several experiments with different event rates and hardware configurations have been carried out to give bounds for this interference. The instrumented OSF/1 server suffers a performance degradation of at most 5 percent compared to the non-instrumented version. Thus, the behaviour of the observed 4-processor-machine is comparable to the behaviour of the same unobserved machine configured with only two CPUs. The generalisation that an observed N-processor system is comparable in performance to an unobserved (N-2)-processor system has to be examined when performing similar experiments on a Sequent multiprocessor system with 16 CPUs in the near future. The overhead on the threads of the SUT caused by instrumentation with sensor statements depends on the relevance of events. For an event that is relevant at the instant of its detection, the sensor execution time has been measured to be

execution time = 12.0 + 2.0*number of arguments [microseconds]

The sensor statement of an irrelevant event is processed within 3.2 microseconds.

The OSF/1 operating system server has been selected as the first "victim" of the Mach adaptation of JEWEL, because of the outstanding importance of the operating system performance having impact on any running application on top. The microkernel-based version of the OSF/1 operating system, called OSF/1 MK, has been developed at the OSF RI, Grenoble [Bar92a]. Most of the original code was reused and combined with the emulation technique of the BSD-server [Gol90a] from CMU. This lead to a so-called *Single Server* approach, where the implementation of the OSF/1-functionality has moved from the kernel-level to one multi-threaded user-level server. The next step of this development could be a decomposition of the Single Server into a number of cooperating servers.

In order to preserve binary compatibility with the monolithic kernel, a user process on top of OSF/1 has not to be aware of this different operating system implementation. It simply executes the usual trap instruction if it needs an operating system service. Involving the server task is done by the trap-redirection mechanism of the Mach 3.0 kernel in conjunction with the so-called *transparent emulator library* [Gol90a] of OSF/1 MK. An instance of this library is mapped into each user task. Some caching of information about the emulated process may be done here. A trap to OSF/1 is caught by the Mach kernel and converted into an up-call to the library of the calling task. The library analyses the system call and, if the necessary information is cached, directly serves the system call without further communication. If this is not possible, the library calls an RPC stub and passes control to the OSF/1 MK server task. In the operating system server a thread receives this request, gets associated with the calling process, and acts in the server task similar to an OSF/1 user process entering the equivalent monolithic kernel. After finishing the service, the RPC returns, and the emulator library transfers control directly back to the calling task.

The objective of the analysis of the system call emulation aspect was to compare the costs of the microkernel related actions (i.e. trap redirection, emulator library execution, and message-based inter-process communication (IPC)) to the amount of time consumed by the servers real work. This amount represents a measure for the overhead introduced by the microkernel approach. The described experiments were made with the OSF/1 Single Server V3.5, the latest available version in March 1992, on top of the Mach 3.0 kernel MK67. Hardware platform for these measurements was the Zenith Z1000 multi-processor described above. The WPI *scomp* and *sdump* benchmark programs [Fin90a] have been chosen to produce the workload for the experiments. These benchmarks produce a synthetic load that is considered to characterize the UNIX *cc* and *dump* programs. For the desired experiments only a few additional lines of code had to be inserted into the code of the OSF/1 Single Server and its emulator library. The first pair of statements located inside the emulator library, marks begin and end of any RPC to the server, and the second pair, inside the servers main loop, signals the receive- and reply-operation caused by such an RPC. This instrumentation results in very fine-grained measurements, as each system call execution occurring during the execution of a (non-modified) benchmark program, can be observed exactly.

The set of UNIX system calls can be divided into three classes: calls handled completely inside the emulation library (like *getpid*), those which are served by a short operation of the server (like *lseek*), and those that impose real hard load on the server (e.g., *read*). Table 3 summarises the results of these three examples as representatives of the three classes. The overall execution time of these calls is compared to the execution time needed by the server (given in brackets). The resulting differences give a measure for the overhead of the RPC.

In some cases the communication time between the emulator library and the server takes up to 7 times as long as the processing inside the server. This is true for simple calls like *lseek*, but even for the complex *read* call communication takes about 37 percent of the total system call service time. Of course, system calls handled completely in the emulation library were executed fastest. The percentage of such system calls heavily depends upon the application: in scomp the percentage is 54 %, in sdump only 1.4 %.

Action	Mean		Std. Deviation		Minimum		Occurrences
getpid	110	(-)	35	(-)	68	(-)	225
lseek	3878	(536)	9848	(9275)	1306	(238)	114
read	5566	(3465)	7474	(7133)	1640	(432)	393

Table 3: *Execution times for selected system calls [microseconds] (based on WPI scomp benchmark)*

Execution Time microseconds	Total System Call %	Server Action %
0 – 99	0.00	0.00
100 – 199	0.00	0.00
200 – 299	0.00	19.59
300 – 399	0.00	49.48
400 – 499	0.00	16.49
500 – 599	0.00	3.09
600 – 699	0.00	2.06
700 – 799	0.00	2.06
800 – 899	0.00	0.00
900 – 999	0.00	1.03
1000 – 1099	0.00	0.00
1100 – 1199	0.00	0.00
1200 – 1299	0.00	0.00
1300 – 1399	4.39	0.00
1400 – 1499	7.89	0.00
1500 – 1599	4.39	0.00
1600 – 1699	1.75	0.00
1700 – 1799	12.28	0.00
1800 – 1899	18.42	0.00
1900 – 1999	20.18	0.00
2000 – 2099	10.53	0.00
2100 – 2199	5.26	0.00
2200 – 2299	3.51	0.00
2300 – 2399	2.63	0.00
2400 – 2499	0.88	0.00
2500 – 2599	0.00	0.00
2600 – 2699	0.00	0.00
2700 – 2799	0.88	0.00
2800+	7.02	6.19

Table 4: *Relative frequencies of the lseek execution times (based on WPI scomp benchmark)*

In contrast to many other tools JEWEL is able to provide even more detailed results. For example, in addition to the mean values the relative frequencies (empirical densities) of the execution time of a specific system call and the induced server action times can be reported, as shown in Table 4 for *lseek* in addition to numbers provided above. This table explains the large standard deviation measured for this call. The plot shows that almost all executions take less than 2500 microseconds. Only a few percent have a duration (very much) longer than 2800 microseconds resulting in a mean value of 3878 microseconds. This effect is related to some phenomenon outside the scope of the original measurement model. It may be due to a thread being preempted or blocked inside the server (currently being verified). Thus, the corresponding individuals should be discarded which is impossible without knowledge of the full population. Recomputing the mean value for the *lseek* system call results in 1861 microseconds with a standard deviation of 259 microseconds. By this example it has been demonstrated how helpful the very fine-grained measurement facilities of JEWEL are for interpreting the measurements and for getting insight into the system behaviour.

As the overall result of our measurements carried out so far concerning the discussion on microkernel-based operating system architectures it has to be stated that the induced message communication seems to be still the bottleneck for achieving comparable performance to the traditional monolithic operating system structure. This might not be so crucial in a single server, where at most one RPC is needed per system call and where caching in the emulator library avoids a lot of work. But in a decomposed multi-server system with a lot of tasks working together to provide the service this will be even more important to realize.

4. Summary

The JEWEL performance evaluation tool has been presented. The main goals of JEWEL are flexibility and preciseness. These objectives have been achieved by implementing a set of cooperating, adaptable software tools that are able to take adventage of additional hardware resources wherever possible and affordable. JEWEL allows for fine-grained event-based measurements in distributed systems. By its ability to observe the full set of individual events JEWEL can provide results which are hard to obtain with conventional tools. The flexibility and reusability of JEWEL has been demonstrated by describing its adaptation to two different environments, namely DOSVAL and Mach.

JEWEL has been used and tested at GMD for more than a year and it is now available for other interested researchers as well. The DCRS component of the Mach 3.0 adaptation has been given in the public domain. We hope it will be useful for further microkernel development.

References

[Are89a] S. Arevalo, F. Gomez-Molinero, M. Sabbatini, and R. Vadillo, *Packet Video Transmission in Amoeba*, ESTEC On-Board Data Division Technical Report, Nov. 1989.

[Bar92a] F. Barbou, P. Bernadat, M. Condict, S. Empereur, J. Febvre, D. George, J. Loveluck, E. McManus, S. Patience, J. Rogado, and P. Roudaud, "Architecture and Benefits of a Multithreaded OSF/1 Server," Research Institute Memorandum, OSF RI Grenoble (Jan. 1992).

[Fin90a] D. Finkel, R. E. Kinicki, A. John, B. Nichols, and S. Rao, "Developing Benchmarks to Measure the Performance of the Mach Operating System," pp. 83-100 in *Proceedings of the Usenix Mach Workshop* (1990).

[Gol90a] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an Application Program," in *USENIX Summer Conference*, Anaheim, Cal. (Jun. 1990).

[Kle92a] J. Kleinhans, J. Kaiser, and K. Czaja, "Hardware Support for Performance Measurements in Distributed Systems," GMD-Arbeitspapier, Gesellschaft fuer Mathematik und Datenverarbeitung mbH, Sankt Augustin (to appear 1992).

[Kro92a] R. Kroeger, F. Lange, L. Klemm, M. Lenz, A. Muenzer, and D. Schwellenbach, "Distributed Systems Validation Method – Final Report," GMD-Studie Nr.208, Gesellschaft fuer Mathematik und Datenverarbeitung mbH, Sankt Augustin (May 1992).

[Lan92a] F. Lange, R. Kroeger, and M. Gergeleit, "JEWEL: Design and Implementation of a Distributed Measurement System," *IEEE Transactions on Parallel and Distributed Systems* (July 1992).

[Lan90a] O. Langmack, "The Remote Procedure Call as Measure for Distributed Operating Systems – Concept and Initial Results Comparing Chorus and Amoeba," Final Report (1990).

[Ras88a] R. Rashid, A. Tevanian, M. Young, D. Young, R. Baron, D. Black, W. Bolosky, and J. Chew, "Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures," *IEEE Transactions on Computers* **37**, pp. 896-908 (1988).

[Ren89a] R. van Renesse, H. van Staveren, and A. S. Tanenbaum, "The Performance of the Amoeba Distributed Operating System," *Software – Practice and Experience* **19** (1989).

[Roz88a] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "CHORUS Distributed Operating System," *Computing Systems* **1** (1988).

Processor-Management in Distributed Systems with a Compute-intensive Workload

D. H. J. Epema

Delft University of Technology
The Netherlands
epema@dutiws.tudelft.nl

Abstract

An important resource in distributed systems is CPU-time, for the management of which clear objectives and good metrics are needed. For a compute-intensive workload, response time is not adequate as an objective or a metric. For such workloads, delivery of pre-defined shares of the total compute power to groups of related jobs is a much more natural objective. In order to measure CPU-usage and judge compliance of scheduling policies with this objective, we propose the concept of *rate of delivery*. We give definitions and properties of rate of delivery, show measurements, indicate problems with evaluating policies trying to achieve target rates of delivery (called *share scheduling* policies), and propose heuristics for such policies.

1. Introduction

A potential advantage of distributed computing systems in comparison to central systems is an increase in flexibility in the availability of resources. For instance, increasing the processing power of a mainframe involves upgrading the CPU or adding a CPU in a multiprocessor system, both of which methods are soon exhausted. Adding a machine to a distributed system is easy, at least from a hardware point of view. However, resource management in distributed systems is more complex than in central systems for a variety of reasons, amongst which lack of adequate software support (e.g., job migration is still not very common), lack of complete information on resource usage when taking allocation decisions at a particular node, and the impossibility to allocate more than a relatively small fraction of a resource to one entity (such as one CPU to a job).

An important resource in a distributed system is CPU-time, for the management of which clear objectives are needed. Additionally, meaningful ways to measure CPU-usage and time spent waiting for the CPUs, and good metrics to evaluate whether the objectives are met, are necessary. Traditionally, response time has played a dominant role as a performance metric, both for total system performance and for the CPU separately. In this paper we will discuss distributed systems with a compute-bound workload, in which the delivery of pre-defined shares of the total system capacity to groups of jobs (such as those belonging to one user, or to all users of a department) seems a natural objective. For policies trying to achieve this objective, response time is an inadequate metric. Therefore, we propose another measure of CPU-usage and discuss possible ways to evaluate to what extent a distributed scheduling policy achieves this objective. In order to do so, we define *rate of delivery (index),* which can also be used to state this objective in the first place. The rate of delivery (*ROD*) is the number of cycles per second delivered (to a job or user), and the rate of delivery index (*RODI*) indicates the equivalent number of machines (across a distributed system) a job or a user receives over some period of time. This concept is sometimes also called service rate.

The notion of allocating equal or pre-determined shares of the available computing power has only been studied before in the context of uniprocessors, where it passes under the name of fair-share schedulers [Ess90a, Hen84a, Kay88a]. A better name would be simply *share schedulers.* We feel that in a distributed system *RODI* is a better concept than share, because first a share is only meaningful when the total system capacity is known, and second the capacity it stands for may change over time because machines may be added or go down.

We further describe measurements for rates of delivery, problems with evaluating policies trying to achieve target rates of delivery, and policy heuristics to achieve them. The emphasis in this paper is on the definition of the concepts and the demonstration of their usefulness, rather than on analysis and evaluation of specific policies, which is planned for the future.

The work reported on in this paper was performed at the IBM T.J. Watson Research Center in Yorktown Heights, NY, USA, in the context of a project aimed at using a cluster of IBM RS/6000 RISC-machines as a high-performance compute service for compute-intensive work. An important issue there is to provide the users with service that is substantially better than they can possibly obtain on their own machines, and so a motivation of the definition of *RODI* is that the measurements should clearly show to a user the benefit of using a cluster of machines instead of only his own. Additionally, because not all users will need their complete share at all times, there is the possibility of receiving many more cycles during some periods.

This paper is organized as follows. In Section 2 we discuss some issues in CPU-time management in distributed systems with a compute-intensive workload, related work, and the present environment. In Section 3, the definitions of *ROD*(*I*) are presented, along with some examples and relations between *RODIs* of jobs and users. Section 4 contains measurements, both general CPU-usage measurements to describe the workload on the cluster, and *RODI*-measurements of individual jobs and users. In Section 5 we show that a policy stated in terms of rates of delivery to be achieved may fall short without this being its own fault, and we discuss possible metrics to judge a policy. In Section 6 we discuss policies stated in.terms of initial placement of jobs, local scheduling, and job migration in order to attain specific *RODIs*. The material in Sections 5 and 6 is preliminary and needs more research. In particular, we do not evaluate policies, but only motivate possible heuristics.

Ideally, a measure of CPU-usage should offer

1. A means for stating objectives for CPU-usage;
2. A means for stating an allocation policy of CPU-time;
3. A meaningful summary of CPU-usage by user and time;
4. A means for evaluating a share scheduling policy;
5. A means for finding the reasons why the policy goals were not met.

Section 7 contains an evaluation of $ROD(I)$ with respect to these points.

2. Managing CPU-Time in Distributed Systems

2.1. Objectives, Policies, and Mechanisms

We envision that in many organizations where compute-intensive work is done such as physics and engineering research institutes and industrial design departments, large clusters of systems (compute servers in a processor pool) will be maintained centrally, or at least on behalf of a group of users, with departments (users) having the option of requesting a fixed share (or a fixed amount) of the total compute power. Whatever computers the departments themselves own, in such an environment there will always be the need to obtain more cycles than locally available on the user's workstation, especially when the workload varies over time. The objective for processor-management is then to allocate a fixed share of the available compute power to certain *groups* of jobs, and a policy, stated in terms of initial placement of submitted jobs, local scheduling on the individual processors, and migrating jobs between compute servers, is successful when the achieved and target shares do not differ by too much. Of these mechanisms, the latter is still not very common. For a compute-intensive workload it may be implemented with current operating systems by means of checkpointing.

2.2. Related Work

In some central systems, attempts have been made to introduce what we may term *share scheduling,* that is, scheduling with as aim the delivery of specific shares to users. Henry [Hen84a] describes a very simple approach for achieving share scheduling on computers running the UNIX operating system. Kay and Lauder [Kay88a] on the other hand, describe a rather complex share scheduling policy for UNIX, based on the standard mechanisms, in terms of a large set of parameters. Essick [Ess90a] describes extensions to the UNIX scheduler in order to support fair share scheduling for large numbers of processes. Among the modifications is the recomputation of priorities when an event occurs instead of periodically, in order to decrease overhead in the scheduler. Hellerstein [Hel92a] analyzes the UNIX System V scheduler in a very elegant way, and describes the influence of the settings of the nice-values of compute-intensive jobs on their shares of the CPU (in the context of UNIX, "job" is synonymous to "process" in this paper). It turns out that the achievable shares depend on the numbers of jobs for the different nice-values. An important conclusion is that with the range of nice-values in current implementations of UNIX, the possible range of ratios of shares is limited. For instance, if there are only two processes, the maximum ratio of shares attainable is 2/3.

Also in IBM's Virtual Machine (VM) operating system, attempts at a mechanism to deliver shares to compute-intensive workloads have been made. In VM/XA, the *set share* command, which takes a numeric parameter, indicates the relative share of the CPU obtained by a virtual machine (vm). If this parameter equals 100, the vm gets the default share (which depends on the number of vms doing compute-intensive work). Higher values cause the vm to get a larger share and vice versa. Vms executing batch work can be given a lower share (say 60). It turns out [Pop91a], that the share does not behave in a linear way, but that it can effectively be used to allocate different shares to different (classes of) vms within a reasonable range.

2.3. The Present Environment

The specific environment discussed in this paper is an institution in which users have their own workstations, and are connected by a 10 Mbit Local Area Network to a cluster of similar machines (IBM System RS/6000 machines, which are RISC-machines), amongst which compute and file servers. This cluster is homogeneous in that all machines run the same operating system (AIX 3.1 of IBM, a version of UNIX), and any compute server is capable of running any job submitted to the cluster. The speeds of the CPUs of the compute servers may be different; in this paper, the capacity of a CPU is supposed to be proportional to its clock speed (which is a good indicator of performance for a compute-bound workload on RISC-machines). Each of the machines has local disk space for only paging, swapping, and temporary files. For access to the ordinary file system, they depend on the file servers. This cluster is mostly used for scientific computing, with a workload consisting of simulations and numerically intensive jobs that may take from a few hours to thousands of hours of CPU-time. The number of compute-intensive jobs per CPU is in the range 0-5 and the number of CPUs is 20 (this number varied over the time when the measurements of Section 4 were taken).

Currently, initial job placement and local scheduling are used to achieve users' shares. Initial placement can either be done by the user, who can login and submit jobs to any machine in the cluster, or through the master-machine. The master picks a compute server based on the 5-minute load averages on all machines. For local scheduling, the UNIX nice-value mechanism is used [Hel92a]. The share of each user is equal to a default, unless he has an entry in a system file with the definition of the shares. This file does not enter in the initial placement decisions, but only in local scheduling decisions. The Watson Share Scheduler [Mor91a] periodically checks the currently achieved shares of users, and if necessary adapts nice-values. Job migration is not used. It is not part of this UNIX-version, and for processes with address spaces in the range of 1 GB, which is not uncommon on this cluster, it is very expensive and so would have to be done carefully. It may be implemented in the future by means of checkpointing.

3. Rate of Delivery: Definitions and Properties

In order to compare the service delivered to a user over some period of time to what he could have obtained on a single workstation, we introduce the notions of *rate of delivery* (*ROD*) and *rate of delivery index*

(*RODI*). Both are designed to capture the amount of service delivered to a job, user or group of users on a cluster of machines, the first in absolute terms, the second relative to some standard machine (e.g., the current top of the line model).

In the following we use a single instruction and a single cycle (interchangeably) as a measure of work performed. In the realm of RISC-machines, this is reasonable; for CISC-machines, the definitions would have to be adapted. We suppose some standard machine X, which executes at a speed of C_X instructions (or cycles) per second (and by definition will have a *RODI* of 1).

Below, we start with the definition of *ROD(I)* to a single job because of the process-oriented nature of UNIX. We assume that a job stays on the same machine during its entire lifetime, although the definitions could easily be phrased so as to include the possibility of job migration. In fact, the definitions below are valid as long as a job cannot have threads executing concurrently on different machines. Then we define *ROD(I)* to a user on a single machine and on a cluster of machines, and the *ROD(I)* of a machine. Some examples are included to illustrate these definitions and some relationships among *RODI*s are derived.

3.1. Rate of Delivery to a Job

Let s,e be the time of submittal and end time of a job J, and let $C_J(t)$ be the number of cycles received by job J up to time t. The *rate of delivery to job J* between t_1 and t_2 is

$$ROD(J,\, t_1,\, t_2) = \frac{C_J(t_2) - C_J(t_1)}{t_2 - t_1}\ .$$

The *rate of delivery index to job J* between t_1 and t_2 is

$$RODI(J,\, t_1,\, t_2) = \frac{C_J(t_2) - C_J(t_1)}{C_X \times (t_2 - t_1)}$$

$$= ROD(J,\, t_1,\, t_2)\ /\ C_X\ .$$

In the context of a single process, it obviously only makes sense to talk about its *ROD(I)* for intervals starting after s and ending before e. However, the definition is more general so that it can enter in computations of *ROD(I)* to a user. In particular, if $t_2 \le s$ or $t_1 \ge e$, $ROD(J,\, t_1,\, t_2) = RODI(J,\, t_1,\, t_2) = 0$.

For a job J, $ROD(J,\, s,\, e)$ is the average number of cycles it gets per second, and $RODI(J,\, s,\, e)$ is the inverse of its expansion factor, were it running on a machine of type X. The expansion factor of a job is usually defined as the quotient of residence time and actual processing time, giving a measure of contention in the system. Obviously, for long-running jobs, the *ROD* computed over relatively short intervals (e.g., minutes) can vary considerably over time.

3.2. Rate of Delivery to a User

Let $C_{UM}(t)$ be the number of cycles received by all jobs of user U on machine M together up to time t. The *rate of delivery to user U on machine M* between t_1 and t_2 is

$$ROD(U,\, M,\, t_1,\, t_2) = \frac{C_{UM}(t_2) - C_{UM}(t_1)}{t_2 - t_1}\ .$$

The *rate of delivery index to user U on machine M* between t_1 and t_2 is

$$RODI(U, M, t_1, t_2) = \frac{C_{UM}(t_2) - C_{UM}(t_1)}{C_X \times (t_2 - t_1)}$$

$$= ROD(U, M, t_1, t_2) \; / \; C_X \, .$$

Similarly as for jobs, this definition only has a real meaning during periods that a user has at least one job on M. A maximal period with this property is called a *busy period* for user U on machine M. In this case, the definition is more general in order to deal with a cluster.

Let $C_U(t)$ be the number of cycles received by all jobs of user U across the whole cluster up to time t. The *rate of delivery to user U* between t_1 and t_2 is

$$ROD(U, t_1, t_2) = \frac{C_U(t_2) - C_U(t_1)}{t_2 - t_1} \, .$$

The *rate of delivery index to user U* between t_1 and t_2 is

$$RODI(U, t_{1,} \, t_2) = \frac{C_U(t_2) - C_U(t_1)}{C_X \times (t_2 - t_1)}$$

$$= ROD(U, t_1, t_2) \; / \; C_X \, .$$

This definition is only meaningful during a period when a user has at least one job in the whole cluster. Again, such a period will be called a *busy period.*

Above we have only taken together jobs of one user in order to define $ROD(I)$. It is clear that the definitions can be extended to define $ROD(I)$ to groups of users, by taking together all their jobs.

3.3. Rate of Delivery of a Machine

The *rate of delivery ROD(M) of a machine M* is the number C_M of cycles it executes per second. The *rate of delivery index RODI(M) of M* is $C_M \; / \; C_X$.

3.4. Examples

1. A user has two jobs J_1, J_2 on machine M with submit and end times s_i and e_i, $i = 1, 2$, with $s_1 \le s_2 \le e_1 \le e_2$. There are no other jobs on the machine from s_1 until e_2, and J_1 and J_2 share the processor equally from s_2 to e_1. From the definitions follows easily:

$$RODI(U, t_1, t_2) = C_M \, / \, C_X, \quad \text{for } s_1 \le t_1 \le t_2 \le e_2$$

$$RODI(J_1, s_1, e_1) = \frac{(s_2 - s_1) + (e_1 - s_2) \, / \, 2}{e_1 - s_1} \cdot \frac{C_M}{C_X} \ge \frac{1}{2} \cdot \frac{C_M}{C_X}$$

$$RODI(J_2, s_2, e_2) = \frac{(e_1 - s_2) \, / \, 2 + (e_2 - e_1)}{e_2 - s_2} \cdot \frac{C_M}{C_X} \ge \frac{1}{2} \cdot \frac{C_M}{C_X}$$

$$RODI(J_1, s_2, e_1) = RODI(J_2, s_2, e_1) = C_M \, / \, (2C_X) \, .$$

2. More generally, suppose a user U has jobs J_i, with submit and end times s_i and e_i, on machine M of type X, $i = 1, \ldots, I$, which together constitute a busy period. There are no other jobs. Let $s = \min\{s_i\}_i$, and let $e = \max\{e_i\}_i$. Let $mpl(t)$ be the multiprogramming level at t, and suppose that during periods when there are m

jobs, they share the processor equally, that is, each job proceeds at rate $1 / m$. Then $RODI(U, s, e) = 1$, and

$$RODI(J_i, s_i, e_i) = \frac{1}{e_i - s_i} \int_{s_i}^{e_i} \frac{1}{mpl(t)} dt .$$

3.5. Relationships of Rates of Delivery

1. Suppose $RODI(U, M_p, t_1, t_2) = r_p$, for $p = 1, \ldots, P$. Then $RODI(U, t_1, t_2) = \sum_p r_p$. This simply says that the *RODIs* add up for one user across the whole cluster.

2. If user U has jobs J_i, for $i = 1, \ldots, I$, on machine M, which are all submitted before t_1 and end after t_2, then

$$RODI(U, M, t_1, t_2) = \sum_{i=1}^{I} RODI(J_i, M, t_1, t_2) ,$$

that is, the *RODI* to a user over such a period is equal to the sum of the *RODIs* to his jobs.

3. Let $U_1, \ldots, U_K$ be all users who have a busy period that overlaps with the interval starting at s and ending at e, and let the cluster exist of machines $M_p, p = 1, \ldots, P$. Then

$$\sum_{k=1}^{K} RODI(U_k, s, e) \le \sum_{p=1}^{P} C_{M_p} / C_X = \sum_{p=1}^{P} RODI(M_p)$$

This says that the sum of the *RODIs* to users is limited to the total processing power of the cluster.

4. One may ask the following question. A job J has submit and end time s, e, and for some t_1, t_2 with $s \le t_1 \le t_2 \le e$, the (time-)average number of jobs during the interval $[t_1, t_2]$ is m (including J; there is only a single machine of type X, and we leave out all reference to it). Jobs always share the processor equally. Then how does $RODI(J, t_1, t_2)$ relate to $1 / m$? We show here that

$$RODI(J, t_1, t_2) \ge 1 / m ,$$

with equality iff the number of jobs during $[t_1, t_2]$ is constant (equal to m).

To see this, suppose that the total time during $[t_1, t_2]$ that there are m_i jobs on the machine is $u_i > 0, i = 1, \ldots, I$, with $\sum_i u_i = t_2 - t_1$, and $0 < m_1 < m_2 < \cdots < m_I$. If $m_1 = m_I$, (and $I = 1$), the assertion is clear. If $m_I = m_1 + 1$ (and $I = 2$), then

$$RODI(J, t_1, t_2) = \frac{u_1 / m_1 + u_2 / (m_1 + 1)}{u_1 + u_2},$$

and the average number of jobs is

$$m = \frac{m_1 u_1 + (m_1 + 1) u_2}{u_1 + u_2} .$$

Now $RODI(J, t_1, t_2) \ge 1 / m$ is equivalent to

$$\frac{u_1 / m_1 + u_2 / (m_1 + 1)}{u_1 + u_2} \ge \frac{u_1 + u_2}{m_1 u_1 + (m_1 + 1) u_2} .$$

Multiplying either side by $u_1 + u_2$ and $m_1 u_1 + (m_1 + 1)u_2$, and a simple algebraic manipulation shows that this holds iff

$$\frac{m_1 + 1}{m_1} + \frac{m_1}{m_1 + 1} = \frac{2m_1^2 + 2m_1 + 1}{m_1^2 + m_1} \geq 2 ,$$

which is of course satisfied.

If $m_l \geq m_1 + 2$, we consider the modified situation in which during (part of) u_l there is one job less ($m_l - 1$), and during (part of) u_1 there is one job more ($m_1 + 1$), in such a way that the average number of jobs during the whole interval remains the same. We also show that $RODI(J, t_1, t_2)$ decreases, and as we can repeat this modification until either $m_1 = m_2$ or $m_1 = m_2 - 1$, we are done.

We assume that $u_l < u_1$, the proof for the reverse or equality being similar. In the old situation,

$$RODI(J, t_1, t_2) = \frac{\sum_{i=1}^{l} u_i / m_i}{\sum_{i=1}^{l} u_i},$$

and in the modified situation,

$$RODI(J, t_1, t_2) = \frac{u_l / (m_1 + 1) + (u_1 - u_l) / m_1 + \sum_{i=2}^{l-1} (u_i / m_i) + u_l / (m_l - 1)}{\sum_{i=1}^{l} u_i} .$$

So $RODI(J, t_1, t_2)$ in the modified situation is smaller than in the old, if

$$\frac{u_1}{m_1} + \frac{u_l}{m_l} > \frac{u_1}{m_1} + u_l \left[\frac{1}{m_1 + 1} - \frac{1}{m_1} + \frac{1}{m_l - 1} \right] ,$$

or

$$\frac{1}{m_1(m_1 + 1)} > \frac{1}{m_l(m_l - 1)} ,$$

which holds because $m_l \geq m_1 + 2$.

4. CPU Usage and Rate of Delivery Measurements

In this section measurements of CPU-usage and $ROD(I)$ are presented.

4.1. General CPU Measurements

In order to give a general idea of what the workload looks like in our environment, we present some data on general CPU-usage in Figures 1 and 2. In Figure 1, the cumulative distribution function of the CPU-time of jobs is depicted. It is based on the data of 5331 jobs. It seems to be a bimodal distribution with a dip in the corresponding density at 1000 s. Around 23 % of all jobs need at least 1000 s of CPU-time. This should be contrasted to the measurements on UNIX systems under a general workload [Lel86a], where an extremely small fraction of the jobs uses a very large portion of the CPU-time.

In Figure 2, jobs are divided in three classes, small (less than 100 s of CPU-time), medium (between 100 and 1000 s), and large (over 1000 s).

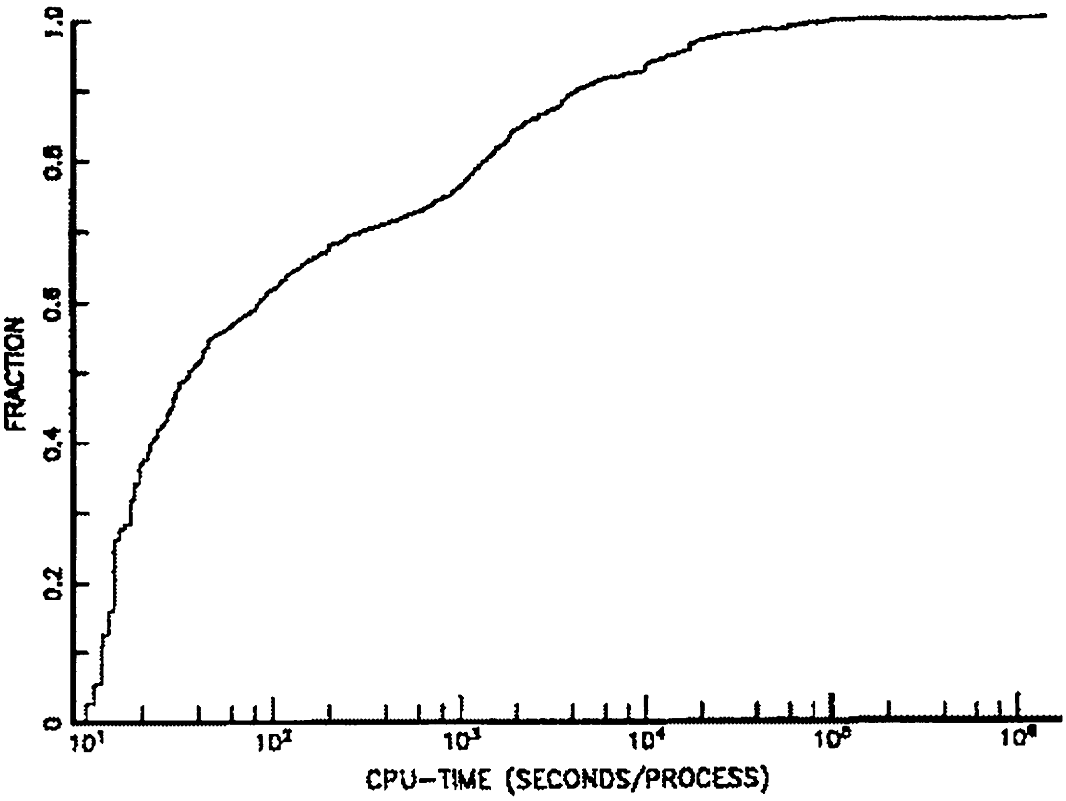


Figure 1: *Rate of delivery index for a user*

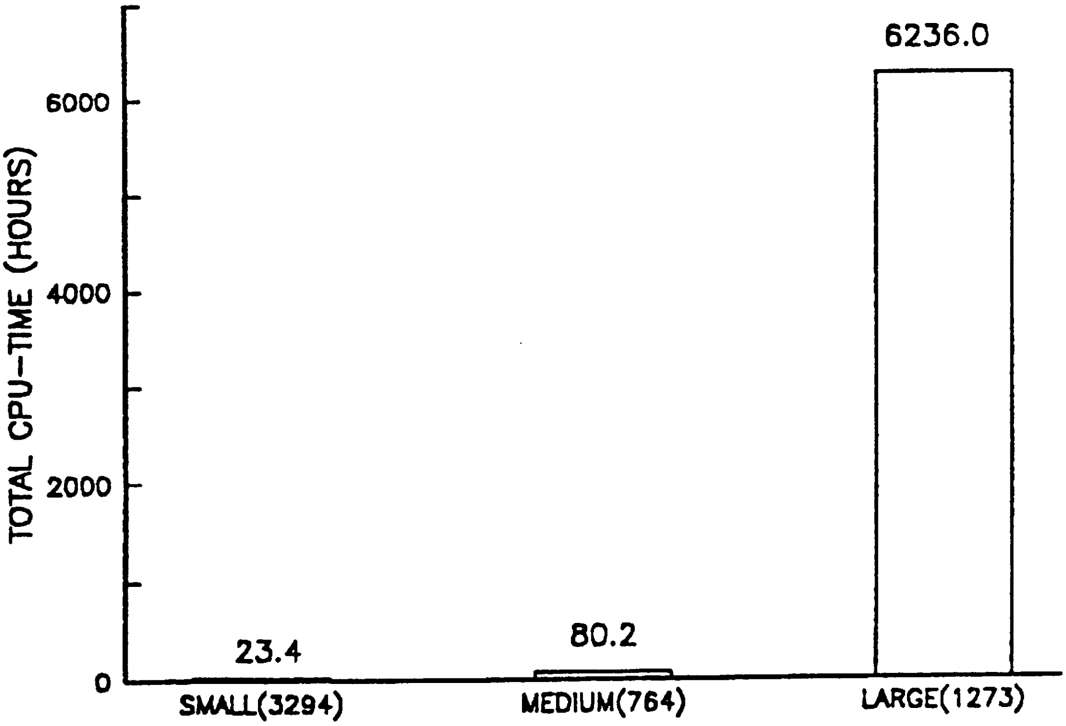


Figure 2: *Total CPU-time consumed per class (August-September 1991)*

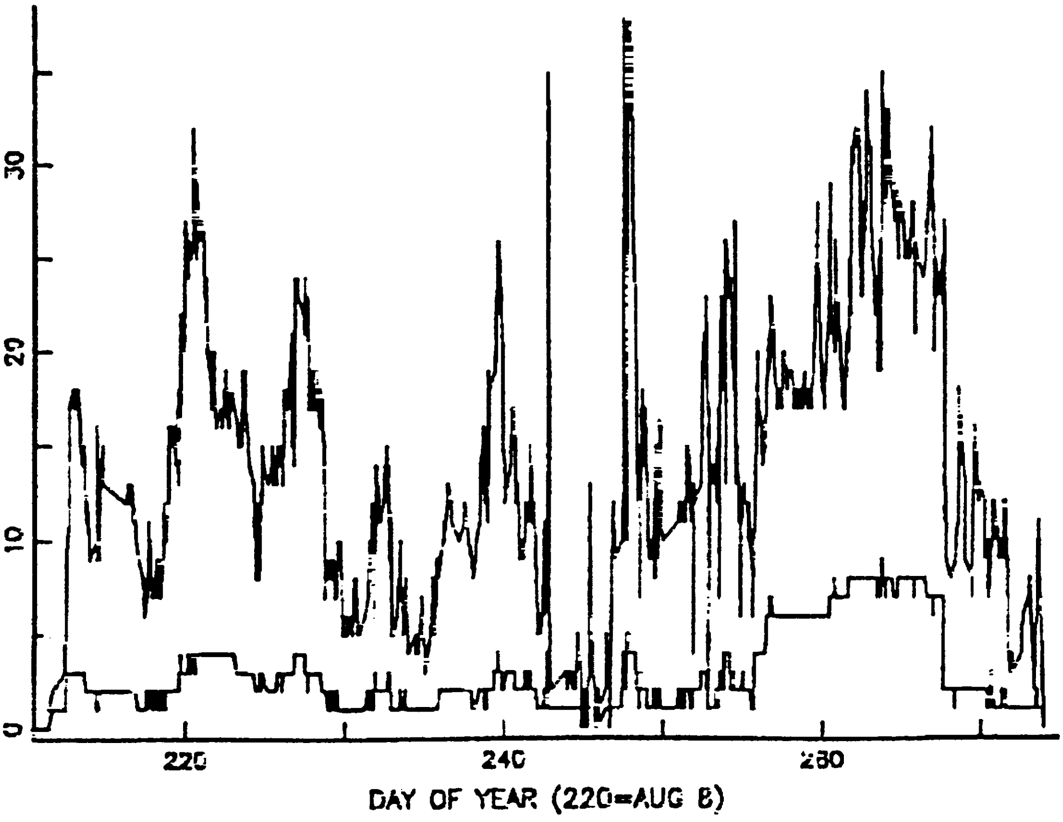


Figure 3: *Total number and 90th percentile of numbers of large jobs (August-September 1991)*

The height of the bars (indicated on top of each) is the total number of hours of CPU-time consumed by the class; the numbers in the parentheses below the horizontal axis indicate the numbers of jobs in each class.

Contrary to what one would expect in an "ordinary" environment, the number of large jobs is much larger than the number of medium jobs, showing the CPU-intensive nature of the workload.

Figure 3 gives an indication of the balance of the workload on the cluster (at a moment when there were still only 13 compute servers). The higher of the two graphs shows the total number of large jobs in the cluster, the lower the 90th percentile of the numbers of jobs on all compute servers. The unit of the horizontal axis is day of the year. Obviously, the load is well balanced when the 90th percentile is close to the average (the total number of large jobs divided by 13).

4.2. Data for Rates of Delivery

We use two sources of data for determining achieved *RODIs*, viz. job accounting data and sampling data. Job accounting data only includes for every finished job the total amount of CPU-time, its start date and time, its end date and time, and the identifier of the machine on which the job ran. What is missing in particular, is in what way a job obtained its CPU-time over its lifetime, and data of still running jobs. To obtain a very crude approximation to the real, achieved *RODIs* during an interval, we simply assume that jobs have a constant *RODI* over their lifetime, and disregard all jobs still running at the end of the interval.

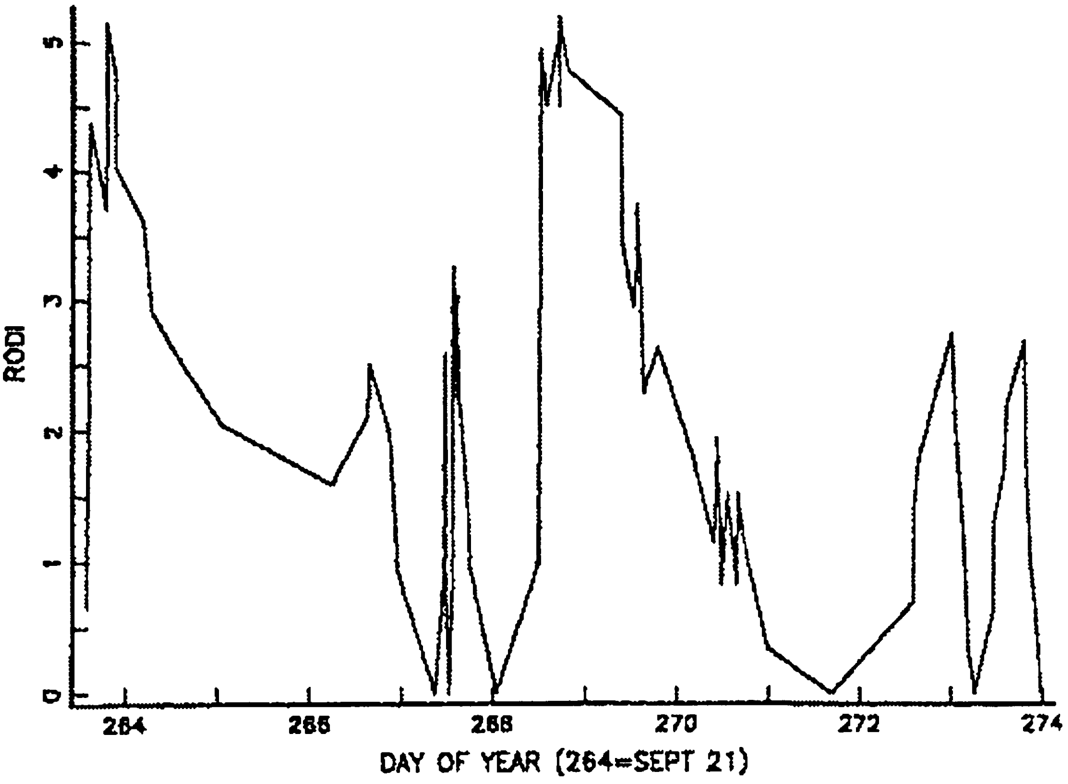


Figure 4: *Rate of delivery index for a user*

Another source of *RODI*-data is a sampler, which collects data of all running jobs on all machines at regular intervals, and at roughly the same time. For every user and machine, we compute the total amount of CPU-time consumed by all jobs contained in the sample since they started. The amount of CPU-time delivered to a user in a sample interval is computed as the difference of these amounts in consecutive samples. There are some potential errors in this computation. First, if a job exits between two samples, the result may be too low, and even negative. In the latter case, we artificially set the CPU-time consumed to 0. Second, jobs may go unnoticed if they start and finish between two consecutive samples. As we are dealing with compute-bound work, the interval over which the achieved *RODI*s are computed may be of considerable length; we use 15 minute intervals. From the sample we also compute the multiprogramming level (mpl) of a user on every machine, which is defined as the number of jobs in the sample that have received at least 5 minutes of CPU-time since they started. Note that even if a job has received less than these 5 minutes, its CPU-time is included in the computation above. To obtain the total mpl of a user, his mpls on the individual machines are simply added. His *RODI* for an interval is computed by dividing the amounts of CPU-time received by the interval length, multiplying each amount by the *RODI* of the machine, and adding across the whole cluster.

Although the measurements obtained from the accounting data are very crude, they turn out to match very well with sampling data. The measurements are used for diagnostic purposes, and to show to users what share they receive.

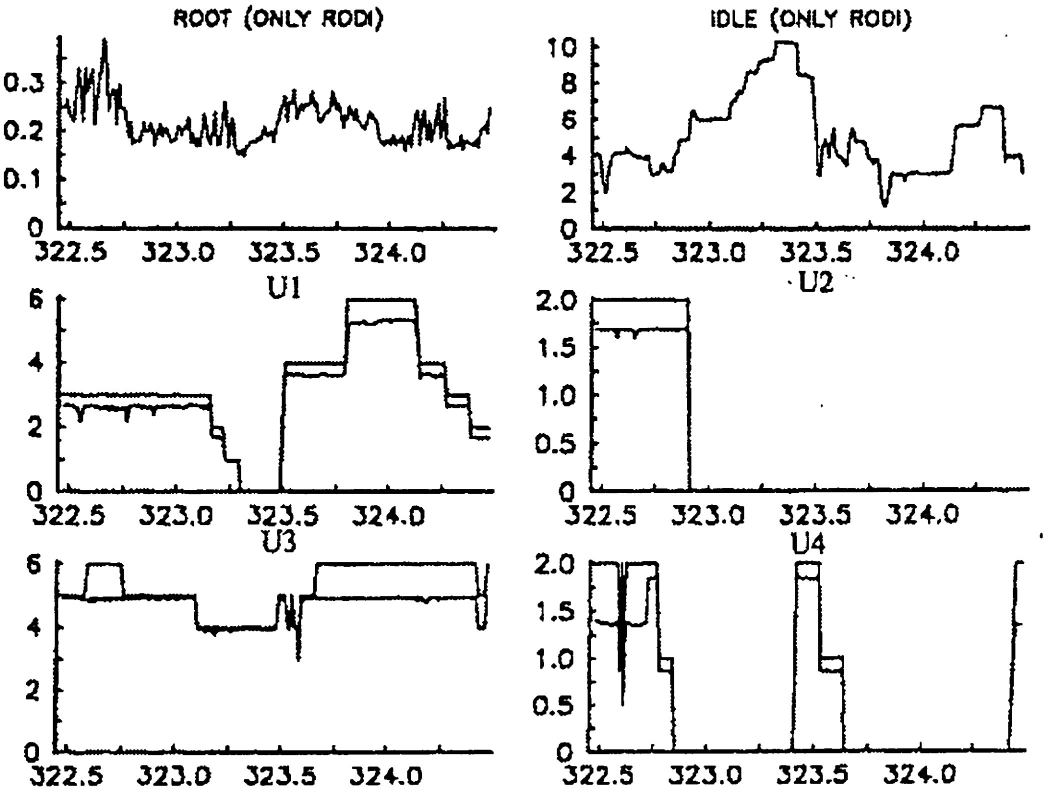


Figure 5: *Multiprogramming level and rate of delivery index of six users*

4.3. Rate of Delivery Measurements for Users

Figure 4 shows the *RODI* for a specific user during an eleven day period, obtained from job accounting data. It clearly shows that during some periods the user obtained an equivalent of 2-5 machines. In Figure 5 we show the *RODI* and multiprogramming level of some users, based on the sampling data. For userid ROOT we do not include the mpl. We introduce an artificial userid IDLE in order to deal with the idle time on the cluster. The *RODI* of IDLE is the equivalent number of idle (standard) machines (here out of a total of 18 machines).

When the two graphs coincide or are almost equal, all jobs of the user are on different machines, all of which are of the fastest type. When the deviation is larger, either his own jobs compete on some machines, some of his jobs are on slower machines, or they compete against jobs of other users. For all users shown, the allocation is near optimal. It only seems that whenever user U3 submits a sixth job, it is placed on a machine where he (and only he) was already present.

4.4. Measurements for Machines and Diagnostics

In Figures 6, 7, and 8, we show the mpl and fraction of the time the CPU is busy for all 18 machines. The mpl of a machine is the number of jobs that have received at least 5 minutes of CPU-time. Obviously, the mpl is the higher of the two graphs. Not surprisingly for a compute-intensive workload, the CPU is busy either almost 0 % or 100 % of the time. From these graphs, we can deduce whether good job assignments were made, and second, by comparing them with Fig-

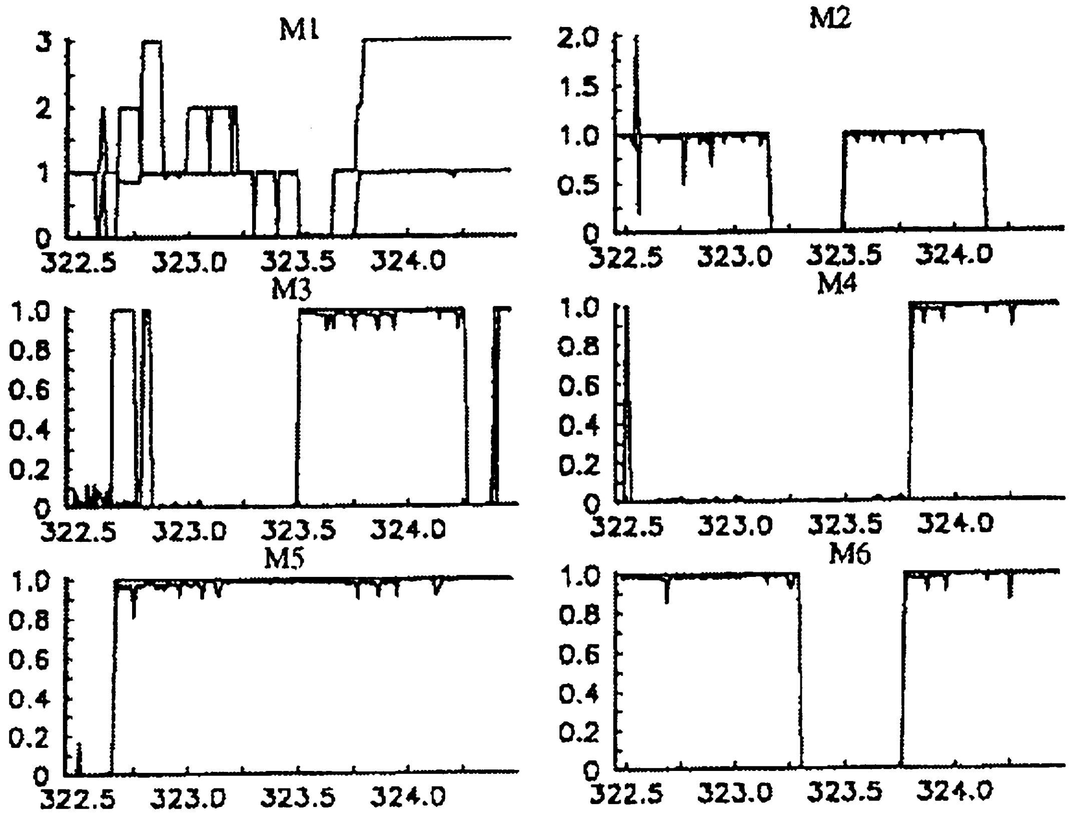


Figure 6: *Multiprogramming level and fraction CPU busy on machines M1-M6*

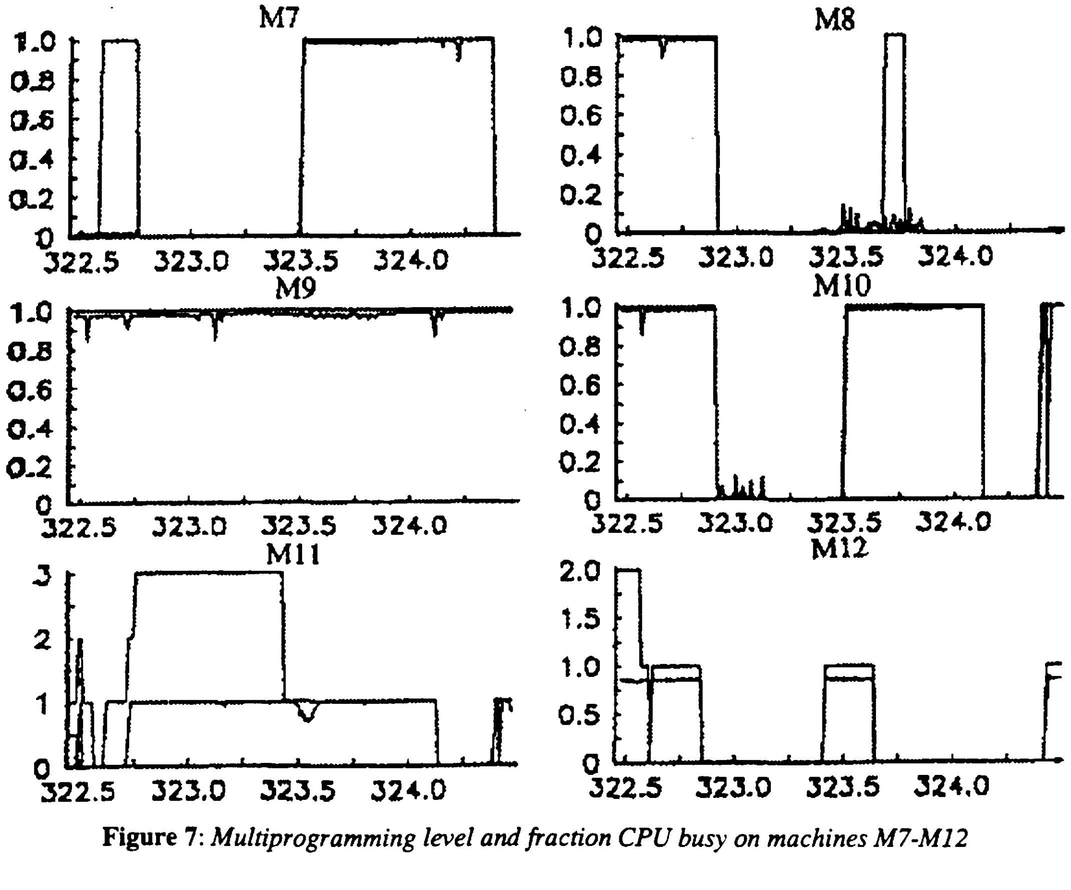


Figure 7: *Multiprogramming level and fraction CPU busy on machines M7-M12*

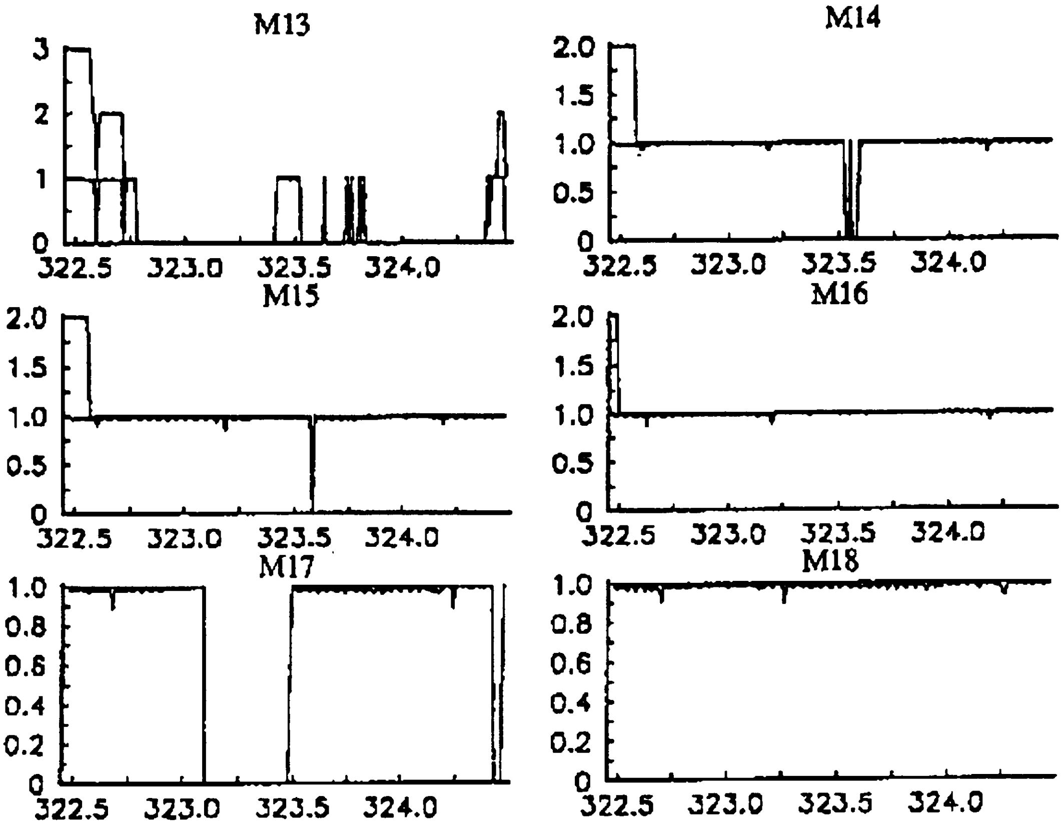


Figure 8: *Multiprogramming level and fraction CPU busy on machines M13-M18*

ure 5, which users potentially suffered from wrong assignments. In this way, these two sets of graphs are a good diagnostic tool for judging the quality of job assignment. One sees for instance, that machine M1 has at times an mpl of 2 or 3 while M3 is idle, and the same for M11 and M7. User U1 starts four jobs at 323.5 on machine 2, 3, 7, and 10.

5. Target *RODIs* versus Achieved *RODIs*

As a first attempt at a useful evaluation criterion for a policy trying to achieve target *RODIs*, if there are G groups of jobs, and group g has a target *RODI* t_g and an achieved RODI a_g, then either $max(0,\ t_g\ -\ a_g)$ or $max(0,\ (t_g\ -\ a_g)\ /\ t_g)$ may be the measure of deviation for group g, and the maximum (or some percentile) of all these values across all groups may be the total metric.

To be more precise with respect to this metric, note that the a_g may deviate from the t_g for reasons inherent to the configuration of the system and the current sets of jobs. For instance, there may be a lack of demand on the part of a group. If t_g is equivalent to more processors than the current number of jobs in a group, clearly $a_g\ <\ t_g$, no matter what the policy decides. In order to quantify this, consider a P-way multiprocessor model M with as scheduling policy in each processor priority processor sharing (PPS) as defined for instance in [Cof68a] (jobs on one processor can have different shares), and in which PPS may also be used across processor boundaries. That is, a job may spend a fraction f_p of the time at processor p, $p\ =\ 1\ ,\ldots,\ P$, with $\sum\ f_p\ \leq\ 1$. Assuming throughout this section that all machines are equal to the reference machine X, the *RODI* to this job is $\sum f_p$. Given a demand D in terms of numbers of jobs for every group, we define a *feasible RODI-allocation* in M as a G-tuple $(r_1, \ldots, r_G)$, with r_g the *RODI* of group g, induced in a natural way by an assignment of all jobs to processors and setting of local scheduling parameters. The set of all feasible RODI-allocations in M for a given demand D is denoted by $A_{M,D}$. It seems reasonable to compare the performance of a share scheduling policy in a distributed system only with elements in $A_{M,D}$.

To express the quality of a share scheduling policy in a distributed system with P processors, let us first consider a static assignment of a certain demand D, that is, there is a fixed set of jobs for each group, and the policy under consideration chooses one fixed assignment (as opposed to a policy that changes the assignments for reasons different from arrivals or departures of jobs). The performance metric of a policy we propose is the following. If the achieved *RODIs* are

$$(a) = (a_1, \ldots, a_G)$$

and the target *RODIs* are

$$(t) = (t_1, \ldots, t_G) \in A_{M,D}$$

then it is $d((a), (t))$ for some chosen metric d on G-tuples (such as the sum of squares).

If $(t) \notin A_{M,D}$, let $d = d((t), A_{M,D}) > 0$, and let

$$A'_{M,D} = \{(r) \in A_{M,D} \mid d = d((t), (r))\}.$$

Intuitively, $A'_{M,D}$ is the set of feasible allocations closest to the target. It seems only reasonable to compare (a) with elements in $A'_{M,D}$, so the metric is $d((a), A'_{M,D})$.

If either the demand changes over time because of arrivals or departures, or the policy reassigns jobs for some reason such as the gathering of more information in some nodes on the assignments in the rest of the system, we may use the expected value of the proposed metric in the static case.

6. Heuristics for Initial Placement, Local Scheduling & Migration

In principle, whenever a job arrives or leaves, one can try to recompute the best assignment of jobs to processors, given the jobs of each user (group) and the users' target *RODIs*. Obviously, in a large system with many jobs and users this is expensive. Therefore, we present in this section heuristics on how to use the three mechanisms of Section 2.1 to achieve target *RODIs* in a distributed system. We will deal with them in the natural order initial placement, local scheduling, and migration. Initial placement has to be done anyway, local scheduling is cheap, and process migration is often difficult and potentially expensive. We phrase the heuristics in terms of users instead of groups needing to achieve target *RODIs*.

We will use two secondary decision criteria. The first, which is related to users, is that jobs belonging to the same user should get more or less equal shares. In particular, no job should starve. The second, system-related, criterion is that the load in the system should be reasonably well balanced, that is, the numbers of jobs on the different processors should not vary too much. When the numbers of jobs per processor are low, this diminishes the probability of a CPU going idle. Finally, we only consider situations in which the jobs on one processor belonging to the same user get equal shares.

As to the local scheduling algorithm, we assume in this paper for simplicity that each job on a CPU can be given any share on a time scale of minutes as long as all shares add up to 1. The heuristics below are not complete and need more research.

6.1. Initial Placement

Suppose a job *J* for user *U* enters the system and has to be assigned to a processor. This can be done according to the following rules.

1. If there is an idle CPU, assign *J* to the fastest idle CPU.
   We now assume there is no idle CPU.
2. If user *U* has attained or exceeded his target *RODI* and U has a complete CPU, assign *J* to the CPU completely used by *U* such that the resulting *RODI* of *J* is largest.
3. If user *U* has attained or exceeded his target *RODI* and does not a have a complete CPU, put *J* on a CPU on which U already has jobs (only the local scheduling parameters on this machine have to be adapted). If there is more than one such CPU, put it on the CPU with this property with the least total number of jobs, and among these on the one such that the resulting *RODI* of *J* is largest.
4. If user *U* has less than his target *RODI*, assign *J* to the CPU on which the user *U'* whose *RODI* exceeds his target most has jobs (among these CPUs a further choice has to be made). Local scheduling policy parameters have to be set in such a way that U does not exceed, and *U'* does not go below, their respective target *RODIs*.

6.2. Local Scheduling

Local scheduling parameters have to be adapted at initial placement, when a job completes and leaves, and when a job is migrated. The first has been dealt with above and the latter is treated below. When job *J* leaves from machine *Y*, it is determined for all users with jobs on *Y* how their achieved *RODIs* on the system excluding *Y* compares to their target *RODIs*. Local scheduling parameters on *Y* are set accordingly.

6.3. Process Migration

Job migration should only be invoked when the current *RODI* of a user falls considerably short of his target *RODI*. Whether this is relative or absolute, and by how much it must fall short, is a parameter of the policy. Starting with the user whose *RODI* falls short most, jobs are migrated, with a maximum number of jobs per user and a maximum number of users considered each time the algorithm is invoked. This can be done periodically, or only after the assignment of a newly arriving job. The following heuristics may be considered.

1. If for a user *U* migration has to be invoked and there are idle CPUs, a job is migrated from the CPU on which *U* has jobs with the largest total number of jobs, and among these with the largest number of jobs of *U*, to the fastest idle CPU.
2. If there is no idle CPU, the choice of the job *J* to be migrated is the same, and the job ιs migrated to a CPU on which the user *U'* whose *RODI* exceeds his target most has jobs (among these CPUs

a further choice has to be made). Local scheduling policy parameters have to be set in such a way that U does not exceed, and U' does not go below, their respective target *RODIs*.

7. Conclusions and Future Research

In this paper we have discussed a performance measure, *RODI*, for compute-intensive workloads in distributed systems, for which a conventional metric such as response time is inadequate. In the introduction we mentioned five requirements for a measure of CPU-usage. Clearly, *RODI* can be used for stating objectives for CPU-usage. To judge its suitability for stating allocation policies, Section 6 needs refinement. We believe that the graphs of Section 4 show the usefulness of *RODI* for the presentation of CPU-usage by user and time. The extent to which it satisfies requirement 4 (a means for evaluating share scheduling policies) will only be clear once the discrepancies between achieved and target *RODIs* due to reasons beyond the policy employed are clear (cf. Section 5), and once we have evaluated share scheduling policies. This also holds for requirement 5 (a means for finding reasons why policy goals were not met), although we have shown that simple graphs such as those in Section 4 can be helpful. Additionally, it turns out that the concept of *RODI* is very useful for clearly stating to users the benefit of using a cluster of machines.

Obvious subjects for further research are

1. Finding a good way to compare target and achieved *RODIs* (cf. Section 5);
2. Dealing with situations in which the target *RODIs* exceed or are less than the available capacity;
3. Creating diagnostic rules for finding the reasons why target *RODIs* were not met;
4. Deriving more and better simple heuristics for scheduling policies (cf. Section 6); and
5. Evaluation of share scheduling policies.

8. Acknowledgments

The support of the High Performance Computing Department, managed by W. G. Pope, of the IBM T. J. Watson Research Center in Yorktown Heights, NY, USA, where the research reported on in this paper was performed, and of IBM The Netherlands, is gratefully acknowledged. The basic ideas of this paper were developed together with W. G. Pope. In addition, discussions with J. L. Hellerstein of the IBM Research Division were very helpful.

References

[Cof68a] E. G. Coffman and L. Kleinrock, "Feedback Queueing Models for Time-Shared Systems," *Journal of the ACM* **15**, pp. 549-576 (1968).

[Ess90a] R. B. Essick, "An Event-based Fair Share Scheduler," *USENIX*, pp. 147-161 (Winter 1990).

[Hel92a] J. L. Hellerstein, "Control Considerations for CPU Scheduling in UNIX Systems," *USENIX*, pp. 359-374 (Winter 1992).

[Hen84a] G. J. Henry, "The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal* **63**, pp. 1845-1857 (1984).

[Kay88a] J. Kay and P. Lauder, "A Fair Share Scheduler," *Comm. of the ACM* **31**, pp. 44-55 (1988).

[Lel86a] W. E. Leland and T. J. Ott, "Load-balancing Heuristics and Process Behavior," *ACM Sigmetrics*, pp. 54-69 (1986).

[Mor91a] C. M. Moruzzi and G. G. Rose, "Watson Share Scheduler," *LISA V*, San Diego, pp. 129-133 (1991).

[Pop91a] W. G. Pope, "An Experiment with VM/XA Share," *Research report RC 17145*, IBM Watson Research, NY, NY, USA (1991).

Performance Evaluation of Job Scheduling in Heterogeneous Distributed Systems

Noha Adly

Alexandria University, Egypt
Cambridge University, UK
Noha.Adly-Atteya@cl.cam.ac.uk

Magdy Nagi   Salah Selim

Alexandria University, Egypt

Abstract

In a distributed system, it is desirable to keep all the resources adequately utilized and equally loaded in order to reduce the response time of jobs and improve the utilization of the resources. Thus, arriving jobs are scheduled to be executed either locally or at a remote host depending upon the current load distribution. In this paper, a distributed scheduler is proposed to assign jobs in a distributed heterogeneous system. Heterogeneity has been considered in two aspects: different resources (type, number and speeds) and heterogeneous job types. A job is selected to be executed on a node which is best matched to the job's resource requirements. Hence, the selection of the remote host is a function of the speed of the host's resources, the current load on each resource and the job's resource requirements. A simulation model was designed and implemented to evaluate the performance of the suggested scheduler. Several experiments have been conducted and compared to an upper bound where the nodes are non-cooperative and to a lower bound where all jobs arrive to a central facility that distributes them across the nodes.

1. Introduction

In distributed systems, due to the stochastic properties of jobs – arrival, execution times and resource requirements – situation can develop whereby some of the resources are excessively busy while others are idle at the same time. This kind of situation is detrimental to performance. Therefore, given a dynamically changing workload, distributed schedulers should cooperate with each other in order to move jobs from the overloaded resources to the less busy resources based on the current state of the system.

In static scheduling [Cho82a, Tan85a, Ni85a], a priori information about the process and its resource requirements are known and routing is based on predetermined probabilities. However, variations in the workload will result in some resources becoming overloaded while others are underloaded which requires dynamic reassignments. In dynamic scheduling [Zho88a, Ram89a, Liv82a, Cas81a, Gao84a, Wan85a, Cho79a, Efe89a, Sta84a, Lin86a, Sta84b, Pul88a, Kun91a, Hac87a, Sta85a, Hac88a, Bry81a, Mir89a, Tho87a, Bon88a], it is unknown in what environment the process will execute during its life time and no decisions are made until a process begins its life in the dynamic environment. So, although making decisions at run time involves extra computation, dynamic systems have potential for adjusting workload fluctuations, enhancing system availability and adapting to system extensibility.

Distributed schedulers fall into centralized schedulers where the control resides on a single processor [Cho79a, Efe89a, Bon88a] and decentralized schedulers where the work involved in making decisions is physically distributed among processors, that is, there is no hierarchy of control within the system [Zho88a, Ram89a, Liv82a, Cas81a, Gao84a, Sta84a, Lin86a, Sta84b, Pul88a, Kun91a, Hac87a, Sta85a, Hac88a, Bry81a, Mir89a]. The weakness of the former system is its vulnerability to failure of the central node and the possibility that the central control may become a bottleneck. This approach is best suited to multiprocessor configurations rather than distributed systems.

The decentralized scheduler is composed of two components: local scheduling and global scheduling. The local scheduler refers to the scheduling discipline used by a node in executing the jobs accepted to its local queue. It provides an orderly and controlled allocation of the processors, memories and I/O devices among the various processes competing for them and decides which process runs first. The global scheduler is the part that interacts with the other hosts in order to take global decisions i.e. to decide where to execute a process.

The global scheduler is composed of two main elements: the control policy element and the information policy element. The control policy determines which jobs are eligible for transfer and selects the destination host for them. The decision is made according to the current available information on the state of the system. It is the function of the information policy to collect data for the control elements concerning the load of the system resources. The information policy decides the type of state information transmitted e.g. the queue lengths of waiting jobs [Zho88a, Liv82a, Efe89a, Sta84a, Lin86a, Pul88a, Kun91a, Hac87a, Sta85a, Hac88a, Mir89a, Mir89b], the accumulated unfinished work on the CPU [Gao84a, Hac88a], the resources utilization [Ram89a]. Also it specifies how this information is transmitted: broadcasted [Zho88a, Ram89a, Liv82a, Gao84a, Pul88a, Kun91a, Hac87a, Sta85a, Hac88a, Mir89a], sent to specific host [Sta84a], sent randomly [Liv82a]. The information is sent either periodically [Ram89a, Gao84a, Sta84a, Pul88a, Hac87a, Hac88a] or asynchronously, e.g. bidding [Sta84b], when conditions change by some amount [Zho88a, Lin86a] etc... Also, stability control is an issue which had been be considered in designing a decentralized scheduler [Cas88a, Sta85b].

Many researchers have addressed the scheduling problem, however, the survey has revealed several limitations among them:

1. Most studies considered that each node consists of a single resource (a CPU) ignoring multiplicity of resources such as disks

and tapes. This is appropriate only if there is a single bottleneck in the computer system, namely, the CPU.

2. They have restricted the local scheduler to a non-preemptive, FCFS discipline with no multiprogramming. This fits in a batch environment but not applicable in an on-line operating environment.
3. They have only evaluated homogeneous systems, that is, identical resources' speed and jobs are assumed to have identical requirements.
4. In order to distribute the work among the nodes, the technique used was equalizing the queue lengths. But unbalanced queue lengths are not necessary detrimental to performance e.g. longer queues should be allowed at faster devices for better performance. Further, queue length is an inadequate load indicator when the nodes possess multiple resources.
5. Most control policies did not specify which job is eligible to move. Once a node is found to be overloaded a job is selected at random.

Several researchers have removed some but not all of these restrictions. In [Mir89a], they have considered heterogeneous speeds but with single resource per node. They have used a thresholding technique based on the queue length where each node is associated with different threshold to cope with the speed heterogeneity. This strategy lacks adaptability to system configuration since the threshold value is set manually. Further, they have ignored multiplicity of resources and heterogeneity in the quality of arriving jobs.

In [Sta84b], they have attempted to match processes to processors based on process' CPU requirements. More specifically, they matched longer processes (in CPU time) to hosts that are less busy. However, less busy is determined, again, by an estimate of the current queue length. Further, they did not account for heterogeneity (neither speed nor job's requirements) nor for multiple resources.

In [Ram89a], they have considered multiple resources per host and described a set of heuristic algorithms to schedule tasks that have critical deadlines taking into consideration their resource requirements. A node is selected to execute a task iff it has sufficient resource surplus on each required resource such that the task will meet its deadlines. However, their technique was based on prescheduling the tasks assuming a non-preemptive policy where a task locks all its resources throughout execution. Although this assumption is acceptable for a hard real-time environment, it results in inefficient usage of resources and is not applicable in an on-line multiuser environment. Further, their objective was restricted to maximizing the number of tasks that meet their deadline i.e. they have solved crisis only with no attempt to enhance system performance.

In [Kun91a], he used different workload descriptors such as queue length, rate of CPU context switch, amount of free memory, amount of free CPU time or a combination of them. Although he used a workload consisting of a job mix (CPU and I/O bound), he concluded that the queue length was the best descriptor. This was due to that he evaluated a system consisting of diskless homogeneous workstations sharing a common file server hence again the CPU is the most important resource. Further, the descriptors were used just to classify a node as overloaded or underloaded then a job is sent randomly to an underloaded node and not based on its requirements. In [Hac87a, Hac88a], they have considered a host consisting of a CPU, a disk and terminals operating in a multiprogramming environment. However, only homogeneous systems were considered and they have relied on a thresholding technique based on the number of active processes.

In this work, a suggested scheme is proposed to alleviate the previous limitations by incorporating the following:

1. Nodes are heterogeneous, that is, each node contains a set of distinct resources (CPU, disks) possibly with different speeds.
2. A multiprogrammed time-sharing environment is considered for the local scheduler in order to maximize utilization of the resources by increasing the number of concurrent active jobs.
3. The workload is heterogeneous. It consists of different types of jobs: highly interactive jobs and batch jobs where each job has different requirements at the various resources.
4. The selection of the remote host is a function of the speed of the resources, the current load (utilization) on each resource and the jobs' requirements of each resource.
5. A job is selected to be executed on a node which is best matched to the job's requirements i.e. the selection is based on the quality of the job rather than the quantity.

Therefore the problem is defined as: given the processing time of each job on each resource, how to allocate the jobs in a dynamic environment to the heterogeneous hosts such that the response time of the jobs is minimized and the overlap between resources' utilization is maximized. This is achieved through optimizing the job mix over the whole network.

The remaining part of the paper is organized as follows. The next section presents the system model. Section 3 discusses the local and global components of the scheduler. The simulation model used to evaluate the scheduler is presented in Section 4 and 5 reports the results of our experiments. Finally, conclusion and final remarks are presented in Section 6.

2. The System Model

We assume that the system under consideration consists of N nodes connected by an arbitrary communication subnet (no specific topology is assumed). Host *i* consists of $nres_i$ heterogeneous resources: a CPU, a number of disks and $nter_i$ terminals. The resources can have different speeds. The model of a host is shown in Figure 1.

At each node, the workload consists of interactive and batch jobs. Each job has different requirements on the CPU and disks. Since interactive jobs require short service times, they are allowed to join the system immediately competing for the CPU, memory and disks. In order to control the overhead of executing many processes simultaneously, the number of concurrent active batch jobs (the batch multiprogramming level) is limited to a certain level controlled by the local scheduler. Therefore, arriving batch jobs are queued at the BATCH-QUE until the local scheduler decides to dispatch them as will be described in the next section.

Jobs arriving at a host (origin node) may be processed either locally or transferred through the communication network to another node (executing node). The results are transferred back from the executing node to the origin node where no more processing is required. Communica-

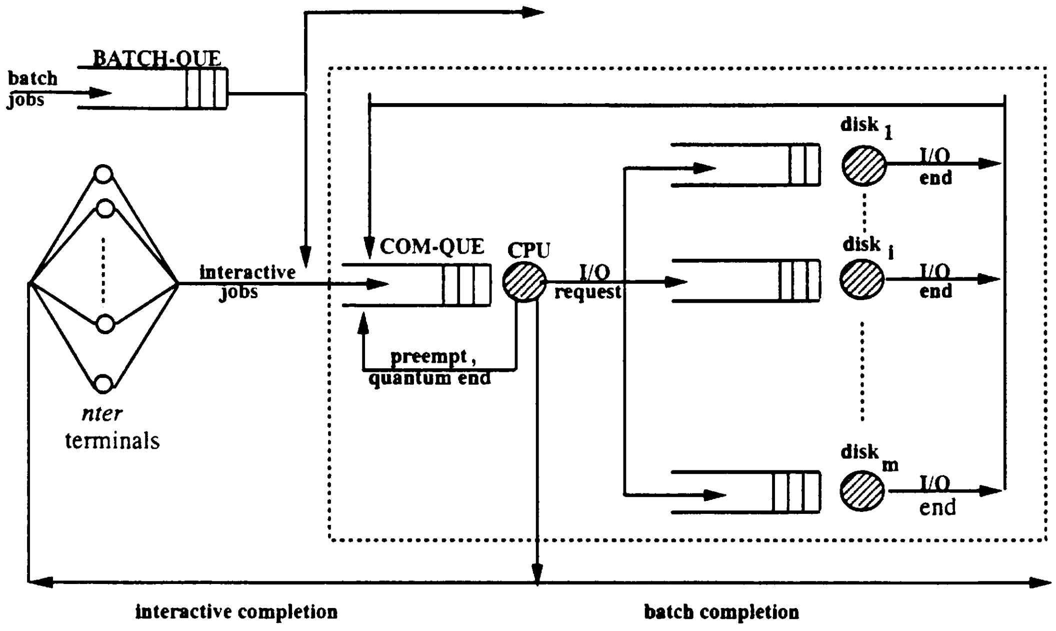


Figure 1: *The System Model of a Single Host*

tion delays are incurred during both transfers. Since interactive jobs require rapid response of their frequent requests for small service demands, their remote execution is likely to result in poor performance due to network latencies. Therefore, batch jobs are the candidates for remote assignment. We assume that entities to be scheduled are independent that is there is no communication between jobs and there is no precedence constraints. Further, once a job becomes active it will execute on that host until completion with no reassignment i.e. no process migration is allowed.

A monitor is constantly executed at each node in order to maintain and update the information about the system state. Periodically, the distributed global schedulers interact with each other in order to remove jobs from the BATCH-QUE which are heavy users of contented resources and assign them to suitable nodes with the goal of minimizing the average response time through optimizing the job mix. For reliability, the technique functions in a decentralized manner, that is there is no master controller and the algorithm runs on each node concurrently.

3. Distributed Scheduling Scheme

Scheduling occurs at two levels within the system namely, local scheduling and global scheduling. In our model, several policies could be adopted for managing the CPU and the disks but we are concerned with the management of the local BATCH-QUE since it should satisfy the same objectives as the global scheduler in order to maximize the possible benefit. This section describes the local and global management of the BATCH-QUEs with the common goal of optimizing the distribution of job mix.

3.1. The Local Scheduler

The subject of this section is the representation of a policy that manages the batch jobs queued at the BATCH-QUE of node *i* and controls the number of batch jobs active in system (the batch MPL).

In multiprogrammed time-sharing environment, performance enhancement could be achieved only by tuning the MPL and by the proper selection of the jobs that join the system. A batch job should be allowed to join the system – increasing the MPL – if and only if it is expected to use underutilized resources (if any). Such a placement approximately does not affect the performance of the running jobs since it is using their slack of resources therefore it would increase the throughput. Moreover, it reduces the time spent by the added job waiting at the BATCH-QUE, hence, it reduces the mean response time.

Therefore, the decision of increasing the MPL and selecting the most eligible job should consider: the nature of the jobs (their resource requirements), the base load on the resources which is changing dynamically, the contention of jobs on the various resources and the overhead incurred by adding the job. This discussion holds for selecting a local batch job to join the local system's queues as well as for the decision of sending a job to be executed remotely. To consider all the factors cited above it is believed that the queue length is not an adequate index to measure the load on the resources.

3.1.1. Batch Job Priorities

A suggested procedure is used to evaluate the eligibility of a job *j* on node *i*. The suggested function assigns the job a priority which is dynamically computed based on the current resources utilization of node *i* and classifies the job as *eligible/non-eligible* candidate to join the system.

The key idea is to compute the expected new utilization of every resource *r* on node *i* if job *j* is placed on the current system and to favor the job that put the maximum load on the underutilized resources while putting the minimum load on the overloaded resources. Consider the following definitions:

$r\_speed_{i,r}$	Speed of resource *r* on node *i*
$ut_{i,r}$	Utilization of resource *r* on node *i*
$req_{j,r}$	Estimated requirements of job *j* from resource *r*
min_ut_i	$= Min_r\{ut_{i,r}\}$
min_res_i	Least utilized resource of node *i* $= r$ s.t. $ut_{i,r} = min\_ut_i$

The algorithmic description of the function that computes the dynamic priority of job *j* on node *i* is given in Appendix A. Since this function will be used by the local scheduler and the global scheduler it is generalized to perform the computation on any node.

The procedure begins by computing the load that job *j* would put on every resource *r* of node *i* if job *j* is placed alone on system *i* for one time unit. This value is normalized in order not to differentiate between two jobs of the same nature but of different durations. Since the system is not empty, the new utilization should be computed taking into consideration the base load existing on each resource *r* which is reflected by $ut_{i,r}$. This is achieved by loading the least utilized resource min_res_i by the given job such that its utilization reaches 100%. This implies that min_res_i is loaded with

Jobs	Requirements			*norm_load*			*new_ut*		
	CPU	DISK1	DISK2	CPU	DISK1	DISK2	CPU	DISK1	DISK2
J1	8	20	10	80%	13%	7%	100	**89.7**	75.2
J2	16	20	40	80%	7%	13%	100	**85.2**	79.7
J3	8	0	60	67%	0%	33%	100	80	**99.5**
J4	1	120	30	9%	73%	18%	100	**567**	190
J5	2	240	60	9%	73%	18%	100	**567**	190
J6	0.5	0	120	6%	0%	94%	100	80	**1010**

Table 1: *Examples on Dynamic Priorities*

$(1 - min\_ut_i)$ / $norm\_load_{min\_res_i}$ of the load placed on min_res_i by the job. Assuming that the resource requests are uniformly distributed, the same percentage of the load placed on the other resources is used to compute their new utilization as given in step 7. Finally, the priority of job *j* on node *i* is considered as the maximum new utilization over all the resources. Jobs that have low priorities imply that they use heavily the underutilized resources while not overloading a loaded resource. On the other side, jobs resulting in high priorities implies that they are using heavily the overloaded resources.

In Table 1, examples on the priorities of different jobs are given assuming three resources one CPU and 2 disks with speeds 2 MIPS, 30 and 30 I/O req./sec and utilization 40%, 80% and 70% respectively. The CPU and disk requirements of jobs are given in Mega instructions and number of I/O respectively. The priority is given by the maximum *new_ut* over the two disks.

The proposed priorities capture the nature of the jobs and order them in such a manner that can tell which job is best suited to the current load on the system.

Then, a condition is made to classify jobs as *eligible/non-eligible* candidates: if the computed new utilization of any resource bypasses 100% and this resource is somewhat loaded, the job is considered as a non-eligible candidate. This condition discards jobs that are heavy users of the overloaded resources. Further, it accounts for the interaction between jobs on the resources: it permits to load a resource heavily if and only if this resource is lightly loaded with respect to the other resources so that their utilizations are not affected by loading this resource. Also, to account for the overhead incurred on the system by placing the job; the resource requirements of the jobs are increased by a certain amount before proceeding in the computation of the dynamic priorities. The function computes the priority of the given job and returns a value which is 0 if the job is a good candidate, 1 otherwise.

3.1.2. Control of the MPL

This function is concerned of determining the current value of the MPL at a single site. That is it decides whether to increase the MPL or not. If yes, it selects the most eligible job to join the system as described above.

It is assumed that there is a minimum level of MPL, *min_mpl*, kept as long as there are jobs in the BATCH-QUE. At job arrival, if the BATCH-QUE is empty and the current number of active jobs (*cur_mpl*) is less than *min_mpl*; the job joins the system immediately. Otherwise, it is queued at the BATCH-QUE.

In order to keep the local resources balanced, the scheduler periodically (every *loc_adj_mpl* sec) measures the utilization over the past window then attempts to increase the MPL by one job if there are good candidates jobs. The attempt is done if:

1. The resources are unbalanced i.e. if there is a significant *gap* between the load on the resources, it tries to fill it by a job of an opposite nature i.e. optimize the job mix. This condition accounts for the interaction of jobs: no need to add a job if the resources are already balanced since such action may lower the throughput. The scheduler computes the dynamic priorities of jobs waiting at the BATCH-QUE as described before; if there exists eligible candidates it selects from them the job with the lowest priority to join the system, otherwise no action is taken.
2. The system is balanced but all its resources are underutilized i.e. when $Max_r\{ut(i,r)\} < low\_thresh$ where *low_thresh* is a system parameter. In this case, the scheduler decides to increase the MPL, that is, all jobs are eligible candidates and the job with the least priority is chosen to join the system.

The *loc_adj_mpl* period should be adjusted such that it reflects the current load on the resources. Therefore, it is suggested to be set to a fraction of the average residence time of jobs in the system.

At job departure, if the *cur_mpl* is greater than *min_mpl* no action is taken. Otherwise, the scheduler waits for *dep_window* seconds during which it measures the resources utilization to indicate the effect of the departing job. Upon these measurements, it computes the dynamic priorities of jobs waiting at the BATCH-QUE and selects the most eligible job to join the system. The algorithms that describes the management of the BATCH-QUE of a single site are presented in Appendix B.

3.2. The Global Scheduler

The main objective of the global scheduler is to minimize the average response time of jobs by keeping all the resources busy – as long as there are jobs waiting for their service – and equally loaded.

3.2.1. The Information Policy

The state information passed between nodes is: the utilization of every resource, and the resource requirements of every job waiting at the BATCH-QUE. Every *info_upd_per* seconds, each node *i* measures the utilization of its resources $ut_{i,r}, \forall r$; over the past window and broadcasts it to every other node. Also, it broadcasts the resources requirements of jobs at the BATCH-QUE.

The broadcasting technique is adopted since there are small number of nodes. More sophisticated updating technique would be used if the network contains large number of nodes and a specific topology was assumed. The information sent is assumed to be delayed *info_trans_del* seconds due to transmission over the network.

3.2.2. The Control Policy

Controlling the remote execution of jobs is accomplished through two distinguished strategies. The first one controls the MPL over the whole network. The second algorithm attempts to equalize the load on the BATCH-QUEs.

Each policy is activated periodically at every node and uses the most recent information received in making decisions. Between updates no attempt is made at estimating either the current state since the last update or any future state. Old information is simply used. It is believed that the additional cost of such estimates is prohibitive in comparison to the potential benefits when the update interval is frequent (as in our case). Two nodes do not need to synchronize their state information update nor their control. Hence, the execution of both algorithms is fully asynchronous and distributed.

1. Control of the MPL

This function could be viewed as a logical extension of the local function that adjusts the MPL described in Section 3.1. It is based on the same idea: it looks for underutilized resources and selects a job to augment the MPL that needs these resources without loading an already overloaded resource. It extends the previous function to cover all nodes in selecting remote hosts. It uses the DYN-PRIO-FUN function to evaluate the jobs waiting at its BATCH-QUE on all the nodes in the network including itself and attempts to increase their MPL. This evaluation is based on the utilization of the resources of every node received from message updates.

The algorithm is executed at each node *i* every *glo_adj_mpl* seconds, a tunable system parameter. This period should be greater than the *loc_adj_mpl* period so that global decisions are less frequent than the local ones. The algorithm is presented in Appendix C. The algorithm considers only nodes that can accept work (they belong to the set *acc_set*), that is:

1. A node that has unbalanced resources i.e. when $(Max_r \{ut_{k,r}\} - Min_r \{ut_{k,r}\}) > gap$. The existence of gaps indicates the lack of work that can optimize the mix locally, therefore a remote node tends to fill the gaps by one of its jobs.
2. A node that its resources are totally underutilized i.e. when $Max_r\{ut(k,r)\} < low\_thresh$. Again, the utilization of the resources would not have been so low except when the local scheduler cannot find work to augment its MPL. Therefore, a remote node tries to send one of its jobs if it has an excess.

If all nodes does not satisfy one of these conditions then the system is observed as a heavily loaded system and the algorithm stops since it is not beneficial to move jobs. The algorithm computes the dynamic priorities of all jobs waiting at the BATCH-QUE of node *i* on all the nodes belonging to the *acc_set*. Then, for every node it considers only its eligible candidates.. It proceeds by selecting the job with the minimum priority over all nodes and sends it to the corresponding node to join its MPL. It sends no more jobs to that node. Then, it selects the next job with minimum priority over the remaining nodes. This sequence is repeated until there is no more nodes that belongs to the *acc_set* or when there are no more jobs at the BATCH-QUE.

Due to network latencies, a job should be transferred only if a significant improvement is achieved from the transfer. Therefore, in computing the dynamic priority of a job on a remote node the final value is multiplied by $thresh1(\geq 1)$. So, if a job is a good candidate locally it would result in better priority than on remote nodes.

It should be noted that placing the job directly at the MPL and not at the BATCH-QUE of the remote node is done to eliminate the delay that the job would experience waiting until the local scheduler at the destination node decides to dispatch it. Further, it eliminates the risk of job shuffling since it allows the job to move only once. So, it preserves the system stability.

2. Balancing the BATCH-QUEs

The objective of this scheme is to equalize the loads on the BATCH-QUEs of all nodes in such a manner that no node is idle while others have jobs queued. It intends to give each node an amount of work enough to keep the node busy until the effect of the next decision. In job placement, it selects the jobs that are most suitable by minimizing the maximum load over all the resources taking into consideration the baseload existing on the node, the jobs' requirements and the speed of the resources. Further, in order not to send many jobs to a node at a time, it considers the actions that could be taken by the other hosts.

The algorithm is activated at each node every *bal_que_per* seconds, a tunable system parameter. The description of the algorithm is presented in Appendix D. It assembles all jobs waiting at the BATCH-QUEs including itself at one set *tot_job*. Then, it proceeds with a greedy approach, to redistribute all jobs at the nodes with the objective of minimizing the accumulated load (in time units) of all the resources over all nodes. The accumulated load at resource *r* of node *k* is defined as the summation of the estimated finish time at *r* of every job placed at node *k*. The estimated time for job *j* to finish its service at resource *r* of node *k* is given by $req_{j,r}$ / ($r\_speed_{k,r}$ * (1 − $ut_{k,r}$)). In order not to differentiate between jobs of the same nature but of different durations; this quantity should be normalized. Further, it is multiplied by *thresh*2 (≥ 1) if job *j* does not belong to the BATCH-QUE of node *k* so remote placement is not done except when a significant improvement is achieved. The Normalized Finish Time of job *j* on resource *r* at node *k* is at step 7.

The algorithm initializes the accumulated load (*accum_load*) and the normalized accumulated load (*norm_accum_load*) of all resources at all nodes to zero then it places every job belonging to *tot_job* at every node and selects to assign the job (*migr_job*) that yields the minimum normalized accumulated load over all resources to the corresponding node (*dest_node*). It updates the *accum_load* and *norm_accum_load* of *dest_node* by placing *migr_job* then discards the job from *tot_job*. If job *migr_job* belongs to node *i* and *dest_node* ≠ *i* then node *i* sends the job to the node *dest_node* to be queued at its BATCH-QUE. This procedure is repeated by placing the remaining jobs at *tot_job* over all nodes. The algorithm stops placing jobs at node *k* when it is loaded with enough work that keeps it busy until the effect of the next decision i.e. when $Max_r \{accum\_load_{k,r}\} > min\_work$ where *min_work* is set to *bal_que_per* + *info_trans_del* + the average job transmission delay. The whole algorithm stops either when all nodes have enough work or when there is no more work to distribute i.e. *tot_job* set becomes empty. The described technique has the following effects:

1. If a node has no work at all while others are loaded; it sends some suitable jobs to the idle node.

2. If all nodes are loaded with suitable jobs; the algorithm will result in placing jobs locally with no new assignment.

3. If a node is loaded with unsuitable jobs; if all other nodes are loaded with jobs of same nature then the technique will assign each node its local jobs. If there are nodes loaded with jobs of

opposite nature it tends to exchange jobs between the nodes in order to rearrange the job mix.

It should be noted that the stability of the system is preserved since the technique simulates the actions that could be taken by other nodes. Therefore, it will not send too many jobs to a lightly loaded node as long as other nodes could send work for it too.

4. The Simulation Model

A simulation model was designed and implemented using the SLAM II simulation language for performance evaluation of the proposed scheduler. Jobs' requirements are parameterized in terms of: *cpu_req* CPU requests (in mega instructions) and $disk\_req_j$ the total number of read/write requests at disk j. The I/O requests are assumed to be uniformly distributed throughout execution. The central server model is chosen to describe the behavior of the multiprogrammed computer system. The service time per visit from the disks and the CPU at node i are drawn from an exponential distribution with mean $1 / d\_speed_i$ and $cpu\_req / (cpu\_speed_i * (1 + \sum_j disk\_req_j))$ respectively. A job may have different execution times at the various nodes due to different resources' speed. The workload consists of interactive and batch jobs. The resource requirements of all types of jobs and all system parameters, description and default values can be found in Tables 2 and 3.

The CPU scheduler adopted in this study is that of VAX/VMS operating systems [DEC88a]. It uses a **modified round-robin** form of scheduling with preemption. Interactive jobs have higher priority than batch jobs. Further, the scheduler uses the priority boosting feature which adjusts priorities dynamically to maximize the overlap of CPU usage and I/O processing. The requests pending at disks are served based on the base priority of jobs. For jobs with the same priorities they are served on FCFS basis.

The performance metrics considered are: the utilization of various resources over the network and the Normalized Waiting Time of jobs defined by $NWT = \dfrac{\sum_{j=1}^{njobs} tdep_j - tarr_j - serv_j}{\sum_{j=1}^{njobs} serv_j}$ where

$tedp_j$ = departure time of job j

$tarr_j$ = arrival time of job j

$njobs$ = total number of jobs

$serv_j$ = total service time received by job j

This metric represents the response time of jobs but normalized since jobs are of different durations. Further, the service time is omitted since the scheduler cannot optimize this component.

Dispatching batch jobs into the system improves their NWT since it reduces their waiting at the BATCH-QUE. But this action causes degradation of the NWT of interactive jobs due to the additional load and system overhead. Therefore, the NWT of each type is represented separately to show the effect of the scheduling policy on both the batch and interactive jobs.

Parameter	Description	Default value
N	Number of nodes in the network	3
$nres_i$	Number of resources at node *i*	3 (1 CPU, 2 disks)
$nter_i$	Number of terminals at node *i*	20
cpu_mips_i	CPU capacity at node *i*	2 (MIPS)
d_speed_i	Disk speed at node *i*	30 (IO/sec)
think_time	Think time delay	10 (sec)
ov_io	CPU cost of I/O operation	0.002 (mega instr.)
ov_switch	CPU cost of context switch	0.002 (mega instr.)
quantum	Time slice quantum	0.025 (sec)
p_long	Fraction of long transactions	0.4
l_long	Length of long transaction w.r.t. short	2
$f\_cpu\_bat_i$	Fraction of CPU batch jobs at node *i*	0.75
$f\_cpu\_int_i$	Fraction of CPU interactive jobs at node *i*	0.75
$int\_arr\_bat_i$	Inter-arrival time of batch jobs at node *i*	7 (sec)
loc_adj_mpl	Period to adjust the MPL locally	2 (sec)
min_mpl	Minimum number of active batch jobs	2
gap	Gap between unbalanced resources	0.3
dep_window	Utilization window length at job departure	0.5 (sec)
ovrhd	% overhead added to the job's requirements	10
low_thresh	Minimum allowable utilization	0.6
mm_qntm	The number of quantums after which MM runs	30
mm_cpu_ovrhd	CPU overhead incurred by MM	0.008 (mega instr.)
mm_io_ovrhd	I/O requests invoked by MM	2
glo_adj_mpl	Period to adjust the MPL of all nodes	8 (sec)
bal_que_per	Period to balance the BATCH-QUEs of all nodes	16 (sec)
info_upd_per	Period to transmit status information	2 (sec)
job_trans_del	Delay of job transmission	1
thresh1	Threshold of global scheduler part 1	1.5
thresh2	Threshold of global scheduler part 2	1.2

Table 2: *System Parameters*

System overhead is considered in modeling the scheduler since it has an impact on the system performance. On the local level the following overheads are considered: the context switch which is modeled as *ov_switch* instructions of CPU, the I/O pre and postprocessing overhead modeled as *ov_io* CPU instructions and the memory manager overhead is modeled by a CPU overhead (*mm_cpu_ovrhd* instructions) followed by *mm_io_ovrhd* disk operations initiated every *mm_qntm*th quantum. Further, there are several costs associated with the decentralized scheduling which include:

1. The cost of transmission delay of update information which is assumed to be a constant delay *info_trans_del* second, equal to the maximum possible delay between two nodes.

Job Type	*cpu_req* (mega instr)	$disk\_req_1$ (requests)	$disk\_req_2$ (requests)	base priority
Interactive short CPU-bound	0.2	1	1	5
Interactive short IO-bound	0.05	4	4	5
Batch CPU-bound	8	15	15	2
Batch IO-bound	1	120	120	2

Table 3: *Job Characteristics*

2. The cost of transferring jobs and results which is modeled as *job_trans_del* times of the total requirements of the job at the various resources (in time units). However, it is assumed that each node possesses a controller that is responsible of running the protocols, packing and unpacking messages, etc...

In the design of the simulation runs, every run was left to execute until the simulated system has reached a steady state. This was tested by the stabilization of the CPUs' utilization of all nodes within a 90% level of significance. After reaching steady state, all statistics are cleared to minimize the transient start up effects of an empty system. Some criterion had to be chosen to set the period of time during which the system's operation would be studied in steady state. The criterion chosen was to test for the response times of each job type separately, within a 90% level of significance.

5. Experiments

In the conducted experiments four main characteristics were studied: the effect of heterogeneous job types, the effect of the delay in the subnet, the effect of the scheduling intervals and the effect of the percentage of badly estimated requirements. Simulation results are compared with:

1. **A non-cooperative algorithm** with no global job scheduling and no network which constitutes an upper bound on system performance.
2. **An ideal scheduling scheme** where all arriving batch jobs are queued at a central BATCH-QUE to be distributed among nodes. Further, it is assumed that the ideal scheduling scheme is accomplished when the communication cost is negligible.

5.1. Effect of Heterogeneous Job Types

This experiment is conducted for a system of 3 nodes where the majority of the workload arriving at N1 and N2 consists of CPU-bounded jobs (interactive and batch) while at N3 it consists of I/O-bounded jobs. More specifically, the $f\_cpu\_int_i$ parameters are set to 0.8, 0.8, 0.15 and $f\_cpu\_bat_i$ are set to 0.75, 0.75, 0.2 for N1, N2 and N3 respectively. Results are obtained under heavy and moderate load conditions (*int_arr_bat* = 7 and 8 respectively). The rest of the parameters are set to the default values. The results of heavy load are shown in Table 4.

It is seen that a significant improvement in the batch NWT is achieved over the non-cooperative case (up to 55% under heavy load and 30% under moderate load). The improvement is on the whole system and on the individual nodes as well which is due to the fact that N1 and N2 exchange jobs with N3 which balance the load on the resources and removes the bottleneck. As a consequence of these exchanges, the interactive NWT is not affected on the overall but it is balanced on the individual nodes. It is noticed that the BATCH-QUE lengths decrease so are the MPLs for all nodes. Therefore, we can say that even though the nodes are loaded but since the jobs are of different nature, exchanging jobs can result in significant performance improvement specially under heavy loads when there is a real resource bottleneck at nodes. It should be noted that these exchanges would not have been done if the decision was based on the BATCH-QUE lengths since the queues are balanced on

Case	System/ Node	Inter NWT	Batch NWT	ut_{CPU}	ut_{IO}	Batch MPL	BATCH-QUE length
Non-cooperative	System	0.8061	9.322	–	–	–	–
	N1	0.6934	9.9	88	36	1.812	7.0
	N2	0.6825	10.4	87	38	1.935	7.7
	N3	0.9944	7.6	39	70	1.906	6.1
Algorithm	System	0.8007	4.139	–	–	–	–
	N1	0.7597	4.3	75	42	1.72	2.3
	N2	0.7546	3.97	75	42	1.75	2.3
	N3	0.86	4.1	62	59	1.82	1.6
Lower Bound	System	0.7814	2.703	–	–	–	4.0
	N1	0.7355	2.6	72	47	1.76	–
	N2	0.707	2.64	71	47	1.75	–
	N3	0.877	2.8	56	60	1.7	–

Table 4: *Heterogenous Job Types/Heavy Load*

high values which reflects that nodes are loaded and cannot receive more work.

5.2. The Delay Effect

In the system considered in this experiment – and subsequent experiments – N1 and N2 receive the same types of jobs, namely, CPU-bound but N2 has a faster CPU and N3 receives the majority of its work as I/O bounded jobs. This configuration is achieved by the following parameters settings:

	f_cpu_int	*f_cpu_bat*	*cpu_speed* (MIPS)
N1	0.8	0.75	2
N2	0.6	0.6	4
N3	0.15	0.2	2

Further, the CPU requirements of jobs arriving at N2 are 1.5 times that of arriving at N1 and N3. The system is tested under heavy loads by setting the *int_arr_bat* to 7 seconds for all nodes. The rest of parameters are at their default values. Performance could be enhanced by moving CPU jobs from N1 to N2 due to its CPU speed or to N3 due to lack of work at the CPU and by moving jobs from N2 and N3 to N1 since it has lightly loaded I/O resources. For space limitations, only the interactive and batch NWT for the whole system are shown. More details can be found in [DEC88a].

To consider the effect of different average delays for jobs moving through the subnet, the parameter *job_trans_del* is varied from 0 to 10. Results are shown in Figure 2. They show performance degradation with increased delay in the subnet as is to be expected. It is observed that under 0 delay the results are very close to the ideal balancing scheme and result an improvement over the non-cooperative case up to 64%. As the delay increases, the batch NWT increases but performance enhancement is still achieved until the transmission delay reaches 5 times the service requirements of the jobs. Beyond this value the performance degrades and it is worse than the non-cooperative algorithm. The primary reason for this degradation is that improvement can be

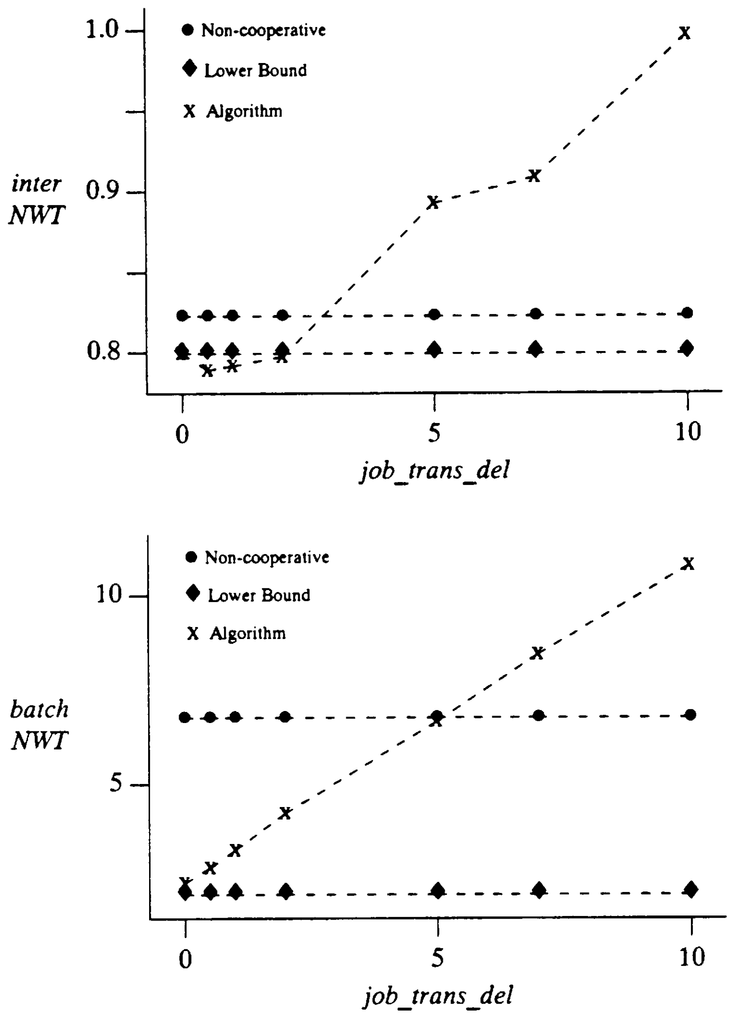


Figure 2: *The Delay Effect*

achieved only if the waiting times at the destination queues plus the time needed to move the job is less than the delay it would have experienced without moving the job. Further, jobs in the subnet are temporarily out of the system in the host's view: jobs in transit do not alter the load indices at the receiving host for longer times as delay increases. Consequently, more outdated state information is transmitted causing jobs to be moved when they should not. Wrong decisions cause increase in the MPLs and degradation in the interactive NWT (up to 20%), unbalance in the utilization of the resources and increase in the batch NWT although the BATCH-QUE lengths are balanced.

5.3. Effect of the Scheduling Intervals

The given method consists of two parts each is activated periodically. The activation intervals are different for each part: *glo_adj_mpl* and *bal_que_per* for Part 1 and 2 respectively. In the selection of these values there are two considerations that should be taken into account:

1. The scheduling period should be greater than *info_upd_per* otherwise a host would take several decisions based on the same information.
2. The scheduling intervals should be much greater than the job transmission delays plus the information update period plus the

information transmission delays so that no decisions are taken while jobs are on the way. In other words, the sender should give the receiver enough time to dispatch the new transferred jobs, update its status and send it before giving him more work.

To show the effect of varying the scheduling intervals, it has been tested by fixing the *bal_que_per* parameter to a certain value and varying the *glo_adj_mpl* parameter. Then, the *glo_adj_mpl* is fixed and the *bal_que_per* is varied.

5.3.1. Effect of Scheduling Interval Part 1

Here, the system described in Section 5.2 was considered with *bal_que_per* set to 16 seconds and the rest of parameters at the default values. The *glo_adj_mpl* was varied from 2 to 35 seconds. The results are shown in Figure 3.

It is noticed that for *glo_adj_mpl* less than 8 the algorithm results in poorest performance: the batch and interactive NWT experience high values. The primary reason for this is that the second consideration mentioned above is not true, since the delay of sending a job + *info_trans_del* + *info_upd_per* sum up to ≈ 10 seconds. Hence, taking decisions more frequently results in unnecessary job movements. In the range from 8 to 16, good performance is achieved with little dif-

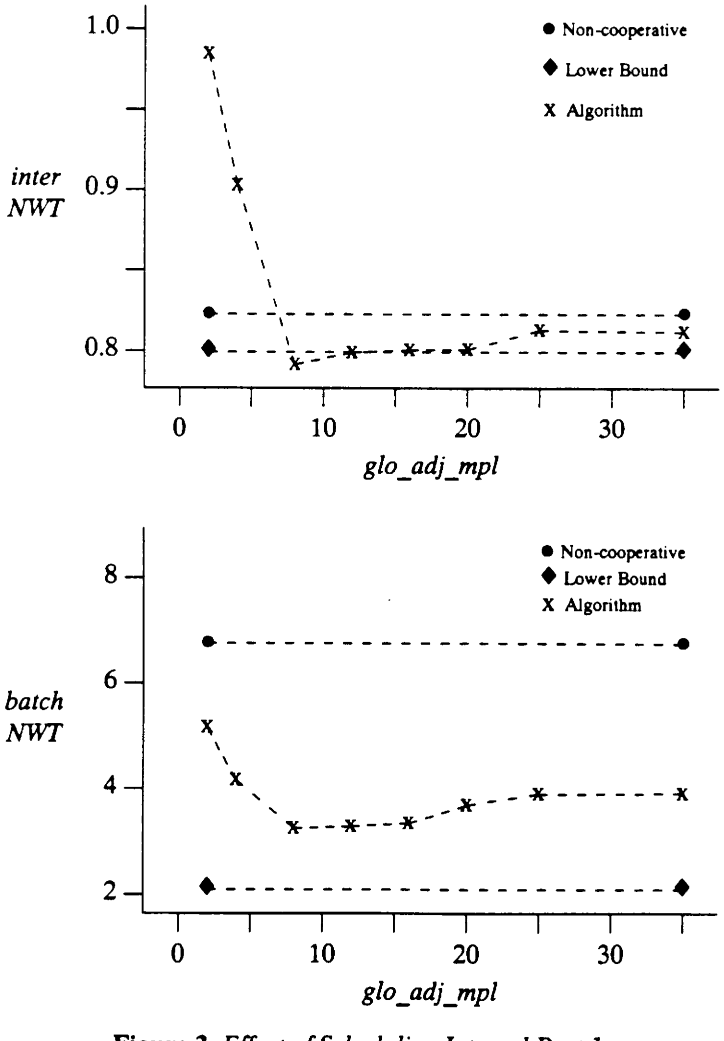


Figure 3: *Effect of Scheduling Interval Part 1*

ference. Hence, there is no need to run the algorithm faster than 16 seconds. When *glo_adj_mpl* is beyond 20, one starts to see a degradation in the batch NWT. This is due to that the reaction time is not fast enough to system dynamicity and there is not enough job movement to produce the required balance. However, the degradation is not significant as Part 2 of the algorithm is still equalizing the queues.

5.3.2. Effect of Scheduling Interval Part 2

Here, the same system is considered with *glo_adj_mpl* set to 16 and *bal_que_per* is varied from 4 to 50 seconds. The same comment applies as the last experiment. For *bal_que_per* less than 16 the algorithm results in poor performance. For *bal_que_per* above 16 and below 35 good results are obtained with little differences. Beyond this value batch performance begins to degrade slightly due to low frequency of making decisions.

5.4. Effect of the Percentage of Badly Estimated Requirements

The algorithm described is based on the assumption that the job requirements on each resource are known. More precisely, what is needed is the knowledge of the job's nature which is reflected by the ratios of the resources requirements. This assumption is valid when the job characteristics are predictable. However, under a development environment we may be faced with the problem of having badly estimated requirements. The effect of this problem had been investigated [Adl91a] and had shown that when less than 60% of the jobs have badly estimated requirements, there is an improvement over the non-cooperative case in the batch NWT up to 33% with no serious degradation in the interactive NWT. Beyond this value, the batch NWT degrades accompanied by degradation in interactive NWT up to 28%.

6. Conclusions

In this paper, we have proposed a decentralized scheduler that assigns jobs to hosts in a heterogeneous distributed system with the goal of minimizing the average response time. Two control policies are activated periodically, the first one controls the MPL of all nodes in order to keep all the resources busy while equalizing their utilization. The second one tends to equalize the load on the BATCH-QUEs such that no node is idle while others are overloaded. The assignment was made as a function of the speed of the resources, the current load on the resources and the job requirements. A simulation model was used to evaluate the performance of the proposed scheduler. We conducted a number of experiments to examine the effect of a workload consisting of heterogeneous job types. The algorithm performed well on each case and showed an improvement in the batch NWT over the non-cooperative case with approximately no effect on the interactive load performance. Further, it managed to balance the load over all the resources. For subsequent experiments, we considered a combination of a workload consisting of a job mix and different resource speeds. We examined the effect of the job transmission delay over the network and concluded that improvements were achieved when the delay is less than 5 times the job total service time. The scheduling intervals have been varied and showed that the results are stable for a wide range with degradation when these periods are too small or too large. Finally, the effect of relying on badly estimated jobs requirements had been investigated and it was concluded that improvements are achieved when the jobs have predictable nature and better results are obtained when more accurate information about the jobs is available. Thus we conclude that major improvements can be obtained by matching jobs to hosts based on their requirements specially when operating under a heterogeneous environment.

Appendix A – Dynamic Priority Function

function DYN_PRIO_FUN(j, k, i)
/* Compute the dynamic priority of job j on node i given the utilization of the resources of node i. The job belongs to node k */

```
for $r = 1$ to $nres_i$
    $norm\_load_r = \frac{(req_{j,r}/r\_speed_{i,r}) + ovrhd * r\_speed_{i,r}}{\sum_{1 \le r \le nres_i} req_{j,r}/r\_speed_{i,r}}$
endfor
$neutral\_ut = (Min_r\{ut_{i,r}\} + Max_r\{ut_{i,r}\}) / 2$
$shape = 0$
for $r = 1$ to $nres_i$
    $new\_ut_r = ut_{i,r} + \frac{(1 - min\_ut_i) * norm\_load_r}{norm\_load_{min\_res_i}}$
    if $new\_ut_r > 1$ and $ut_{i,r} > neutral\_ut$ then
        $shape = 1$ /* non-eligible candidate */
    endif
endfor
$dyn\_prio_{j,i} = \theta * Max_{r \wedge r = min\_res_i}\{new\_ut_r\}$
        $\theta = 1$          $k = i$
          $= thresh1$  otherwise
return($shape$)
end
```

Appendix B – BATCH-QUE management algorithms executed on node *i*

1. Every *loc_adj_mpl* seconds

```
Compute $ut_{i,r}$ over the past window $r = 1 \ldots nres_i$
$max\_ut = Max_r\{ut_{i,r}\}$
$min\_ut = Min_r\{ut_{i,r}\}$
if $(max\_ut - min\_ut) \le gap \wedge max\_ut \ge low\_thresh$ then stop
$\forall_j \in job\_set_i$ Do
    $shape = $ DYN_PRIO_FUN($j$, $i$, $i$)
    if $shape = 0 \vee max\_ut < low\_thresh$ then
        /* Job $j$ is an eligible candidate */
        $good\_cand_i = good\_cand_i \cup \{j\}$
    endif
enddo
if $good\_cand_i = \Phi$ then stop
$min\_prio = Min_j\{dyn\_prio_{j,i}\}$          $j \in good\_cand_i$
$best\_job = j$ s.t. $dyn\_prio_{j,i} = min\_prio$
Increase MPL by $best\_job$
end
```

2. At Job Departure

```
if $cur\_mpl \geq min\_mpl$ then stop
Wait for $dep\_window$ seconds
Measure $ut_{i,r}$ over $dep\_window$      $r = 1 \ldots nres_i$
$\forall j \in job\_set_i$ Do
    $Shape = \text{DYN\_PRIO\_FUN}(j, i, i)$
enddo
Select job $j$ with $Min_j\{dyn\_prio_{j,i}\}$ to join the system
end
```

P.S. $job_set_i = $ {set of all batch jobs waiting at BATCH-QUEi}

Appendix C – Global Scheduler Part 1

Executed every glo_adj_mpl seconds on node i ($i = 1 \ldots N$)

```
$acc\_set = \Phi$
for $k = 1$ to $N$
    $max\_ut = Max_r\{ut_{k,r}\}$
    $min\_ut = Min_r\{ut_{k,r}\}$
    $neutral\_ut = (max\_ut + min\_ut) / 2$
    $min\_res\_k = r$ s.t. $ut_{k,r} = min\_ut$
    if $(max\_ut - min\_ut) > gap \vee max\_ut < low\_thresh$ then
        /* node $k$ can accept jobs */
        $\forall j \in job\_set_i$ do
            $shape = \text{DYN\_PRIO\_FUN}(j, k, i)$
            if $shape = 0 \vee max\_ut < low\_thresh$ then
                /* job $j$ is a good candidate at node $k$ */
                $good\_cand_k = good\_cand_k \cup \{j\}$
            endif
        enddo
        if $good\_cand_k \neq \Phi$ then $acc\_set = acc\_set \cup \{k\}$
    endif
endfor
repeat until $job\_set_i = \Phi \vee acc\_set = \Phi$
    $min\_prio = Min_{k,j}\{dyn\_prio_{j,k}\}$   $\forall k \in acc\_set$
                                                 $\forall j \in job\_set_i \wedge j \in good\_cand_k$
    Let $best\_job$ and $dest\_node$ s.t.
        $dyn\_prio_{best\_job,\, dest\_node} = min\_prio$
    Send $best\_job$ to $dest\_node$
    $acc\_set = acc\_set - \{dest\_node\}$
    $job\_set_i = job\_set_i - \{best\_job\}$
endloop
end
```

Appendix D – Global Scheduler Part 2

Executed every *bal_que_per* seconds on node i ($i = 1...N$)

```
$tot\_job = \cup_k \{job\_set_k\}$        $k = 1...N$
     /* set of all batch jobs on all BATCH-QUEs */
$acc\_set = \cup_k \{k\}$                 $k = 1...N$
$\forall j \in tot\_job$ do
   for $k = 1$ to $N$
      $tot\_ser\_time = \sum_r \frac{req_{j,r}}{r\_speed_{k,r}}$
      for $r = 1$ to $nres_k$
         $norm\_fin\_time_{j,k,r} = \frac{req_{j,r} / r\_speed_{k,r}}{tot\_ser\_time * (1 - ut_{k,r})} * \alpha$
            $\alpha = 1$             $k = i$
              $= thresh2$     otherwise
      endfor
   endfor
enddo
$accum\_load_{k,r} = 0$            $\forall k, r$ /* accumulated load*/
$norm\_accum\_load_{k,r} = 0$      $\forall k, r$ /* normalized accum.load*/
repeat until $acc\_set = \Phi \vee tot\_job = \Phi$
   $\forall j \in tot\_job$ do
      $\forall k \in acc\_set$ do
         $max\_load_{k,j} = Max_r \{norm\_accum\_load_{k,r} + norm\_fin\_time_{j,r,k}\}$
      enddo
   enddo
   $min\_load = Min_{k,j} \{max\_load_{k,j}\}$
   let $migr\_job$ and $dest\_node$ s.t. $max\_load_{dest\_node,\, migr\_job} = min\_load$
   if $migr\_job \in job\_set_i \wedge dest\_node \neq i$ then send $migr\_job$ to $dest\_node$
   for $r = 1$ to $nres_{dest\_node}$
      $accum\_load_{dest\_node,r} = accum\_load_{dest\_node,r} + \frac{req_{migr\_job,r}}{r\_speed_{dest\_node,r} * (1 - ut_{dest\_node,r})}$
      $norm\_accum\_load_{dest\_node,r} = norm\_accum\_load_{dest\_node,r} + norm\_fin\_time_{migr\_job,r,dest\_node}$
   endfor
   if $Max_r \{accum\_load_{dest\_node,r}\} \geq min\_work$ then $acc\_set = acc\_set - \{dest\_node\}$
   $tot\_job = tot\_job - \{migr\_job\}$
endloop
end
```

References

[Adl91a] N. Adly, *Performance Evaluation of Job Scheduling in Heterogeneous Distributed Computer Systems*, M.Sc. dissertation, Computer Scinece Dept., Fac. of Eng., Alexandria University, 1991.

[Bon88a] F. Bonomi and A. Kumar, *Adaptive Optimal Load Balancing in a Heterogeneous Multi-server System with a Central Job Scheduler*, Proc. 8th Int. Conf. on Distributed Computing Systems, 1988.

[Bry81a] R. Bryant and R. Finkel, *A Stable Distributed Scheduling Algorithm*, Proc. 2nd Int. Conf. on Distributed Computing Systems, 1981.

[Cas88a] T. Casavant and J. Kuhl, "Effects of Response and Stability on Scheduling in Distributed Computing Systems," *IEEE Trans. on Soft. Eng* **13**, pp. 1578-1587 (November 1988).

[Cas81a] L. Casey, "Decentralized Scheduling," *Australian Computer Journal* (May 1981).

[Cho82a] T. Chou and J. Abraham, "Load Balancing in Distributed Systems," *IEEE Trans. on Soft. Eng* **SE-8**(4), pp. 401-412 (July 1982).

[Cho79a] Y. Chow and W. Kohler, "Models for Dynamic Load Balancing in Heterogeneous Multiple Processor System," *IEEE Trans. on Computers* **C-28**(5) (May 1979).

[DEC88a] DEC, *VMS System Management Manual*, Digital Equipment Corporation, 1988.

[Efe89a] K. Efe and B. Groselj, *Minimizing Control Overheads in Adaptive Load Sharing*, Proc. 9th Int. Conf. on Distributed Computing Systems, IEEE, June 1989.

[Gao84a] C. Gao, J. Liu, and M. Raily, *Load Balancing Algorithms in Homogeneous Distributed Systems*, Proceedings of the 1984 Int. Conf. on Parallel Processing, August 1984.

[Hac87a] A. Hac and X. Jin, *Dynamic Load Balancing in Distributed Systems Using a Decentralized Algorithm*, Proc. 7th Int. Conf. on Distributed Computing Systems, June 1987.

[Hac88a] A. Hac and X. Jin, *Dynamic Load Balancing in a Distributed System Using a Sender-Initiated Algorithm*, Proc. 8th Int. Conf. on Distributed Computing Systems, 1988.

[Kun91a] T. Kunz, "The Influence of Different Workload Descriptors on a Heuristic Load Balancing Scheme," *IEEE Trans. on Soft. Eng* (July 1991).

[Lin86a] F. Lin and R. Keller, *Gradient Model: a Demand-Driven Load Balancing Scheme*, Proc. 6th Int. Conf. on Distributed Computing Systems, IEEE, May 1986.

[Liv82a] M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed systems," *Proc. ACM Computer Networks Performance Symp* (September 1982).

[Mir89a] R. Mirchandaney, D. Towsley, and J. Stankovic , *Adaptive Load Sharing in Heterogeneous Systems*, Proc. 9th Int. Conf. on Distributed Computing Systems, 1989.

[Mir89b] R. Mirchandaney, D. Towsley, and J. Stankovic, "Analysis of the Effects of the Delays on Load Sharing," *IEEE Trans. on Computers* **38**(11) (November 1989).

[Ni85a] L. Ni and K. Hwangi, "Optimal Load Balancing in a Multiple Processor System," *IEEE Trans. on Soft. Eng* **SE-11**(5) (1985).

[Pul88a] S. Pulidas, D. Towsley, and J. Stankovic, *Imbedding Gradient Estimators in Load Balancing Algorithms*, Proc. 8th Int. Conf. on Distributed Computing Systems, 1988.

[Ram89a] K. Ramaritham, J. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks With Deadlines and Resource Requirements," *IEEE Trans. on Computers* **38** (August 1989).

[Sta84a] J. Stankovic, "Simulation of Three Adaptive, Decentralized Controlled Job Scheduling Algorithms," *Computer Networks* **8**(3) (June 1984).

[Sta84b] J. Stankovic and I. Sidhu, *An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups*, Proc. 4th Int. Conf. on Distributed Computing Systems, May 1984.

[Sta85a] J. Stankovic, "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling," *IEEE Trans. on Computers* **C-34**(2) (February 1985).

[Sta85b] J. Stankovic, "Stability and Distributed Scheduling Algorithms," *IEEE Trans. on Soft. Eng* **SE-11**(10) (October 1985).

[Tan85a] A. Tantawi and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," *J. ACM* **32**(2), pp. 445-465 (April 1985).

[Tho87a] A. Thomasian, *A Performance Study of Dynamic Load Balancing in Distributed Systems*, Proc. 7th Int. Conf. on Distributed Computing Systems, 1987.

[Wan85a] Y. Wang and R. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. on Computers* **C-34**(3), pp. 204-217 (March 1985).

[Zho88a] S. Zhou, "A Trace Driven Simulation Study of Dynamic Load Balancing," *IEEE Transactions on Software Eng* **14**(9) (September 1988).

PARABASE – A Federated DBMS Architecture Supported by a Shared Nothing Database Server

M. Kollingbaum
T. Mueck G. Vinek

Dept. of Information Systems
University of Vienna
Austria
{ mueck | vinek }@ifs.univie.ac.at

Abstract

The **PARABASE** research effort focuses on shared nothing MIMD machines used as high performance DB servers in federated DBMS architectures. In particular, one of the project goals is to improve the responsiveness of large scale online transaction processing applications in commercial environments using the performance provided by massively parallel machines. At the moment, all prototype components are designed and implemented for an iPSC/2 hypercube machine linked into an TCP/IP based network.

Two key features of the PARABASE approach which are addressed in this paper are

- An application transparent multi-level parallel query and update concept yielding a high degree of parallelisation for DB client requests and
- A highly specialised parallel file system which is able to support multiattribute tuple access operations on the file system call level.

Those two key features are described in the context of the overall hardware and system software environment provided by the shared nothing parallel server machine as well as with respect to the federated DBMS paradigm used to handle the DB services in a distributed and heterogeneous client environment.

1. Introduction and Project Motivation

Considering the practical requirements for modern database management system architectures, as stated by large business oriented corporations with long range information management policies, yields at the moment at least three important topics, namely

- High performance DBMS architectures designed to handle large scale online transaction processing applications based on broadband (e.g. B-ISDN) data networks,
- Federated DBMS architectures combining private databases to be held on local workstations and shared databases to be held on performant DB server machines by means of a simple information exchange mechanism (i.e. CHECK-OUT and CHECK-IN of data sets), and
- Heterogeneous DBMS architectures designed to integrate different DBMS products eventually belonging to different DBMS paradigms (relational, object oriented, hierarchical, ...) thus providing at least a minimum of interoperability between different products and DB paradigms.

Reacting to those requirements to a certain extent, considerable DBMS research efforts take place. Some of those efforts aim at the efficient utilisation of parallel processing power in database management and result in DBMS development projects on parallel machines (see [DeW86a, Cop88a]). Others deal with federated architectures in distributed hardware environments (see [Kim91a]).

Using the experiences and results of those projects, the **PARABASE** research effort outlined in this paper focuses on shared nothing MIMD machines used as high performance DBMS servers participating in distributed workstation environments, in particular on a prototype designed and implemented for an iPSC/2 hypercube machine linked into an TCP/IP based network. The main reason not to use shared memory architectures (see for example [Hon90a] for a description of a shared memory approach) is the rapid progress in the development of high-speed processor interconnection technologies. Considering for example the internode bandwidth claimed for the latest member of intel's supercomputer family (i.e. PARAGON), an improvement of about 2 magnitudes between the iPSC/2 and the PARAGON production machines (from 2.8 MB/sec to about 200 MB/sec) can be observed. In [Fri90a], an internode bandwidth of 10GB/sec for fibre optic links is claimed under laboratory conditions.

Such technologies will compensate for the current inter-processor data transfer bottleneck caused by the huge data transfer volumes of data intensive applications like DBMS. In other words, they will provide superior DBMS performance on shared-nothing massively parallel architectures according to standard arguments like speed-up and scaling for shared-nothing machines (see for example [Fri90b, Fri90a] or [Pen92a]).[†]

Focusing on the high performance DBMS server software of our federated architecture, two key features of the PARABASE approach are an application transparent multi-level parallel query and update concept on the one hand and a highly specialised parallel file system supporting multiattribute access on the other hand. Our multi-level approach towards query and update parallelisation includes the parallel processing of concurrently issued query and update requests, the parallel processing of query chunks (e.g. four joins belonging to the same SQL-style query are processed in parallel) and the parallel processing of some query chunks itself by the underlying file system (e.g. a multiat-

[†] It should be mentioned, however, that *at the present moment,* any DBMS system optimised for a shared memory architecture with a relatively small number of processors which is *not going to hit the limit of the system bus bandwidth during query processing* will very likely outperform any other DBMS system designed for a shared nothing architecture with the same number of processors (in [Hon90a] a factor of 2 is claimed). However, considering realistic project turn-around times even for prototype developments, we are quite convinced about a timely appearance of adequate link technology.

tribute range query is spread over a disk array). Consequently, a specialised file system is needed in order to process range queries against tuple sets as low level file system operation instead of conventional low level read/write operations against flat files (see [Wit91a] for an outline of the file system and [Mue91a] for a description of the data structure).

After a brief description of the current hardware platform and the corresponding system software environment in Section 2, we give an outline of the PARABASE federated DBMS architecture with respect to the multi-level query and update parallelisation concept. Since the resource investment for the design and implementation of a nonstandard parallel file system has to be motivated, we describe our rationale for this step in Section 4. Additionally, a brief description of the underlying index data structure is given. Section 5 deals with a number of technical issues regarding the new file system, as there are organisational structure, data distribution policy and interprocess communication pattern. Section 6 provides conclusions and a short outlook at work in progress and the research agenda in general.

2. Hardware Environment and System Software

At the present moment, an eight processor iPSC/2 machine is used as shared nothing database server. For workstations running local applications or other DB client machines, the iPSC/2 DB server is accessible via TCP/IP. System software includes UNIX System V at the "system resource manager" (SRM), an i386 based workstation which serves as a front end system to the actual hypercube, a UNIX derivate called NX/2 as symmetric node operating system and the intel supplied "concurrent file system" (CFS, see [Pie89a] for details) used to operate the disk array.

In contrast to ordinary iPSC/2 or iPSC/860 platforms, all processors of the project configuration serve both as computing nodes **and** as I/O nodes. Each node is equipped with a standard SCSI controller and, at least at the present moment, with one 650MB SCSI disk. The architectural distinction between computing nodes accessible for application processes and mass storage nodes (so called I/O nodes) only accessible via file system calls (as used by intel for NIC applications) would be counterproductive for a mass storage oriented project and has been omitted for that reason. Consequently, the current hardware configuration is shown in Figure 1.

The different communication hardware technologies depicted in Figure 1, namely standard Ethernet for client-server communication, intel proprietary DirectConnect for node-node communication and SCSI for node-controller communication, yield to challenging problems with respect to bandwidth balancing. Additionally, considering the rapid change in communication hardware and the resulting rapid change of bandwidth ratios between the communication layers (client-server, node-to-node, node-controller), any parallel DBMS architecture has to provide means for optimisations in case of changing bandwidth ratios.

Under these circumstances, the absolute performance of our project configuration is not relevant.† The important point with respect to the hardware and system software configuration is that it allows for the design, the implementation and, above all, the *evaluation* of different parallel database system concepts in a common framework, namely a

† Obviously, an arbitrary commercially used state-of-the-art database machine will clearly outperform our current configuration as well as, to our present knowledge, any other research project configuration in a university environment.

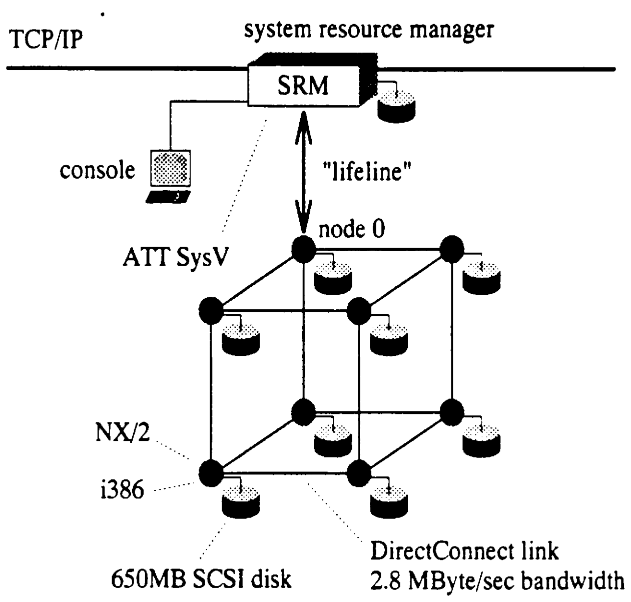


Figure 1: *iPSC/2 database server hardware configuration*

shared nothing MIMD machine running a symmetric node operating system based on well-known standard concepts.

3. An Outline of the PARABASE Architecture

The database paradigm chosen for the PARABASE approach is known as *federated* DBMS paradigm. In a federated DBMS architecture, several clients use their private databases (PDB) on local machines and a number of shared databases (SDB) on one or more host machines. Additionally, some data sets (tuple sets, object sets, ...) can be transferred to or from a shared database employing a simple yet elegant exchange mechanism, namely the CHECK-OUT and CHECK-IN protocol known from [Kim91a]. This concept is based on a data set transfer from a shared database to a private database (CHECK-OUT), an eventually long lasting data manipulation phase and a final retransmission of the data set to the shared database (CHECK-IN). Figure 2 is meant to illustrate that data exchange schema.

According to this paradigm, the PARABASE process architecture consists of

- Local DBMS server processes, called **LSP**, running on local workstations for private database manipulation and DB server communication.
- A load-balancing process located on the SRM used to dispatch client request, e.g. insert, update and delete requests as well as query requests.
- A pair of service processes per processing node, i.e. a file sever process, called **FSSP**, for mass storage requests and a DB server process, called **NSP**, for high-level data manipulation and query processing.

Figure 3 illustrates the PARABASE overall process architecture. Additional information concerning the query processing policy is given below.

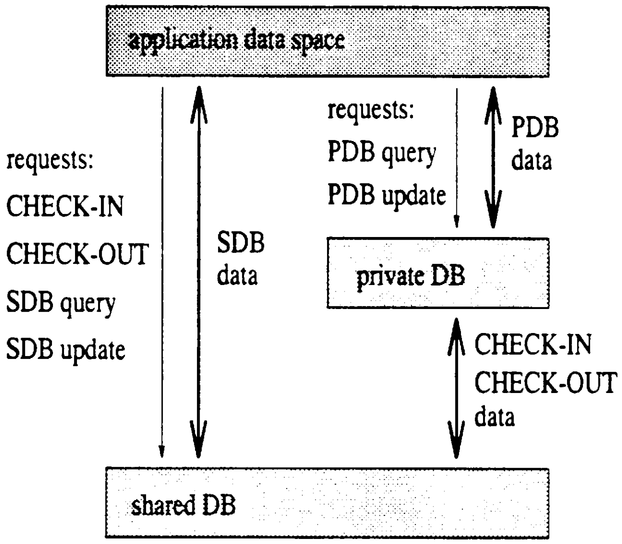


Figure 2: *CHECK-IN and CHECK-OUT in a federated DBMS architecture*

Basically, each local DBMS server process is able to launch DBMS requests, i.e. data manipulation and query operations, schema modifications and CHECK-IN/CHECK-OUT requests which are dispatched by a high-level load balancing process on the SRM.

Focusing the discussion at query requests, each concurrently issued query, is routed to one responsible node server process, parsed and broken up into several query chunks. Typical query chunks are selections, projections, joins, unions and so on. Some of those query chunks can be handled directly by the underlying file system, e.g. orthogonal range queries, exact match queries and partial match queries issued against single tuple sets. All other query chunks do not correspond directly to file system calls and require additional actions by the NSP. The most prominent example for this type of query chunk is the relational join operator.

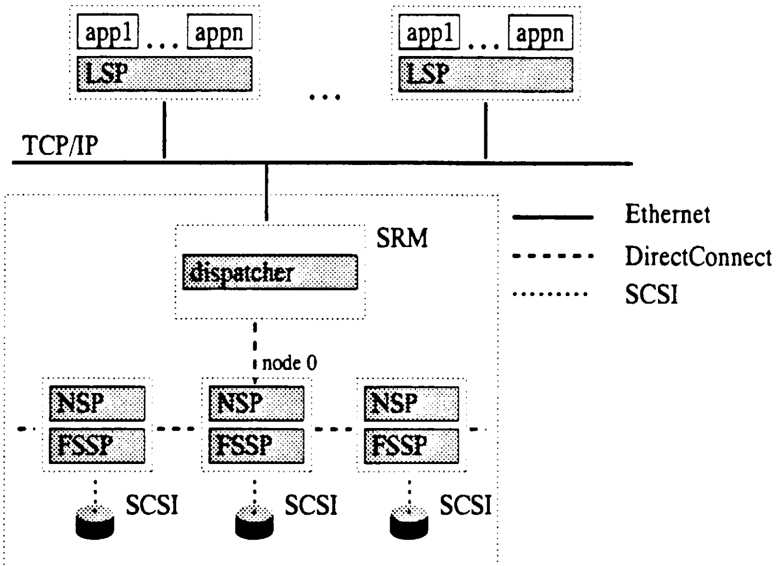


Figure 3: *PARABASE overall process architecture*

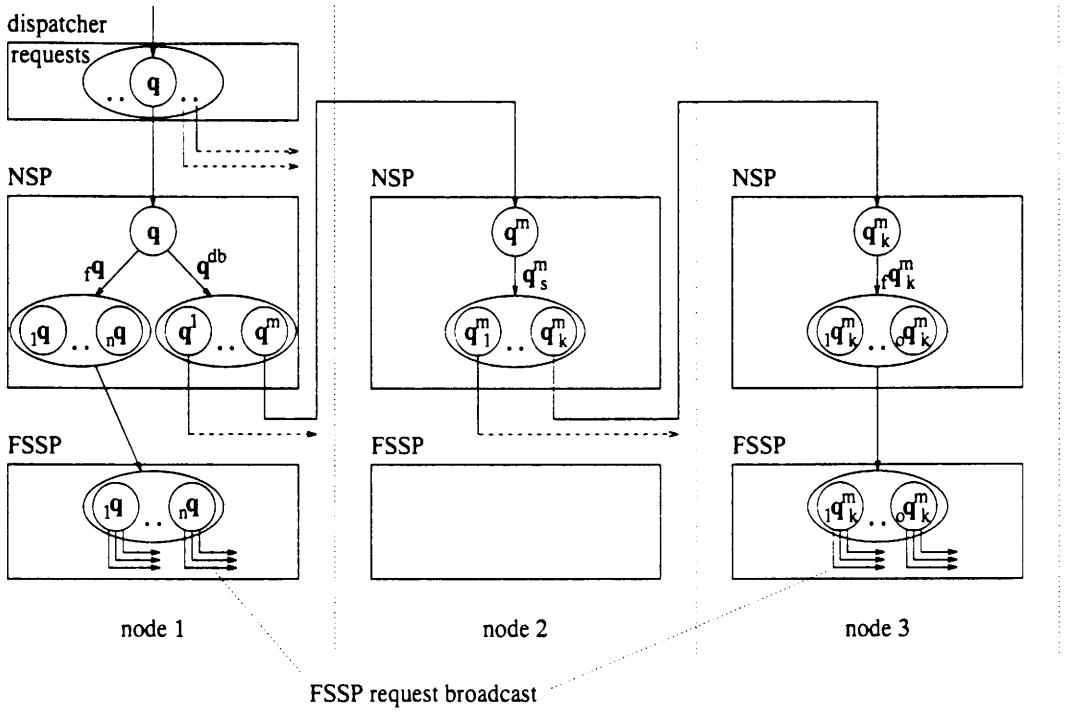


Figure 4: *Stepwise decomposition and parallel processing*

In the following, those two categories of query chunks are called *FSSP chunks* and *NSP chunks* respectively. If $\mathbf{q}$ denotes a particular query request, by convention the sets ${}_f\mathbf{q} = \{ {}_1\mathbf{q}, .. {}_n\mathbf{q} \}$ and $\mathbf{q}^{db} = \{ \mathbf{q}^1, .. \mathbf{q}^m \}$ shall denote the corresponding FSSP chunks and NSP chunks. All members of ${}_f\mathbf{q}$, i.e. all query chunks corresponding to file system calls, are passed to the local FSSP. Basically, those chunks are broadcasted and processed in parallel by all operational FSSPs. A detailed description of this level of parallel processing can be found in Section 4. An additional level of decomposition and parallelisation takes place for all members of $\mathbf{q}^{db}$. In a first step, these query chunks are spread over all operational NSPs in such a way that each $\mathbf{q}^i$ is assigned to one responsible NSP. In a second step, the NSP responsible for a particular $\mathbf{q}^i$ decomposes, if possible, the query chunk into a set of sub-chunks. In the following, this set is denoted by $\mathbf{q}^i_s = \{ \mathbf{q}^i_1, .. \mathbf{q}^i_k \}$. Consequently, all members of $\mathbf{q}^i_s$ are distributed over all operational NSPs and processed in parallel. Finally, each member of $\mathbf{q}^i_s$ causes file system calls upon execution. These local requests issued in order to process sub-chunk $\mathbf{q}^i_j$ are denoted by ${}_f\,\mathbf{q}^i_j$. Figure 4 illustrates this system of stepwise decomposition and parallel processing.

Considering the process of collecting and assembling the results of query chunk execution yields a bottom-up schema reflecting the decomposition schema described above. Each local FSSP collects the results for all members of ${}_f\mathbf{q}$ which have been executed in parallel by all other FSSPs. Those results are passed continuously to the corresponding NSP on the same node.

Quite similar, each NSP responsible for a query sub-chunk, say $\mathbf{q}^i_j$, issues appropriate file system calls, collects the corresponding data from the local FSSP, executes the sub-chunk and passes the subresr'lt to the NSP responsible for the query chunk $\mathbf{q}^i$. The data resulting from the overall execution of the query chunk $\mathbf{q}^i$ is passed to the NSP

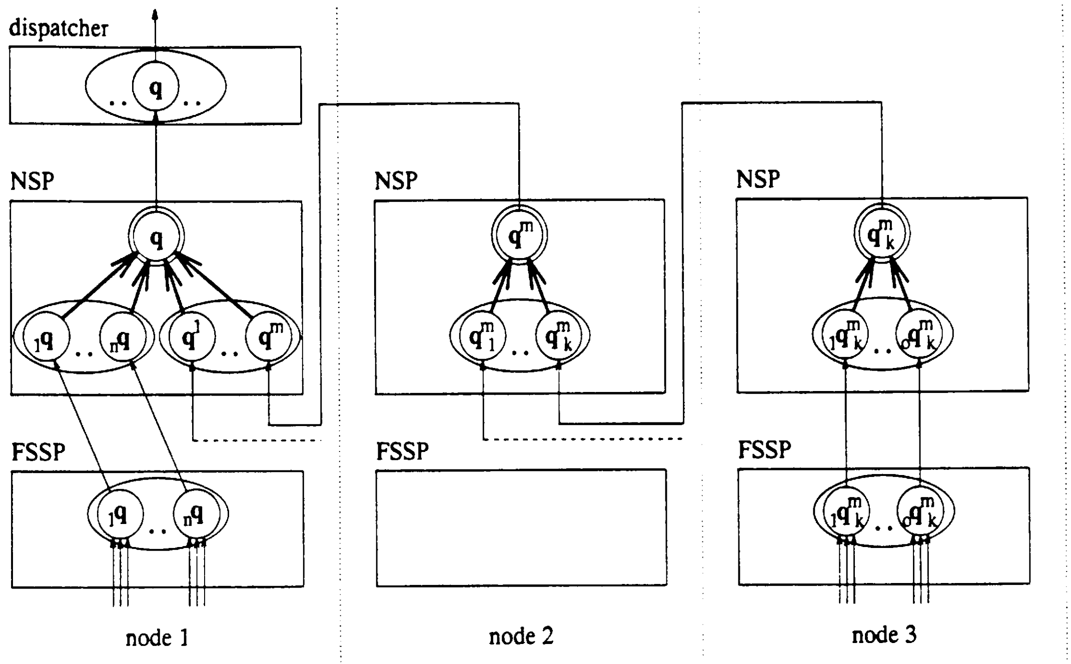


Figure 5: *Data collection and delivery*

responsible for the original query request **q** which in turn passes the data to the LSP responsible for the remote data delivery to the application. Figure 5 describes this collection schema.

The parallel processing of query chunks corresponding to file system calls, i.e. the processing of FSSP requests, is described in Section 5. However, recalling the multi-level concept towards query parallelisation as depicted in Figure 4 and 5, any query in the PARABASE architecture is handled in 4 levels of parallel processing (Table 1).

After this brief outline of the parallel query processing policies, we focus on the non-standard file system used to execute the FSSP chunks mentioned above. In a first step, the rationale for the design and implementation of a non-standard file system (see [Cho85a] for an earlier example) is given. Subsequently, the technical characteristics like data structures, data distribution policies and communication structures are described.

4. Rationale and Basic Data Structures for a Non-Standard File System

Standard UNIX file systems as well as state-of-the-art parallel file systems (see [Pie89a]) maintain *flat files*, i.e. unstructured byte strings held on mass storage devices. Common access primitives on such files are *read*, *write* and *seek*. The operating system supports atomic data transfer actions for continuous byte segments belonging to files. In other words, each file system read or write call issued by an application is intended to transfer a certain amount of uninterpreted data from a mass storage device into the application address space or vice versa. In most cases, the continuous byte segments to be transferred are specified as a number of bytes relative to a so called *file pointer*. The crucial point is that all the data *have to be uninterpreted*, i.e. without any structure or semantics, as far as the file system itself is concerned. In any other case, the flexibility of the data type *file* and the general usability for all kinds of applications would vanish.

This type of mass storage subsystem is well established and absolutely sufficient for non-database environments, especially for numerical computing. However, data intensive applications which have to rely on high-performance persistent storage management subunits (e.g. database management systems) reveal the inherent weaknesses of flat file systems very quickly. In particular, there is a strong need for access operations acting on *tuple sets* (or even on *object sets*, see [Mos90a]) instead of classical access operations acting on continuous byte segments. Basically, a data intensive application issues mass storage requests for tuple sets fulfilling certain logical conditions defined over certain attributes of the stored tuples. A file system designed to execute such requests with reasonable performance has to include two key features, namely internal (in the sense of tightly integrated) multikey indices and parallel request processing. The former supports fast attribute-symmetric search operations whereas the latter helps to bypass the ever present disk I/O bottleneck.

Consequently, the design of a non-standard persistent storage management system at the operating system interface requires a decision whether the persistent storage management system should use internal index structures *on top of the common flat file system* or *instead of the flat file system*. The second alternative implies a complete logical bypass of the original file system which actually ends up in a physical replacement in most cases since the partitioning of mass storage devices for different file systems seems to be rather unattractive for various reasons. Some basic performance considerations favour the

Level 1	**High-level dispatching**	
	actor (location)	Dispatcher (SRM)
	actions	• System load dependent distribution of incoming query requests (interquery parallelisation), i.e. assignment of each query request $\mathbf{q}$ to a responsible NSP
Level 2	**Parsing, partitioning and chunk distribution for query request q**	
	actor (location)	NSP responsible for $\mathbf{q}$ (node)
	actions	• Query request parsing • Parse tree analysis and isolation of independent query chunks $\mathbf{q}^1$ .. $\mathbf{q}^m$ (NSP chunks) and ${}_1\mathbf{q}$ .. ${}_n\mathbf{q}$ (FSSP chunks) • Distribution of isolated NSP query chunks (intra-query parallelisation), i.e. assignment of each query chunk $\mathbf{q}^i$ to a responsible NSP • Propagation of isolated FSSP query chunks to the local FSSP
Level 3	**Partitioning and sub-chunk distribution for NSP chunk $\mathbf{q}^i$**	
	actor (location)	NSP responsible for $\mathbf{q}^i$ (node)
	actions	• Isolation of independent query sub-chunks ${}_1\mathbf{q}^1$ .. ${}_m\mathbf{q}^n$ • Distribution of isolated query sub-chunks, i.e. assignment of each query sub-chunk ${}_j\mathbf{q}^i$ to a responsible NSP
Level 4	**Broadcasting and parallel processing of FSSP chunk ${}_k\mathbf{q}$**	
	actor (location)	FSSP responsible for ${}_k\mathbf{q}$ (node)
	actions	• Production of an request broadcast for all operational FSSP • Management of FSSP subresults produced in parallel

Table 1: *The 4 levels of parallel processing in PARABASE*

second alternative, even in spite of the need for additional development work. Intuitively, each additional layer in a persistent storage management system consumes a certain fraction of the overall system power, therefore the integration of the basic data storage functionality and of the index maintenance functionality yields significant performance improvements.

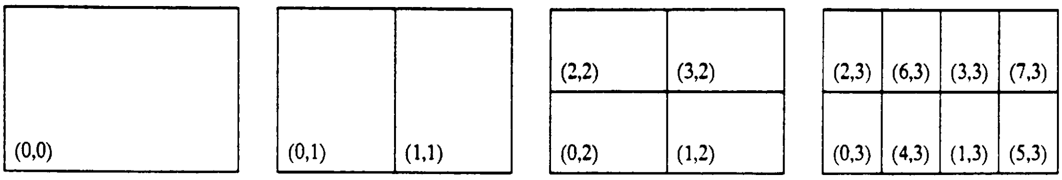
Consequently, the new file system is meant to replace the current flat file system (in particular the CFS) in case of data intensive applications. It has to provide flat file system capabilities as well as the tuple set capabilities outlined above. Fortunately, CFS source code has been already supplied by intel, therefore an integration of DiNG file and flat file functionality does not end up in too much additional effort at the moment.

Following from the above, a file system prototype based on distributed and nested grid files (called DiNG files in the sequel, see [Wit91a] or [Mue91a] for details) has been implemented which supports tuple insert and tuple delete operations as well as parallel exact match, partial match and range queries quasi at system call level. Distributed and nested grid files are a multikey index structure designed for mass storage subsystems on shared nothing MIMD machines, i.e. an index structure which allows for parallel queries against key attribute sets. Prior to the description of the file system, a few words about the underlying basic data structure, i.e. nested grid files as presented in [Fre87a] or [Fre89a], seem to be appropriate.

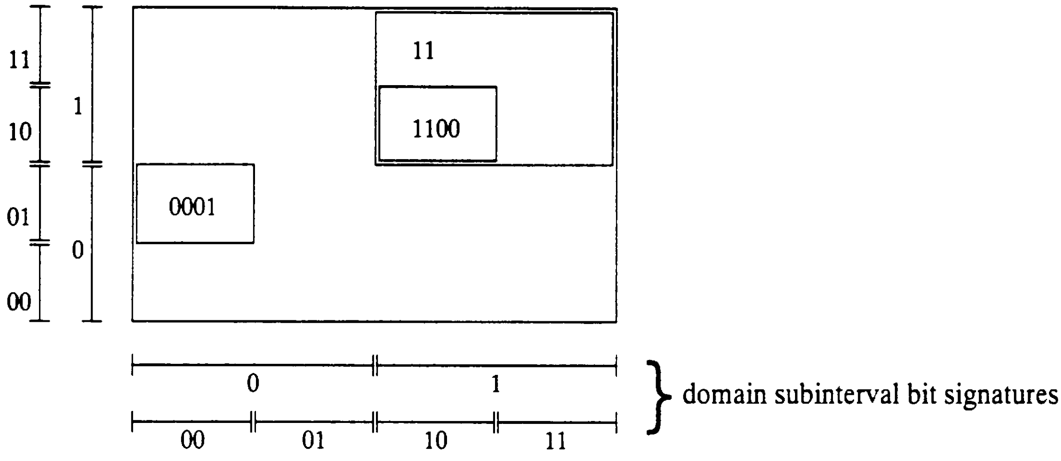
The key idea common to all grid file design approaches is the interpretation of n-tuple as elements of an n-dimensional space. This space, called data space in the sequel, has to be successively partitioned into smaller subspaces as the number of tuples increases. The resulting set of smaller subspaces used to give a partitioning of the initial data space has to be mapped to a totally ordered set, namely the disk block address space. This is to ensure that each relevant subspace of the n-dimensional data space corresponds to one physically transferable storage unit, i.e. a disk block, since any possible n-tuple has to be stored in one of the allocated disk blocks if passed to the mass storage subsystem for insertion. In the original grid file design (see [Nie84a]), the geometric contents of any two subspaces have to be disjoint. With respect to a reasonable directory expansion behaviour in case of non-uniformly distributed or correlated raw data, the nested grid file approach relaxes this condition to some extent. The relaxed partitioning condition reads as follows: if any two hyperrectangles intersect, one has to enclose the other.

The resulting partitioning schema, which in turn determines the geometric shape of the subspaces, is conceptually simple. Basically, it is a buddy system with subsequent *binary* partitioning of the initial data space. All hyperrectangles are created as a result of alternating and cyclic binary domain splitting as depicted for the 2-dimensional case in Figure 6(a) below. Each hyperrectangle is identified by a pair of values, namely (***RegionNumber***, ***SplitLevel***). Region numbers are created by successive bit interleaving of the domain subinterval bit signatures. The interleaving sequence is given by the cyclic domain split sequence, i.e. 1.bit of domain$_1$, 1.bit of domain$_2$ $\cdots$ 1.bit of domain$_n$, 2.bit of domain$_1$, 2.bit of domain$_2$ $\cdots$ 2.bit of domain$_n$ and so on. Figure 6(b) illustrates the bit interleaving concept.

The physical directory structure is implemented as height-balanced multiway tree. Figure 7 shows a nested grid file, both ir the geometrical representation and in the data structure oriented representation. The



6a: *Successive data space partitioning*



6b: *Identifying hyperrectangles (region number and split level)*

Figure 6: *Data space partitioning and subspace identification*

file in Figure 7 contains two subspaces stored in the directory and two tuples stored in data buckets.

A search for tuple x_1 in the file of Figure 4 includes a directory traversal to find the subspace identifier corresponding to the enclosing block region. The data bucket reference attached to block region identifier provides access to the data bucket in which x_1 is actually stored. However, considering x_2 reveals that the smallest enclosing subspace has to be found. Actually, x_2 is contained in both subspaces but stored in the data bucket referenced by the directory entry of subspace (3,2).

This symmetric multikey approach is in contrast to B^+-tree approaches, which either favour certain attributes or attribute combinations or force a database administrator to use an unacceptably large number of index

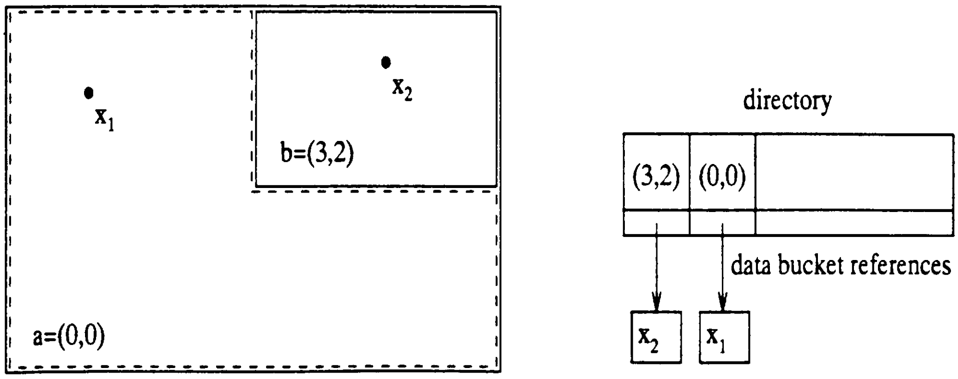


Figure 7: *Nested grid file example*

structures for one single file. In particular, $2^n - 2$ B^+-trees would be needed for a file with **n** key attributes. This figure corresponds to the cardinality of the powerset of **n** minus 1, since the empty subset of the **n** attributes has to be excluded. Additionally, the attribute *sequence* in a compound single-key index is not even considered in this figure although it is of prime relevance for any query optimiser.

5. The DiNG File System – Structure, Data Distribution and Communication

After the motivation for the design and implementation of a non-standard file system and the brief outline of the underlying basic data structure, the technical characteristics of the resulting parallel file system can be described. In particular, we elaborate on the actual file system structures, i.e. on superblock and i-node maintenance, on the current data distribution policies and on the inter-node communication structure of the FSSP farm with regard to NSPs acting as client applications.

5.1. The File System Structure

The basic mass storage block allocation, handling and administration schema of the DiNG file system is conceptually simple and similar to the UNIX mass storage block administration. At that level, the main differences between a standard parallel file system and the DiNG file system stem from the separate handling of data blocks and index block. Free space administration is done with bit map structures, i.e. the FSSP control process maintains a super block, an i-node bit map, a bit map for data and index block, a list of i-nodes and a list of block containing data blocks as well as index blocks. Since data blocks and index blocks are handled by different subprocesses of the file system, a differentiation between the two block classes is necessary even at this lowest level of block administration. At the moment, the block buffer cache employs a standard hash table based LRU displacement strategy (see [Tan87a] for example). Other displacement strategies are currently considered, however, a detailed discussion of buffer cache considerations would be beyond the scope of this report. Readers interested in this topic may refer to [Mue91a].

At this point, the internal process structure of the FSSP (depicted in Figure 8) has to be described. However, the discussion of the control flow and the data flow between the subprocesses and the NSPs in case of tuple insert requests or query requests is delayed to Subsection 5.2.

Each NSP acting as DiNG file system client has to use a FSSP *client library* which is responsible for correct protocol handling and appropriate message formats. This library provides access to the local file system server, i.e. the FSSP located on the same node as the NSP. All FSSP requests issued by NSPs are initially handled by the FSSP local to the requesting NSP. Subsequent parallel processing is transparent to the NSP, since requests are passed to and results are obtained from the local server.

The FSSP client library has a counterpart in the FSSP, namely the *client message handling library* used by the FSSP *control process*. The control process uses a second message handling module, namely the control message handling library for all internal communications with other control processes on different nodes. Besides all coordination and control tasks in the context of insert, delete and query request handling, the control process manipulates directly all index blocks. In other words, the control process executes all index searches and passes the resulting data block numbers subsequently to the *fetch & send process*, which is responsible for data block handling. A third process, the so called *get & send* process is responsible for all query result deliveries. It collects all query subresults from all fetch&send processes, i.e. from the local f&s as well as from all other f&s on different nodes, and passes the collected data to the local NSP.

The data distribution policy can be described as round robin tuple distribution. Each insert request `INSERT <tuple> INTO <file>` is passed to the local control process and triggers a lookup operation in the corresponding i-node which yields the appropriate node number for the next insert into `<file>`. In particular, if `last_FSSP(<file>)` denotes the number of the FSSP which received the last tuple previously inserted into `<file>` and if **p** denotes the number of operational FSSPs, the expression (`last_FSSP(<file>)+1` modulo **p**) yields the number of the FSSP which has to insert `<tuple>`. If the calculated FSSP number refers to the local node, the correct data block number is determined, the tuple together with the data block number is passed to the f&s process and finally inserted by the f&s process. If the calculated FSSP number refers to a different node, the control process passes the tuple to the corresponding control process which takes the appropriate steps for local insertion.[†]

As a result of this distribution schema, each DiNG file is spread over all available nodes. In other words, each logical DiNG file as seen from a client's point of view consists of a number of physical DiNG files. The contents, i.e. the tuple set of one logical file equals the union of all corresponding physical files.

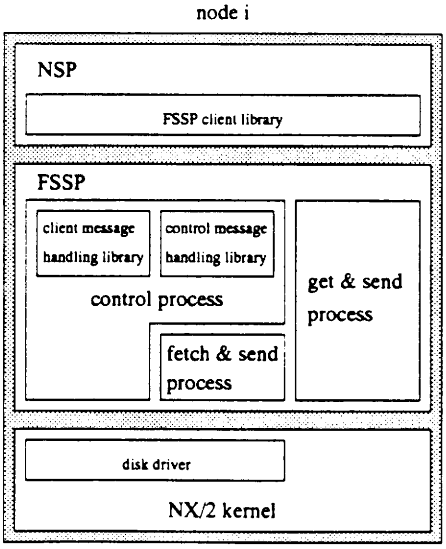


Figure 8: *FSSP subcomponents*

[†] All further implementation details of the distribution process (e.g. the node counter update per file) are omitted due to space considerations.

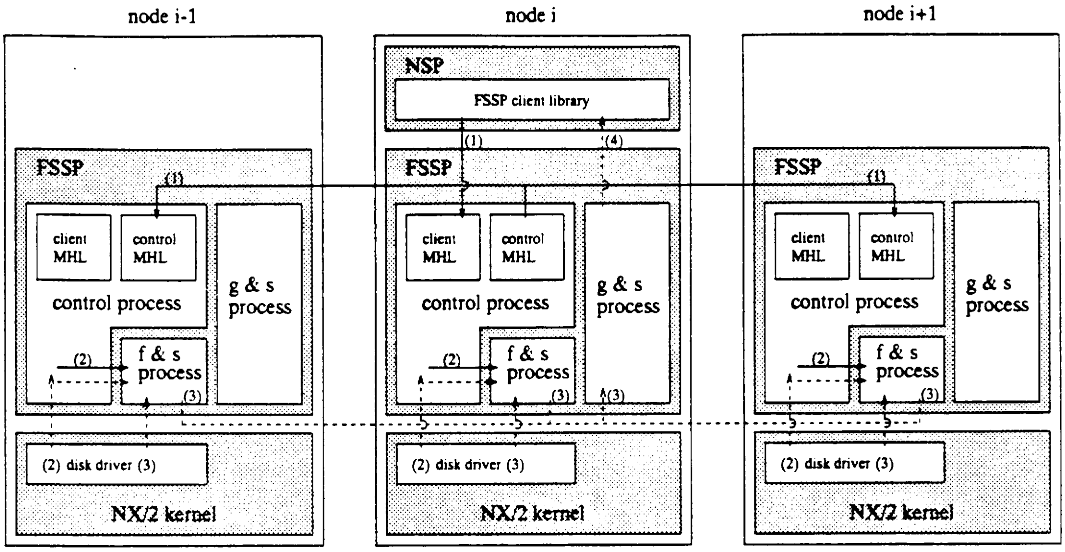


Figure 9: *Query execution*

5.2. Control Flow and Data Flow in the File System

Control flow and data flow in the context of query executions represent probably the most interesting parts of the interprocess communication in the file system. The following description refers to the process structure discussed in Subsection 5.1 and to Figure 9, which depicts the situation in case of an FSSP query execution.

A particular query request on node **i** is launched via an FSSP library call, received by a client message handling function and passed to the local control process on node **i**. This responsible control process sends the query request to all other control processes (phase 1). All control processes perform an index search on their local part of the DiNG file (see above) in parallel and extract all relevant data block numbers from the index (phase 2) and pass these numbers to the corresponding f&s processes. All f&s processes fetch the appropriate data blocks in parallel and send the retrieved data to the g&s process on node **i** (phase 3). As soon as the g&s buffer area on node **i** is filled, the g&s process broadcasts some kind of `<stop_transmission>` signal to all f&s processes and engages in the data delivery to the client process (phase 4). As soon as the g&s buffer contents has been delivered, a `<restart_transmission>` signal is broadcasted by the g&s process. The protocol iterates in phase 3 and phase 4, until all the data qualified by the query request has been delivered.

6. Conclusions and Near Future Research Agenda

The PARABASE research project aims at shared nothing MIMD architectures to be used as high performance DBMS servers in federated environments. Central parts of the PARABASE project are a multilevel approach towards query parallelisation and a high performance parallel file system, namely the DiNG file system. The file system has been tailored for DBMS needs and includes a low level query processing policy based on a particular multiattribute search structure, namely on nested grid files. The index in^formation is stored in a balanced multi-way tree similar to a B^+-tree.

The near-future research agenda contains various open problems concerning different tuple distribution policies, some considerations about bandwidth balancing between links and SCSI-devices with regard to query processing, a number of topics in the context of replicated data maintenance and a number of additional DBMS oriented problems like concurrency control (see [Lue92a] for a first non-distributed solution). From a different point of view, a considerable research effort is needed in order to investigate the relevance of *throttling* and *speculative work* for DBMS topics. In particular, certain throttling policies in case of heavy join processing (i.e. *not* engaging *all* processing nodes for *all* joins) seem to be promising.

References

[Cho85a] H. -T. Chou, D. J. DeWitt, R. H. Katz, and A. C. Klug, "Design and Implementation of the Wisconsin Storage System," *Software – Practice and Experience* **15**(10) (1985).

[Cop88a] G. Copeland, "Data Placement in Bubba," in *Proc. ACM SIGMOD Conf. on Mangement of Data*, ACM Press, Chicago (1988).

[DeW86a] D. J. DeWitt, "GAMMA – A High Performance Dataflow Database Machine," in *Proc. of the 12th VLDB Conf*, ACM Press (1986).

[Fre87a] M. W. Freeston, "The BANG file: A new kind of grid file," in *Proc. ACM SIGMOD Conf*, ACM Press, San Francisco (1987).

[Fre89a] M. W. Freeston, "Advances in the design of the BANG file," 3rd Int. Conf. on Foundations of Data Organisation and Algorithms, Paris (1989).

[Fri90a] O. Frieder, "Parallelism – Using the Right Tool for the Right Job," in *Proc. of PARBASE-90*, IEEE Computer Society Press (1990).

[Fri90b] O. Frieder, "Multiprocessor Algorithms for Relational Database Operators on Hypercube Systems," *IEEE Computer* **23**(11) (1990).

[Hon90a] W. Hong and M. Stonebraker, "Parallel Query Processing in XPRS," Tech.Rep. UCB/ERL M90/47, Electronics Research Laboratory, University of California at Berkeley (1990).

[Kim91a] W. Kim, "A Distributed Object-Oriented Database System Supporting Shared and Private Databases," *ACM TOIS* **9**(1) (1991).

[Lue92a] G. Luef and T. A. Mueck, "Concurrent Operations in Balanced and Nested Grid Files," in *Proc. of the 6th Int'l Working Conference on Scientific and Statistical Database Management*, Ascona (1992).

[Mos90a] J. E. B. Moss, "Design of the Mneme Persistent Object Store," *ACM TOIS* **8**(2) (1990).

[Mue91a] T. Mueck and M. Schauer, "Sorting in the BANG file," Tech.Rep. #109, Dept. of Information Systems, University of Vienna (1991).

[Nie84a] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM-TODS* **9**(1), pp. 38-71 (1984).

[Pen92a] M. A. Penaloza and E. A. Ozkarahan, "Parallel Algorithms for Executing Join on Cube-Connected Multicomputers," in *Proc. of the 8th Conf. on Data Engineering*, IEEE Computer Society Press (1992).

[Pie89a] P. A. Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem," 4th Conf. on Hypercubes, Concurrent Computers and Applications, Pasadena (1989).

[Tan87a] A. S. Tanenbaum, *Operating Systems,* Prentice Hall, Englewood Cliffs (1987).

[Wit91a] J. Witzmann, "The DING file system," Master Thesis, Dept. of Information Systems, University of Vienna (1991).

Distributed Object Management within a Loosely-Coupled Repository Environment

Alexander Schill

University of Karlsruhe
Germany
schill@ira.uka.de

Abstract

This paper describes a system to manage objects in a distributed environment. The basic configuration consists of a set of loosely interconnected object repositories storing strongly typed objects. Objects are collections of data and associated operations and can have complex, nested structures. They can be accessed remotely and can move dynamically between repositories. Objects and object classes are only shared in a limited way, i.e. we do not require a global schema as in the distributed database model.

We present several alternative mechanisms to locate mobile objects, and to implement object mobility. Both aspects are shown to be closely related. Finally, we illustrate the application-level functionality of our approach by describing a distributed office procedure facility on top of the basic system.

1. Introduction

This paper describes the architecture and implementation of a distributed object management system. The basic system configuration consists of a set of loosely interconnected object repositories storing strongly typed objects. Objects are collections of data and associated operations and can have complex, nested structures. They can be accessed remotely and can move dynamically between repositories.

This basic model provides important advantages as compared to centralized solutions and to other distributed system approaches. It allows a natural modeling of applications in decentralized organizations. For example, the major organizational units of a corporation can be modeled as coarse-grained objects with a large population of fine-grained data objects circulating between them in order to do data processing. The objects can be statically or dynamically mapped to network nodes of a distributed environment; nodes can operate relatively autonomously. As opposed to distributed database systems, the different nodes may perform their individual schema management and typically only share a limited amount of type and instance information. We do not require a global schema as required by the distributed database model. The performance of local processing can be improved by an appropriate initial object placement and by adapting the object distribution dynamically. As opposed to more conventional distributed systems, the distribution can be relatively transparent once a specific object placement has been determined. In particular, objects can be accessed and invoked in a location independent way. In our mobile environment, this is achieved by an extended forward addressing mechanism.

We present several alternative mechanisms for locating mobile objects and for implementing object mobility. Both aspects are shown to be closely related. We also illustrate the application-level functionality of our approach by describing a distributed office procedure facility on top of the basic system.

Conceptually, our approach is language-independent: we define a generic distributed systems service to manage a distributed object environment. Several object-based languages can exploit our facilities by adding special distribution-related object classes and modest language extensions. The only basic requirements of our approach are that objects are typed and have a global identity, that layout description information for object data structures is available, and that operation invocations on objects can be intercepted in order to perform distribution-related actions.

In summary, our major contribution is to design a generic approach to manage mobile objects in a persistent environment, to integrate the approach with supporting mechanisms, and to present a sophisticated application-level service which directly exploits our facilities. The paper is organized as follows. Section 2 discusses related approaches and outlines their implications for our work. The main Section 3 describes our approach to distributed object management. We also give an outline of an associated implementation structure that is the conceptual base of our current UNIX implementation, as well as of a former implementation in an IBM PC/Host environment. In Section 4, we describe our application-level office procedure service. Section 5 concludes with an outlook to future work.

2. Related Work

A number of existing approaches have been concerned with distributed object-based systems [Chi91a] and also with the distribution of object repositories. The *Emerald* system [Bla87a, Jul88a] supports fine-grained, mobile objects in a distributed environment. It offers operations to locate and move objects explicitly and makes object invocations location transparent. A specific feature is the ability to move objects even if they are currently being accessed. Moreover, mobile Emerald objects can contain a set of internal objects which cannot move independently; this allows for larger grains of mobility and for more efficient implementation of local invocations. A major prerequisite for such mechanisms has been the tight integration of the *Emerald* language and system. The *Amber* system [Cha89a] is a follow-on project and implements a distributed version of *C++* with comparable facilities. Most important, it supports both inter-node and intra-node parallelism to implement applications on a combination of tightly and loosely coupled multiprocessors. While the use of an existing language makes the approach very attractive, this also complicates the implementation of mobility features and requires significant restrictions concerning the use of memory pointers within mobile objects. There have also been several distributed extensions to the *Smalltalk* system, for example [Ben87a, Cor90a]. These approaches enable Smalltalk objects to move between workstations and to be invoked transparently.

The approaches summarized above only support transient objects and do not incorporate persistent storage. As opposed to that, the *Hermes* system [Bla89a] supports persistent mobile objects. A subset of the system nodes serve as storesites, i.e. as persistent object repositories. Objects can move freely between nodes with volatile memory and can also checkpoint their state and log state changes at a selected storesite. An object can be reassigned to a different storesite dynamically. Special protocols to locate objects using storesites and to manage objects in persistent storage are part of the approach. Likewise, the *Comandos* system [Kra90a, Ber90a] also supports persistent objects and object mobility. A new language named *Guide* is introduced to implement application programs and is integrated with the runtime system. A specific feature of *Comandos* is the support of object clusters with dynamic cluster management facilities [Kra90a]. This way, coherent units of local execution can be configured at runtime. A proposal to support a highly available distributed object repository is outlined in [Lis90a]. In addition to basic object distribution facilities, object replication and flexible language integration is supported by the described design.

We have borrowed basic ideas from many of these approaches to support object mobility and to locate objects in a distributed environment. We provide major extensions by introducing a location algorithm which combines different distributed runtime mechanisms. Moreover, we provide additional features that address access to system objects. We also provide a concrete application service which takes advantage of our basic system.

3. Managing Objects in Distributed Repositories

This section first introduces the overall architecture of our approach. It presents the object mobility features and the mechanisms to locate mobile objects. Finally, our implementation is outlined.

3.1. System Architecture

Our basic system architecture consists of a set of object repositories and workstations which are interconnected via a network.[†] An object is basically a collection of data and associated operations. The physical network topology is transparent at our level of discussion; that is, we assume a logically fully interconnected network where each node is reachable from any other. Access to a remote object, however, is assumed to be about two orders of magnitude more expensive than local access in terms of response time. Figure 1 shows an example topology with a number of interconnected repositories and workstations, and stored objects with logical interobject references. Workstations can fetch and cache copies of objects stored in repositories and can store updated copies back in the repository later. The cache management is based on conventional pessimistic locking; an investigation

[†] The repository structure is only supported by the IBM environment; our current UNIX implementation only consists of workstations.

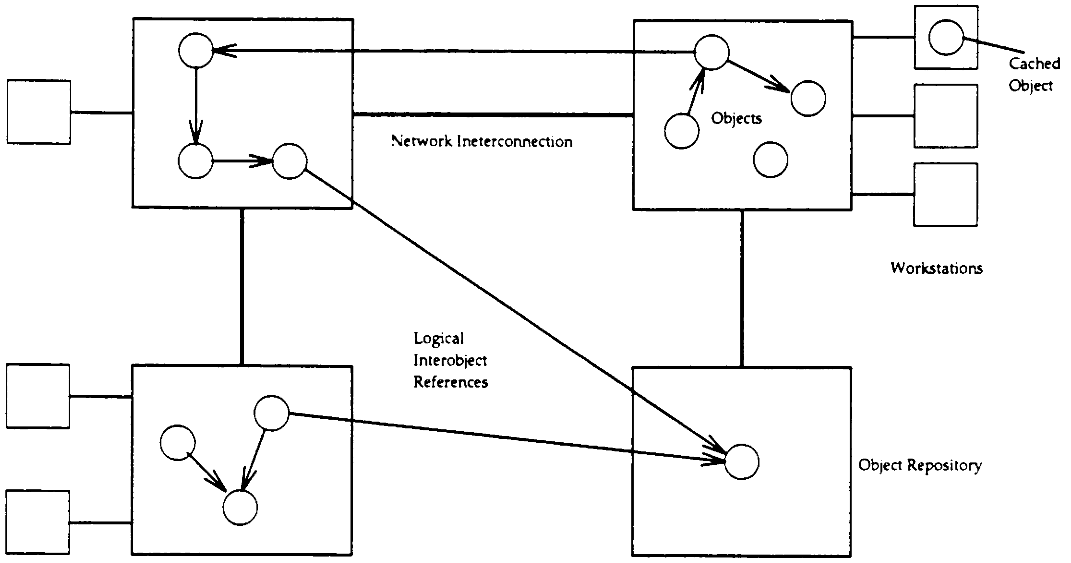


Figure 1: *Example of a distributed repository/workstation topology*

of alternative optimistic cache management policies is beyond the scope of the paper.

All objects are typed, i.e. each object belongs to a specific object class. The class information is partially shared and is available to both repositories and workstations. Operations on objects are executed on cached copies within workstations; caching is performed transparently on demand by mechanisms on top of our approach. Objects can be accessed and fetched in a location independent way; a prerequisite is that the invoker has a valid reference to the object. Each object has a globally unique identifier which never changes; it is generated by concatenating the identifier of the node where it is created with a number which is unique within the scope of that node. Objects can refer to each other locally and remotely using these identifiers. In addition, objects can dynamically move between repositories in order to increase locality of reference. Details of migrating and locating an object are discussed below. To summarize the functionality, the signatures of the most basic interface operations are briefly explained in Figure 2 using a simple notation (<operation> (<parameter_types>) -> <return_type>). We assume several predefined types for operation parameters which are passed by reference. In particular, *Location* is a descriptor for a node object, *Object* is the superclass of all objects, *Status* holds operation status indications, *Class* is a descriptor for an object class, *Operation* is a descriptor for an operation to be performed on an object of a particular class, *ObjectName* is simply a string, and *ParameterSet* and *AttributeSet* are data structures holding tags and values of parameters or attributes, respectively. Beside the conventional operations to *create* and *invoke* objects, we offer operations to *locate*, *move*, and *copy* an object remotely. In addition, there are operations to *fetch* objects into a workstation cache and to *flush* them back to a repository. *Move*, *Fetch*, *Flush*, and *Invoke* can also be applied to a set of objects at once.

Moreover, an operation to resolve object names and attributes returning an object identifier is supported (*FindObject*). Names and attributes to be resolved can be abbreviated by using wildcards in order to perform a fuzzy search. As a consequence, the operation can also return a set of objects which match the query. The invocation-related operations are hidden by additional mechanisms on top of our approach by a remote

```
Locate (Object) -> Location           // locates a given object
Move (Object, Location) -> Status     // moves an object to a new location
Copy (Object, Location) -> Object     // creates a copy of an object and moves it

Create (Class,ParameterSet) -> Object // creates a new object with initialization parameters
Delete (Object) -> Status             // deletes an object
Fetch (Object) -> Status              // fetches a copy of an object to the local site
Flush (Object) -> Status              // writes the copy back to the original

Invoke (Object, Operation, ParameterSet) -> Status
                                      // invokes an object operation location independently
FindObject (ObjectName, AttributeSet) -> Object
                                      // retrieves an object reference from the name service
                                      // based on the given name and attributes
```

Figure 2: *Interface operations for basic object management*

invocation manager. The other operations are exposed to applications via an application programming interface.

The described architecture is suitable for a variety of decentralized applications, for example of the office automation area. It supports persistent data management which is crucial for most real applications like forms processing or distributed decision making. As opposed to centralized or distributed databases, the approach emphasizes site autonomy by requiring only a limited sharing of class information. This way, a high degree of flexibility and efficiency can be achieved for a wide class of applications, especially if most processing is done locally. However, there is still the need to access limited amounts of data on remote nodes and to move objects from time to time.

In the following, we outline a number of alternatives to manage objects within the described environment and motivate our concrete design decisions.

3.2. Locating and Migrating Objects

As described above, objects are referenced via globally unique identifiers. However, to make object access location independent and efficient in a mobile environment, additional mechanisms are required.

Our basic approach to refer to remote objects is to introduce proxy objects. A proxy translates a unique object identifier into an internal location hint and possibly a location dependent identifier. This way, a local object can query the appropriate local proxy to gain access to a referenced remote object. Several alternatives are possible concerning the creation of proxies and concerning the update of their location information and are discussed below.

Access to a moved object

When an object moves from site A to site B, it must still be accessible via references from site A and from other remote sites. This access can be done via a broadcast request to all sites; eventually, node B will respond if it is available. However, this solution is very inefficient in large networks and can introduce significant overheads even in small local area networks. Therefore, a proxy must be installed at node A which refers to node B. We further distinguish between two internal solutions (see Figure 3). If objects are referred to locally via an object table, the appropriate table entry holds the physical address of the object. When the object moves, the address can then be directly replaced by the proxy information or by a pointer to it (alternative *a* in

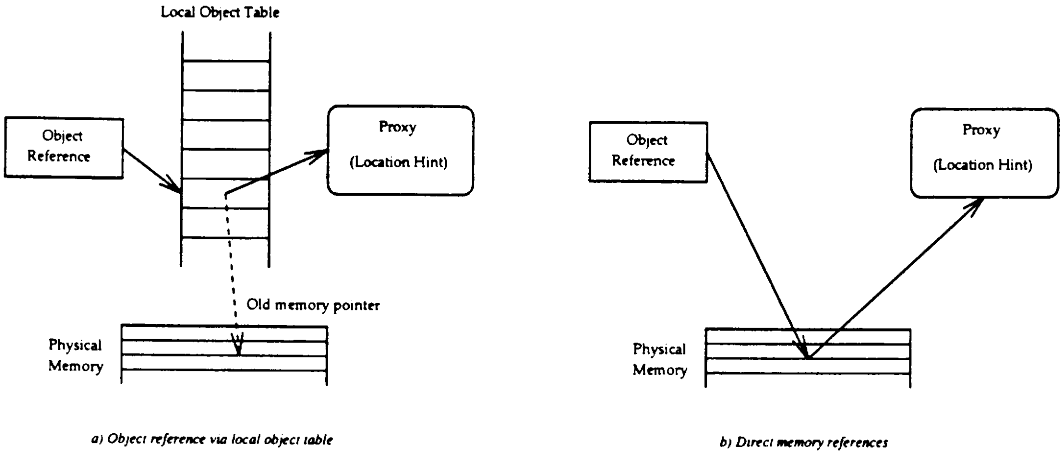


Figure 3: *Installation of a proxy for a moving object*

the figure). The physical storage used by the object at site A can be reused for other data. If objects are referred to by direct memory addresses, the proxy information must be inserted at that address (alternative *b* in the figure).

References to the moved object from an other remote site, C, will be based on a proxy at that site pointing to the location A.† It is not necessary to modify them as C can access the object by contacting site A which will forward the request to B based on B's proxy. We call this mechanism *forward addressing without immediate location update*. It improves the performance of a migration but makes further remote access more expensive. However, the location hint can at least be updated after the next access by returning current location information to C. The scheme makes access also less reliable: there may exist a whole chain of proxies after a set of migrations. If one node holding a proxy is unavailable, the object will not be accessible (except by using broadcast).

An alternative is *forward addressing with immediate update*. With this scheme, A will inform C about the new location of the object, i.e. B, immediately and C can adjust its proxy accordingly. However, this requires additional messages during the migration phase. It also requires backward pointers from an object to all proxies which currently exist for it. Both may not be feasible in large environments with frequently referenced objects. However, the performance and availability problems of following multiple proxies is avoided by this scheme. In summary, the immediate update improves object access but makes mobility and the maintenance of object references more expensive.

A third and more promising alternative is a combination of both: usually, forward addresses without immediate update are used. In addition, a moving object can optionally register its new location at its creating node and/or at a global name service. At these nodes, special proxies are maintained for it which are subject to immediate update. Access to the creating node is enabled as it is included in each object identifier (see Section 3.1) and is therefore known to all referencing objects.

† The object can also be accessed by contacting its creating node which is included in its object identifier as discussed below.

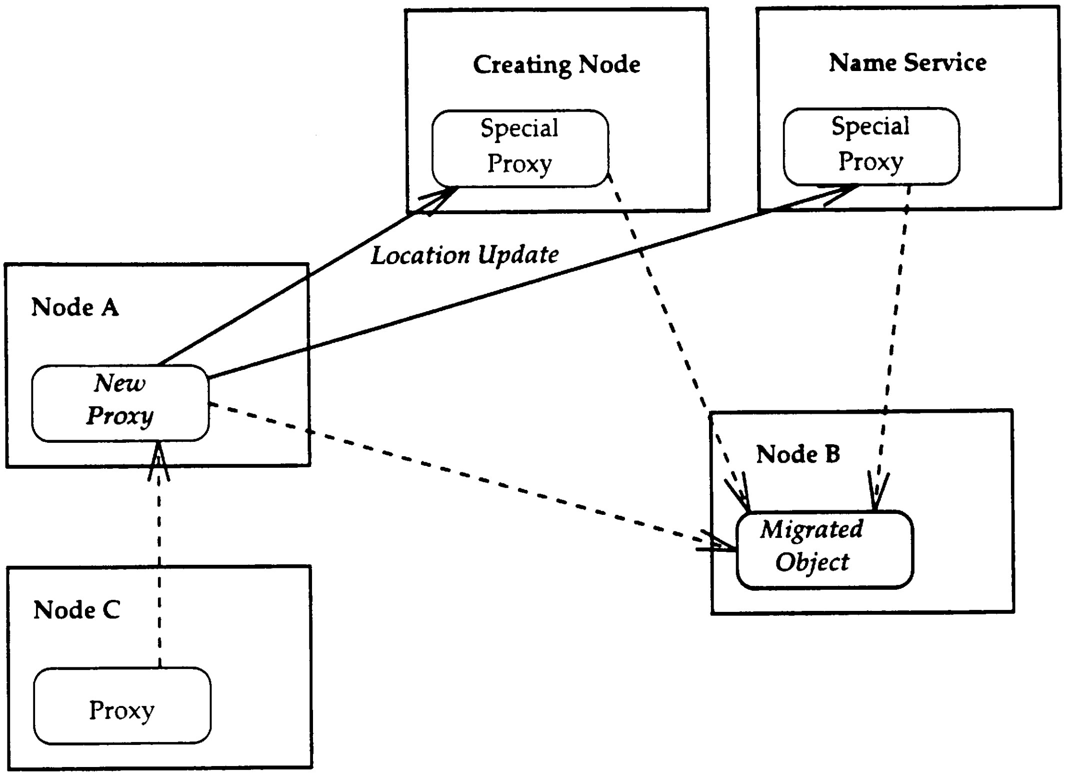


Figure 4: *Combined solution for remote proxy management*

An object is located by following forward addresses as long as proxy chains do not exceed a maximum length and as long as all intermediate nodes are available. Otherwise, the creating node and/or the name service are contacted and may yield the object's current location if it has been registered there. Depending on the estimated cost of accessing the creating node or the name service, the maximum size of a proxy chain to be followed can be adjusted (typical biases are expected to be below 10). We adopt this combined solution but provide the option of not informing the name service or the creating node in order to fine-tune an application. The combined solution is summarized in Figure 4, a proxy for the migrated object is installed at node A and the creating node and the name service are informed about the migration. They update the special proxies which then point to the new location, i.e. to node B.

In order to improve performance, location hints can also be cached directly within object references or within an attached intermediate data structure. This way, a referencing object can directly forward an invocation without having to examine the proxy. Moreover, such augmented references can be exported to other nodes easily without the need to install associated proxies there (see below). However, proxies are still required: the location hint within a reference may be completely outdated if the reference has not been used for a long time.

Finally, there may already exist a proxy for the moved object at the destination node. It is simply deleted and replaced by the actual object data. The internal implementation again depends on the local object access mechanism, i.e. whether a table indirection is used or not (see above).

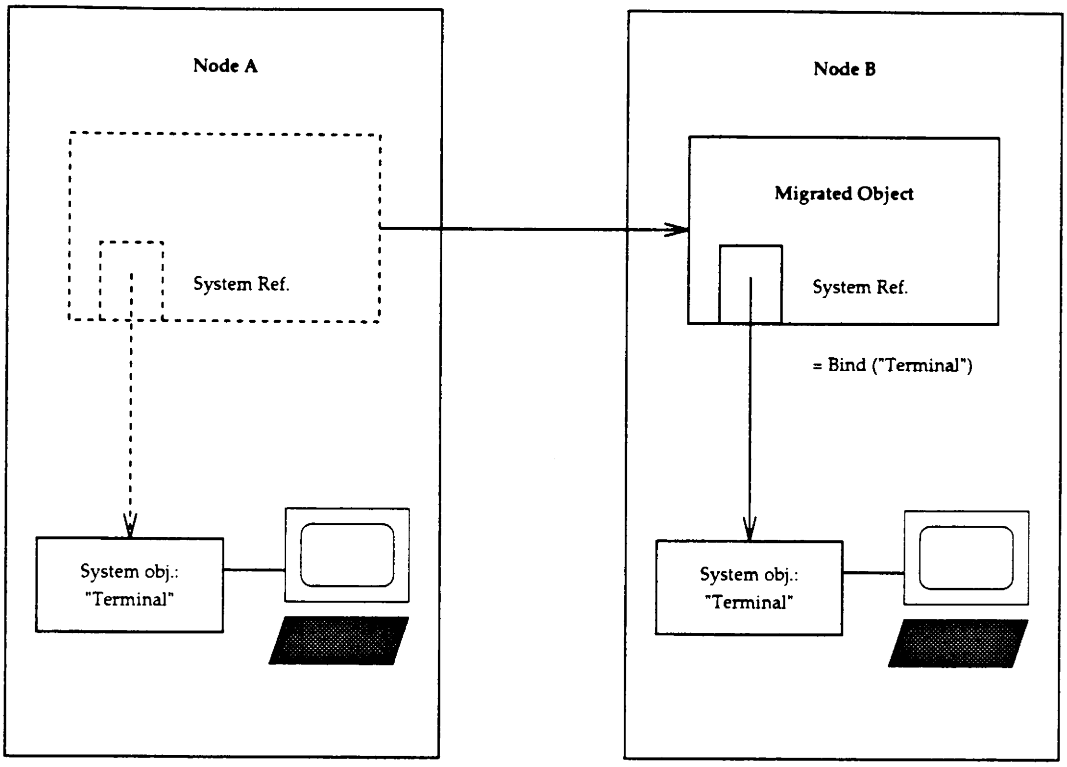


Figure 5: *New binding of imported references to local system objects*

Management of Imported References

In a mobile environment, the other fundamental question that arises is how to manage object references which are imported at a given node. Such reference import takes place at a site A if an object containing references moves to A, if an invocation passes reference parameters to A, or if an invocation returns references to A. There are again various solutions: if the imported references contain location hints, no proxies have to be installed for them immediately. The location hint can be used for the first access to a referenced object; after that, a proxy can be installed lazily in order to make further accesses more efficient. Even if no explicit location hint is available within references, the creating node within the object identifier can still be used as an implicit hint. Such a solution may even be recommended if it is not possible to identify all references originating from a moving object (e.g. if typed layout information is not available for the different fields within the object's data structure).

Alternatively, proxies can be installed immediately for all imported references. If a proxy or even the original object already exists at the given node, no additional installation is necessary, of course. This policy makes further access more efficient and reliable but makes migrations more expensive with respect to implementation and runtime performance. We selected this policy for its conceptual clarity and relative simplicity. However, it will be worth to investigate a combination of both approaches, too: proxies could be installed for all references imported via invocations but not for references originating from moved objects. The basic motivation is that object references passed as parameters are much more likely to be used for immediate further invocations. These references have to be marked explicitly with a special attribute.

```
const Bias = 5;
AddressType* Locate (Location* node, Object* object, int pathLength)
    { AddressType* address;
        if (Local (object)) return Address (object); // object is local

        if (pathLength < Bias) {                    // try forward address lookup
            if (LocationHint (object) != Null) {    // a proxy was found
                address = Locate (LocationHint (object), object, pathLength+1);
                if (address != Null) return address; // remote lookup was successful
            }
        }
        address = Locate (CreatingNode (object), object, 0);
        if (address != Null) return address; // lookup at creating node was successful

        address = Locate (NameService (), object, 0);
        return address; // finally: return result of name service lookup
    }
// Initial invocation:
objectAddress = Locate (thisNode, object, 0);
```

Figure 6: *Basic algorithm to locate objects*

Access to system objects

Another option is to rebind imported references locally instead of maintaining a pointer to the original object. This solution seems especially important for a variety of system objects which represent local devices like a terminal, a printer or a system queue, for example: it may be desirable to do input/output based on local devices instead of redirecting it to a remote site. System objects are assigned known logical names and references to system objects are marked explicitly. When such a reference is imported, the appropriate system object is searched locally based on the logical name and its object identifier replaces the identifier of the previous system object. The replacement can be performed immediately or lazily depending on the policy described above. The mechanism is illustrated in Figure 5: an object moved from node A to node B and had a system reference pointing to a terminal object at node A. The reference is bound to a different terminal object at node B via a logical name.

Summary: Algorithm to locate objects

As a summary, the basic algorithm to locate objects is given in Figure 6. It is given as the *C* function *Locate*. We assume several predefined types as described in Section 3.1. *AddressType* is a descriptor for an object's location and its location-dependent storage address at this location. Several macros are used: *Local(Object)* tests whether an object is local to the invoker's location by examining its object table entry, *Address(Object)* returns the address descriptor of an object if it is local, *LocationHint(Object)* returns an existing proxy for a remote object, *CreatingNode(Object)* extracts the creating node out of an object identifier, and *NameService()* returns a global reference to the name service.

If the location lookup is successful, the function *Locate* finally returns a location dependent address of an object, i.e. its current location and memory address. Its formal parameters are initially instantiated with the location of the caller, the identifier of the object to be located, and an initial zero search path length. The function first checks the local object table and returns the local object address if successful. Otherwise, the object table is checked for a proxy containing a location hint. If successful, *Locate* is invoked recursively at the node indicated by the hint.† If no proxy is available or the forwarding chain is longer than a given bias, we try to locate the object at its creating node. Our final attempt to locate it is to inquire the name service. The mechanism can be optimized by doing several inquiries in parallel and by employing a more flexible asynchronous message communication service.

Control of mobility and migration of object groups

The above discussion presented basic mechanisms and alternatives to manage mobile objects. In addition, we basically support some defaults to control mobility and also movement of logically or physically related groups of objects.

Objects can be marked as *highly mobile* or as *relatively fixed*. This information is given at the class level and can be overwritten at the instance level. If a reference to a highly mobile object is exported via an invocation, the object is moved together with the invocation immediately. This way, further access to an object at the destination site will be local and therefore more efficient. Relatively fixed objects are only moved on demand as their migration costs are assumed to be significant.

In addition, object references can be marked as being *strong*; if an object moves, all objects referenced this way move together with it as they are assumed to form a coherent unit [Bla87a]. A stronger version of such units is supported by issuing a migration request to a logical group of objects (given by a collection of object references). In both cases, the required amount of physical communication can be significantly reduced by internally concatenating all coresiding objects which are moved.

In order to enable larger grains of mobility, we have developed basic concepts for physical object groups similar to [Hab90a]. In particular, a mobile object is not necessarily only a collection of flat data but can also contain complex nested structures internally. However, no direct references from other objects into such a cluster are allowed. This way, references within the cluster can be implemented as relative pointers and no table indirections are necessary (see Figure 7). Access to internal objects can then be implemented more efficiently than access to enclosing mobile objects. When internal access is performed, the base address of the enclosing object is added to each relative pointer. When a cluster is moving, internal pointers do not have to be adjusted as they are relative. We do not intend to support the transformation of cluster-internal objects to mobile objects as outlined in [Hab90a].

Finally, we suspend migration requests for an object if it is currently accessed and cancel them after timeout. That is, we do not consider the migration of activated objects as for example in [Jul88a] due to its severe implementation problems. Moreover, objects can be explicitly fixed at their current location; in this case, a migration request is simply rejected. The migration procedure itself consists of transforming an object or a group of objects into a flat form, shipping it to the destination site, reinstalling it, performing the described proxy management, and acknowledging the migration at the source site. We do not go into further details of this procedure as related implementation problems have been described by other authors in depth [Jul88a].

† For simplicity of presentation, we assume the existence of an underlying RPC service. In reality, we optimize the described mechanism by employing a TCP/IP message passing service.

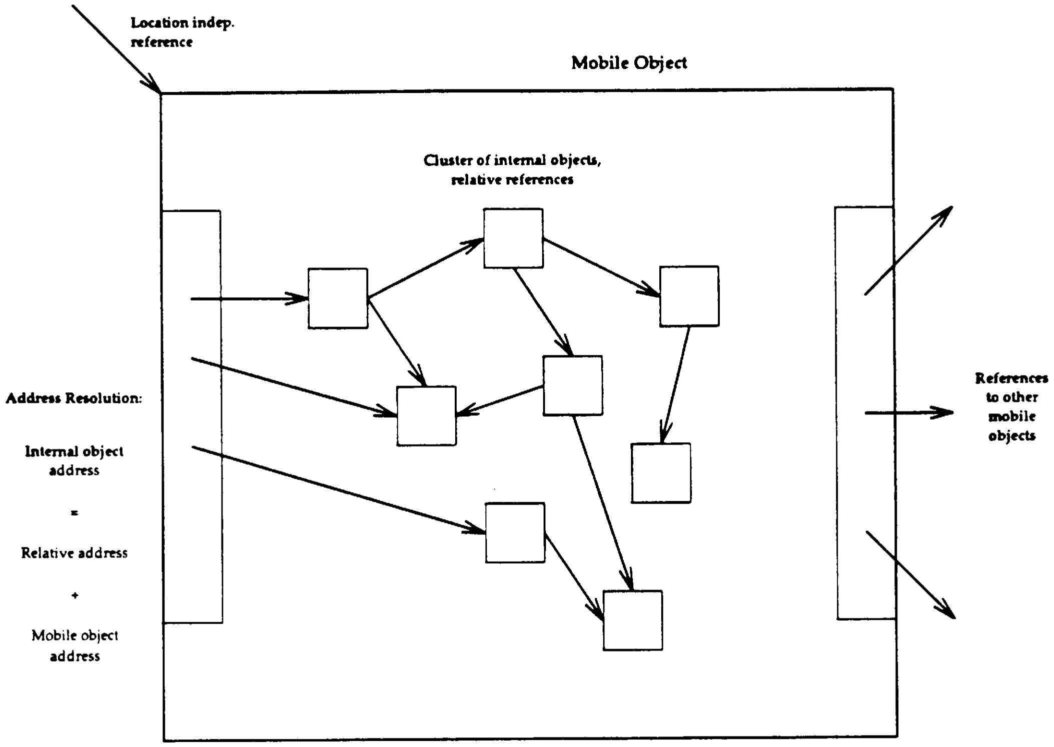

Location indep. reference
Mobile Object
Cluster of internal objects, relative references
Address Resolution:
Internal object address
=
Relative address
+
Mobile object address
References to other mobile objects

Figure 7: *Structure of a cluster within a mobile object*

3.3. Implementation

An implementation of our approach is being performed at our university. It is based on former experiences gathered with a similar environment named *Object Manager* during a post-doc stay at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. This system was based on an existing OODB prototype named *DEPOT* which provides a client/server object repository model. One or several OS/2 clients can access a DEPOT database on an MVS or VM/CMS host or OS/2 system in order to retrieve and store back objects using an extended C interface. The database is built on top of relational databases (DB2 and SQL/DS).

Our extended effort is now focusing on a C++-based UNIX implementation of distributed object management. This support system has been implemented as a C++ class library under UNIX (Ultrix) on DECStations 5000 and 3100. For low-level network communication, it uses TCP/IP sockets. Concurrent object invocations within an operating system process are implemented by a modified version of the AT&T C++ threads package.

All distribution-related object management operations are defined within a specific class *DObject* and are inherited by application-specific classes. In addition to remote invocation, basic operations for migration of non-activated objects between nodes and for related locating, fixing and unfixing purposes are supported:

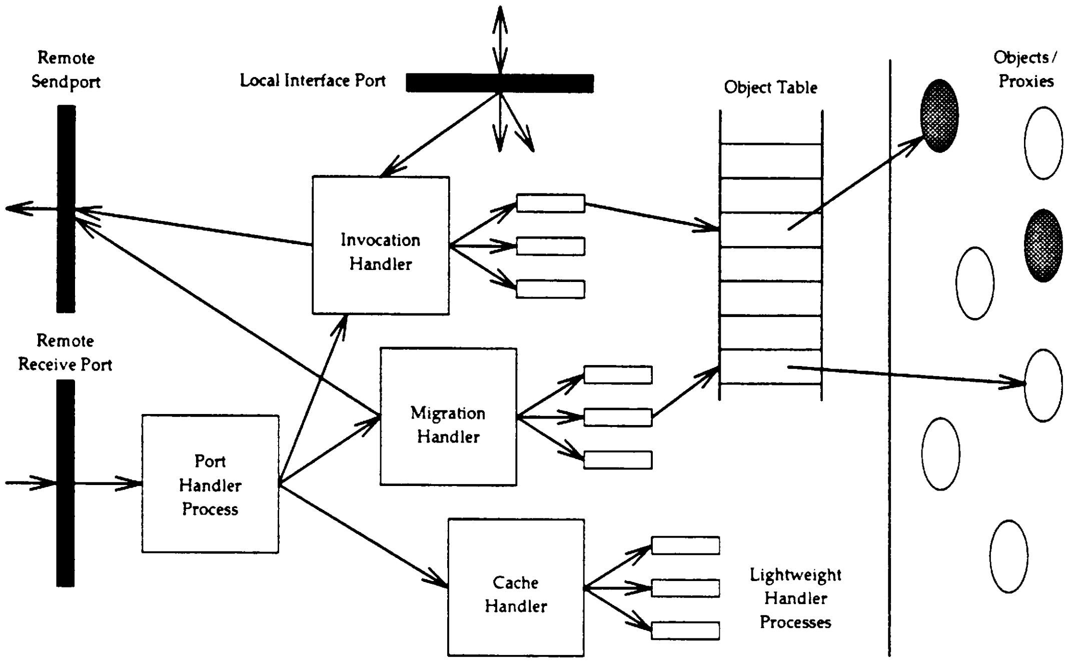


Figure 8: *Implementation structure per node*

```
class DObject {  // ... instance variables
 public:  DObject (LogicalNode*);          // constructor to create at node
          ~DObject ();                     // destructor
          LogicalNode *locate ();          // to locate the object
          boolean move (LogicalNode*);     // to move the object
          void fix ();                     // to fix the object (avoid move)
          void unfix (); };                // to release the object
```

Recent performance measurements show numbers in the range of 100 milliseconds to retrieve an object from a server database to a client workstation.

Figure 8 shows an outline of our implementation structure per node. It is based on a port-oriented message communication mechanism and on lightweight processes sharing an address space. A port handler process is passively waiting at a remote receive port and is processing all incoming messages. Depending on their contents, the handler process passes them to an invocation, migration, or cache handler module. These modules can also be directly invoked via a local interface. If invoked via the receive port handler, the modules spawn lightweight handler processes to perform local requests. The port handler is then able to listen to the receive port again. In order to process local requests, the object table is accessed to retrieve objects or proxies in persistent memory. Remote requests are passed to the appropriate node via the remote sendport. They can result from local applications, e.g. if they perform remote object invocations, or from previous incoming remote requests, e.g. if an object is searched along a forwarding chain.

Several possible optimizations to this basic structure are possible. The lightweight processes spawned by the handler modules can be preallocated. That is, a limited number of processes is permanently existing and is only increased on demand. Local invocations can bypass our special modules and can operate directly on the object table. They would then only fall back to handler modules if they make remote invocations necessary. Other optimizations concern the message passing between nodes. Most important, messages along forwarding chains are sent asynchronously; a response is only sent from the final destination to the original source. Communication during migration is synchronous, i.e. after a moved object has been installed at its destination, a response message is sent back to the handler at its source. Response messages are associated with request messages based on unique message identifiers.

The selected implementation structure enables flexible interfacing with the local system components. The handler modules make distribution almost transparent. However, an application can still control object locations explicitly.

4. Application Service: Distributed Office Procedures

As an example application of our system, we designed a service to specify and manage distrib:ıted office procedures [Sch91a]. It has some basic ideas in common with other systems like [Art90a]. The service has partially been implemented within a joint IBM project.

In particular, our approach provides a declarative notation to specify a conditional sequence of processing steps which are required to perform a structured office task. A concrete example is the processing of a travel expense form (see Figure 9). In this example, an employee fills out a request form at his personal workstation and attaches an itinerary description and some other required documents. All these data are represented as mobile objects. The employee then creates an office procedure object of the appropriate predefined type. The header section of the object contains type information, its owner, its create time, and some other data. It also has a routing specification which specifies the processing steps (services) which are to be applied to its data. The employee can instantiate the specified services with concrete servers/office workers or can rely on a dynamic, system-controlled ser-

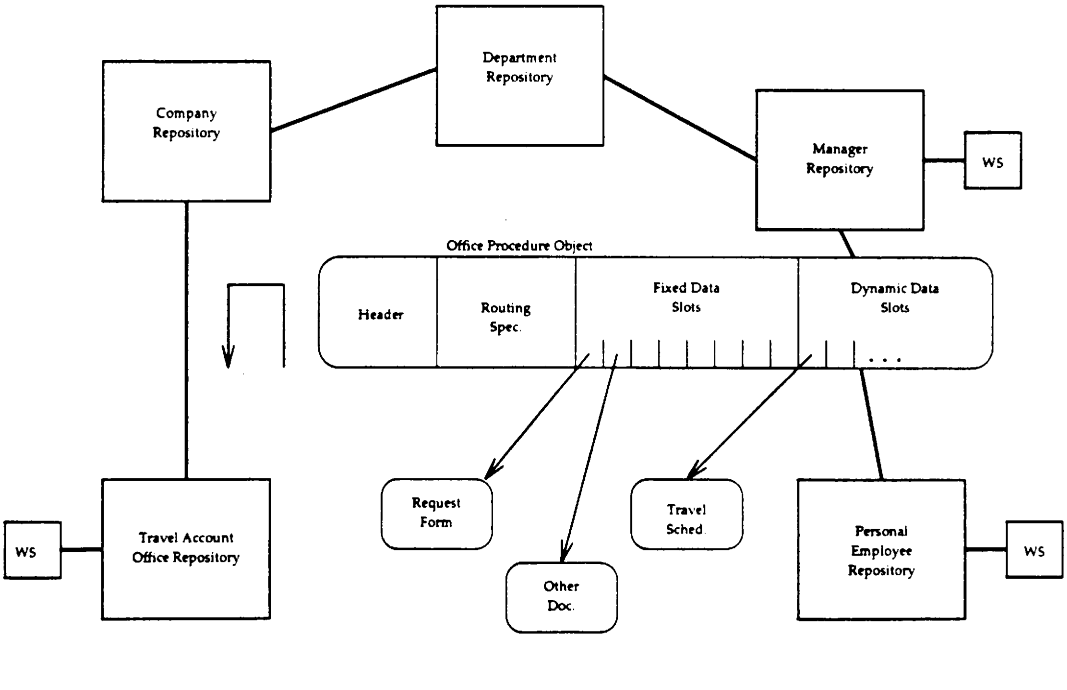


Figure 9: *Example of a distributed office procedure*

vice binding. References to the attached data objects are inserted in a number of predefined fixed data slots. Additional slots can be defined in order to hold references to dynamically created data.

The office procedure object is then forwarded to the respective servers. It is therefore physically moved to their repositories. The attached data are either moved with the object or are accessed remotely. First, the object is moved to the manager's repository; the manager accesses it there from his workstation in order to approve the form. Then it is migrated to the travel account office repository and then the account office calculates the amount to be refunded to the employee. Additional repositories may be involved: the manager may request data objects from the department repository in order to analyze the previous travel record of the employee. Likewise, the travel account office may access the company repository to get additional address and employment information about the employee. In both cases, copies of the required data objects are moved to the travel account office repository or to the manager repository, respectively.

The declarative office procedure specification language of our approach consists of a notation to describe the server environment, i.e. to specify server types and instances, a notation to specify, instantiate and modify the data slots of an office procedure, and a notation to describe the routing of the office procedure object as a graph with service nodes, interconnection edges, and attached routing conditions. In addition, several other features are supported, for example, timeout control of service execution or different means to specify service binding. In particular, the binding can be completely static or completely dynamic or it can be basically dynamic but guided by static binding hints and by server evaluation functions. The associated runtime support consists of an initiation phase (office procedure setup, attachment of data objects, and specification of binding hints), a major execution phase (dynamic office procedure routing and service execution), and a termination phase (notification of success or failure, distribution and persistent storage of result data). In addition, supervisor commands can be issued via a remote command interface, for example to query the current state and location of an office procedure, to suspend and resume its execution, to inspect and modify its attached data objects, and to setup a facility to monitor its execution continuously.

While server interfaces can be implemented by objects which are usually fixed at their location, office procedures and attached data are mapped to highly mobile objects which move between the different service sites. In addition, we perform location independent remote object access to servers, via the remote command interface, and for data objects which are not moved together with the office procedure.

5. Conclusion

The paper described the design and implementation of a mobile object environment. We presented several different mechanisms to support the required facilities, especially an extended forward addressing policy with several specific technical aspects. We also validated the applicability of our approach by designing an application-level office procedure service on top of it.

Future work will focus on the development of concepts to integrate the approach with different underlying communication facilities. Namely, we are porting the UNIX implementation onto the OSF Distributed

Computing Environment using Threads, RPC, and the Cell Directory Service. In addition, we would like to achieve better insight into the usability and performance of our extended forward addressing mechanisms. Moreover, the cooperative office application field will be investigated further.

References

[Art90a] Y. Artsy, "Routing Objects on Action Paths," *10th IEEE Int. Conf. on Distributed Computing Systems, Paris*, pp. 572-579 (1990).

[Ben87a] J. K. Bennett, "The Design and Implementation of Distributed Smalltalk," *ACM OOPSLA Conf., Orlando*, pp. 318-330 (1987).

[Ber90a] E. Bertino, M. Negri, G. Pelagatti, and L. Sbatella, "An Object-Oriented Data Model for Distributed Office Applications," *ACM Int. Conf. on Office Information Systems, Cambridge, Mass*, pp. 216-226 (April 1990).

[Bla87a] A. Black, N. Hitchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering* **13**(1), pp. 65-75 (Jan. 1987).

[Bla89a] A. Black and Y. Artsy, "Implementing Location Independent Invocation," *9th Int. Conf. on Distributed Computing Systems, Newport Beach*, pp. 550-559 (1989).

[Cha89a] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield, *The Amber System: Parallel Programming on a Network of Multiprocessors*, Internal Report, Univ. of Washington, Seattle, 1989.

[Chi91a] R. S. Chin and S. T. Chanson, "Distributed Object-Based Programming Systems," *ACM Computing Surveys* **23**(1), pp. 91-124 (March 1991).

[Cor90a] A. Corradi, L. Leonardi, and M. Zannini, "Distributed Environments Based on Objects: Upgrading Smalltalk Towards Distribution," *IEEE Int. Phoenix Conf. on Computers and Communication, Phoenix, Arizona*, pp. 332-339 (March 1990).

[Hab90a] A. El Habbash, J. Grimson, and C. Horn, "Towards an Efficient Management of Objects in a Distributed Environment," *2nd Int. IEEE Symp. on Databses in Parallel and Distributed Systems, Trinity College, Dublin, Ireland*, pp. 181-190 (July 1990).

[Jul88a] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems* **6**(1), pp. 109-133 (Feb. 1988).

[Kra90a] S. Krakowiak, M. Meysembourg, and H. Van Nguyen et al., "Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications," *Journal of Object-Oriented Programming* **3**(3), pp. 11-22 (Sept/Oct 1990).

[Lis90a] B. Liskov, R. Gruber, P. Johnson, and L. Shrira, "A Highly Available Object Repository for Use in a Heterogeneous Distributed System," *4th Int. Workshop on Persistent*

Object Systems, Martha's Vineyard, Mass., pp. 247-258 (Sept. 1990).

[Sch91a] A. Schill, "Distributed System and Execution Model for Office Environments," *Computer Communications* **14**(8), pp. 478-488 (Oct. 1991).

Comprehensive Automated Storage Mangement for Networked Environments

Gottfried B. Bertram
Michael Sieber

Network and System Management Division
Hewlett Packard GmbH
Germany
michas@hpbbn.bbn.hp.com

Abstract

Knowing that the "Network is the Computer" implies distribution of data on many storage devices throughout a network, the system. People responsible for the information processing architecture, i.e., the distributed data center, must be capable for managing these stored information elements and storage locations. This requires complex designs and operations. The advantage of open distributed systems must not be lost because of problems associated with managing them.

To resolve these problems, highly automated solutions for unattended operations, ease of use and setup, hiding of unnecessary complexity, etc., is currently under development by HP's System Management Division (SMD). The Automated Storage Management (ASM) project will address the full complexity of networked environments we encounter today. A comprehensive Approach.

Introduction

The networked environments we see today, and those we expect and envision to serve with an automated storage management solution consist of several servers linked together with a variety of desktop clients. These servers, typically HP 9000 (UNIX), or HP 3000 (MPE) mini computers serve Terminals, PC's, and Workstations. The network being the system, to manage it is characterized by a possible co-existence of three important network operating systems (NOS):

- UNIX networks on TCP/IP offering OSF's Distributed Computing Environment (DCE) services
- Novel's Netware
- Lan Manager

Since these are portable versions of Netware and Lan Manager, we expect all three operating environments to coexist in the same network.

Such an environment implies a large number and variety of storage locations and organizations to serve, hence increasing the environment complexity.

The source of information objects are from:

- Personal Computer Clients (DOS)
- Workstation clients (UX)
- Personal Computer Servers (Netware, LM)
- Mini Computer Servers (UX and MPE)

These information objects, for storage management purposes, may be organised as follows:

- Disks, volumes, raw device contents
- Files, directories of files
- Data bases

To implement Storage Management, the following operations are required:

- Backup and recovery/restore
- Archival and retrieval
- Storage space management

From the above sources, the kinds of information objects and the kinds of operations, there are three dozen combinations that a comprehensive storage management solution must address. Although, not all of these combinations make sense, e.g., to archive disks for historical or legal purposes is certainly not meaningful, and data bases will almost always be backed up from servers and not from desktop clients, a considerably long list of storage management solutions need to be developed for distributed environments:

Disk Backup/Recovery for:

- DOS PC's, MAC's and Workstations
- UN*X Servers with portable Netware and portable LAN Manager (LM/X)
- MPE Servers with portable Netware and portable LAN Manager (LM/iX)
- Native Netware (INTEL-PC) Servers
- LAN Manager Server on OS/2 PC's

Data Base Backup/Recovery for:

- UN*X Servers
- MPE Servers
- INTEL PC Servers

If data bases are to be backed up from servers, the different Data Base Management System (DBMS) suppliers must be taken into account as they require different support for their special flavour of backup processes.

File Backup/Restore from and to:

- DOS PC's, MAC's and Workstations
- UN*X File Servers
- Novell Netware File Servers

- LAN Manager File Servers
- MPE Servers

Assuming that all three major network operating environments coexist on the same network, file backup and restore must be able to handle the different file name conventions of DOS, MAC, Netware, OS/2, UN*X, NFS, DCE/DFS and some proprietary environments.

Archival and Retrieval

For some environments, both archival and retrieval may be required. Archival differs from backup as it's purpose is not to safeguard a copy, but to retain historical copies of files for legal and other purposes, e.g., version maintenance.

Data base archival is a required storage management task, and can be offered as an integrated solution consisting file archival and procedures offered by the major DBMS suppliers, e.g., ORACLE, INFORMIX, INGRES, SYBASE, DB2, ALLBASE, ADABAS, etc. (Most DBMS have utilities to unload/export data base tables to the file system of their host server and to reload/import them). Archiving is then achieved by moving these files into archive together with appropriate attributes to retrieve them later.

Storage Management

The purposes of storage space management is to monitor usage levels of disks and to free up space, by rolling out stored information objects that are infrequently accessed or are already old and obsolete. Storage space management maybe considered for all disks in the system:

- DOS PC's, MAC's workstations (i.e., all clients)
- UN*X, MPE Servers
- PC Servers (Netware, OS/2)

Some scientific and technical applications use huge amounts of data, organised in file systems too big to be always locally present. Storage space management may serve these environments offering high performance, automatic, roll out/roll in processes to integrate data from fast local disks to networked storage servers, and to removable high density media in autochangers, and to migrate them back transparently if accessed.

The Storage Management Process Model

Managing systems/networks must focus on people who perform the management role, the methods and tools they use to perform that role, and the processes that describe how people perform their management role using the methods and tools at hand. This applies to system management in general and to storage management in particular.

Technology changes like distributed or client-server computing not only introduce more services, possibilities, capabilities and advantages over traditional centralized mainframes, but also introduce more complexity. Processes to manage the<e distributed computing environments must reduce this complexity through automation.

In the case of storage management, the key trend identified is to shift the focus from different stored information objects and their management tasks, e.g., backup, recover, archive and retrieve, to the people who manage them. Only the automation of tasks and entire processes people perform, will cope with the challenge to keep pace with the changes in information processing environments.

To manage the backup, recovery, archival and retrieval of such different objects as disks files and databases, together with control over storage space available across an entire network, cannot be done any more by the selective and individual usage of single tools and methods. It needs a bundle of processes to assure safeguarding and archiving of valuable information. Ideally, we think of finding a way to free up the people from carrying out the processes to let them only manage the process of storage management.

The solution is not to build tools people can use to manage stored information objects efficiently, but to actually remove the people from the process as much as possible. Human reaction and interaction speed is limited, as are human abilities to cope with situations of even increasing complexity like data bases, file systems and storage locations scattered throughout the network. What needs to be automated are processes that are large complex collections of tasks performed by people today.

Backup a data base from one server in the network to a special backup and storage server that offers all suitable peripheral devices and media, requires a long sequence of task to be carried out, to be understood and mastered. It requires the knowledge of specific DBMS's on line backup procedures, the way and format of the data to be backed up, how they need to be transported to the backup up server, and where they will be placed on what media.

Recovery is even more complicated. Such management processes need improvements in the people, in the tools and methods and the processes. Most solutions today only enable storage management for some of the different purposes, and fall short of what customers need as they do not deal with the processes that people actually have to perform.

For the definition of advanced storage management solutions, we therefore need a solution maturity framework, and a target level of maturity we want to achieve for the next generation of storage management solutions. Todays solutions are characterized by the fact that all knowledge on the processes has to be in the heads of the people managing the system. These solutions only offer tools to automate single steps and tasks. To explain, we may call this an "initial" level of maturity, followed by a level called "repeatable", and another level described as "definable"

Initial

A variety of single step solutions address partial needs only. An absence of consistent, comprehensive and integrated set of methods and tools impede the creation of reliable policies, procedures and processes, to safeguard and archive valuable information.

Methods and tools for storage management are known and tools for all specific purposes may be available, but none of them are integrated or based on a consistent architecture or framework. Operations are carried out on an individual, ad hoc basis. There is no common or centrally available knowledge in case there is loss of information and requires recovery.

Repeatable

Solutions offer a set of fairly comprehensive tools and methods. These tools can be put together and configured to carry out a variety or sequence of tasks.

But the configuration and operation of such sequences still requires an expert to design and set up the processes desired. Very often these tasks are performed by a specialized person or a guru who knows about the "what", "when" and "how" of the processes. He is the person everybody knows they need to call just in case a complicated recovery has to be carried out. Processes depending on persons bear the risk of loosing the experience and expertise with changes of either environment/technology or with personnel/organizational changes. A shift from centralizing computing environments already implies such a shift in paradigm, resulting in demand for new processes to manage the backup, archival and storage space available in networked systems. Of course, this change does introduce greater complexity together with tremendous advantages of having computing power everywhere on a network. But these complexities aggravate the problems of management of such systems. SMD's ASM project is therefore targeted to achieve the next level of management solution maturity.

Definable

As size and complexity of distributed data processing environments grow continuously, so does an increase in the risks of managing them by repeatable processes only. Too rapid a change, complexity and variety have to be learned and mastered by too many experts. One can always increase management resources and improve processes, but beyond a certain level of complexity, help can only be expected by another level of automation. The "definable" level of solution maturity will allow policies to be carried out automatically and independently from personally owned knowledge.

If solutions are policy controlled, the policies will be available, documented and independent of specific experts. They may be inspected, changed and adapted. The defined solution is people independent, consist of documented policies that record process characteristics, steps and tasks that can be passed over to people, and either are executed manually or by pre-programmed reactions, operations and administrative measures. Hence, it achieves a higher degree of automation and decreases the complexity seen by people managing the system. The definable policy level of management solutions enhance management by managing the processes.

For design of storage management solutions we need a general management process architecture that enables definable level of solutions. We see a four stage architecture for storage management purposes and processes: operation, execution, administration, change of policy and planning.

An Architecture of Storage Management

An architecture for the definable solutions of self-managed systems consists of three process loops on three operational levels. The lowest level for execution of storage management operations offer storage services together with transportation and transformation of stored information objects from and to storage management clients. These services operate on the distributed system and record results from the operations. The results gathered are measurements fed into the next higher level, i.e., operational and then administrational levels. For the operational level, the operations executed and events recorded from a loop to control the correct execution of the operations. The record loop is associated with the administrational level to keep track of the state changes and the history of operations. The third loop is between the administrational and operational levels. Application strategies get policies and schedules set by administrators and put them into the administrational services. From these administrational services, the application strategies are feedback records, to be offered as reports. The administrational services execute the policies regarding what to do when, through schedules applied to the application operations.

Both the application strategies and application operations, come in different flavours, e.g. database backup and archival or file backup and archival, and their inverse operations, restore and retrieval. The application operations may monitor and report events and certain states to local operators, if there is no way to continue operations without human intervention. But usually, the application operations execute automatically, and only report results to the administrational services and application strategies, thus executing all processes automatically, controlled by policies and schedules that are set up as application strategies by administration. This is the criterion that applies to the definable level. One might argue that in the case of restore/recover or retrieval operations, there will always be a need for an operator, but such need depends on the nature of the event that requires a restore/recover/retrieval. In many cases these operations may be carried out and triggered automatically.

Besides the strategic and operational components of the storage management architecture, services for storage, transportation, transformation and administration have been mentioned. All these services are common to the case set of storage management functions.

Backup, archival and administration of available storage space use copying or moving mechanisms to get stored information objects from storage clients to storage servers and to place them on the available hierarchy of storage devices. Regardless of application, such transportation can be achieved on LAN or local BUS data channels, by generally available methods for file transfer and networked file management (e.g., NFS, DFS, Netware Lan Manager, FIP, FTAM). The administrational components will have to record the names, locations (sources and destinations) owners, etc., and provide these information attributes to storage management applications. To execute copying and moving, a good architecture should provide separate command and data channels.

On the command channel, management applications would talk to local agent processes for the transport execution. A remote management application may thus activate local managers, that in turn, use local agents. The communication of remote activation of operations, and the feedback of events signalling correct or unsuccessful operations, would be carried out using standard management protocols.

Transportation is not the only class of function required for central operational core. Usually all information objects that will be entered into storage management undergo some transformation or translation. Desk files and database tables will be transformed into a standard object, consisting of a header and a bit stream to be stored on disk or other devices. One may choose some standard formats for that, and conversions between them, e.g., tar, cpio, etc. Very often, encryptions, compressions or import/export transformation for databases may be asked for. Since there will be different file systems that coexisting on the netware, it will be required to translate between the different file name specifications of UN*X, DFS, Novell, Mac, DOS, HPFS, etc. The administration service mentioned before, is a database driven application, using a relational data base service either centrally or distributed. This database will serve as the repository for:

- Clients and users, their contracts, rights and obligations.
- Contracts on what to do when, i.e. policies, worklists and schedules.
- Histories of executed operations (e.g. backup sessions) to provide information on what was executed and the results achieved.
- Information objects, their identifications, names, sizes, time and date stamps, and more attributes as required, e.g., archival.

The transformation and transportation services, together with the administrational component, require one more server to store the managed information, objects on devices and media. The storage server and or services has to handle all devices and media. A storage medium, either a fixed or a removable disk, that allows a file system to maintain its format, e.g., of UN*X, will be adequate.

The storage service has the knowledge about different media, their locations, and the placement of objects on the media. Is the service or the server used to access the backed-up or archived objects. The storage services maintain the information regarding differences between diverse media and their access mechanisms, transparent for other services for transportation. A storage server need not be in central location. The set of all computers in the network that offer devices suitable for backup and archival on different media is the storage server.

Integration

When proper integration is achieved, the end is more than the sum of its components. This is due to synergy.

For a comprehensive, cohesive, and consistent set of storage management solution, we want to achieve a means of using the same operation for the same purposes.

File backup and archiving only differ by additional file attributes required to store an object or retrieve it. Archiving usually adds an additional time mark to differentiate between different versions.

Data base archival and file archival differ additionally in that, data base tables are unloaded or reloaded prior to archiving or after retrieval to or from a file system.

System backups and recoveries using raw device images are done pretty much the same way many data bases are handled. Recovery of a database is not the same as system recovery. After restoring the disk image, data base recovery also needs recovery of transactions that are recorded in log files being archived. Such applications always differ from each other because of their different purpose. But many building blocks are of the same nature for common mechanisms, and so is the handling of stored information objects to transport, transform, store and recall them from devices and media. This serves in an integrated way for all solutions. But integration is more, as it focuses on how people, processes, applications and the pieces of the system work together.

Within an organization people perform the tasks of system management and storage management. This must be planned and integrated with the processes storage management applications offer. These applications are reached using a UI (user interface) that offers a common and consistent way of interconnecting the people executing management processes using programmed solutions. The purpose of UI integration is to provide a close, spatial location of the information and tool available to the end user under a common presentation and behaviour. Integration is one of four dimensions:

- UI
- Data
- Process
- Network

Data integration comes as a common data base for backup, archival, storage space control and media management.

Process integration is the ability to automatically coordinate tasks and group them into processes the user would otherwise have to do himself. For example, most DBMS's offer functions for the task to unload tables from a data base for archiving, but the archiving itself has to be carried out by some system vendor supplied function. Process integration must be assured to have mechanisms at hand to implement entire process sequences. This can be achieved through means of simple event driven actions to sophisticated object orientated exchange and message mechanisms.

Network, or the system itself is the last dimension of integration, that allows control over distributed storage management solutions. The ultimate integration point is single point management, e.g., having the means to access all solution parts from any one desktop system on the network. This single point console could offer the integrated UI (preferably graphical), with access to tasks, processes, data and all capabilities to monitor what happens during the execution of management processes.

Software System Requirements for Multicomputing

Brian N. Bershad

School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
brian.bershad@cs.cmu.edu

Abstract

A multicomputer is a collection of a large number of distributed, independent processors connected by a high-speed network. Software systems for high-performance, scalable multicomputers have been evolving far less rapidly than hardware systems. As a result, multicomputers continue to be viewed as an anomaly and not as a cornerstone of mainstream computing environments.

In this paper I discuss several requirements for multicomputer software systems. These requirements include a parallel programming environment which allows programmers to balance programmability and efficiency, application-level networking protocols, and replicable distributed operating system services running on a portable microkernel.

1. Introduction

A multicomputer is a collection of a large number of distributed, independent processors connected by a high-speed network. In contrast, a multiprocessor generally consists of a small number of processors sharing a common physical memory. Multicomputers offer the promise of greater performance, scalability and cost effectiveness than uniprocessors and shared-memory multiprocessors.

Software systems for high-performance, scalable multicomputers have been evolving far less rapidly than their counterpart hardware systems. As a result, the improvements in processor speed, and networking bandwidth and latency have not yet been matched by improvements in system usability and scalability. In other words, while it's possible to build, or buy a teraop multicomputer, effective utilization of that

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035, by a National Science Foundation (NSF) Presidential Young Investigator award, by the Open Software Foundation (OSF), by Digital Equipment Corporation's External Research Program, and by the Xe.ox Corporation.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF, the U.S. government, OSF, DEC, or Xerox.

machine remains a significant problem. Although there have been some successes in constructing specialized parallel applications, and running coarse-grained multiprogrammed loads, multicomputers continue to be viewed as an anomaly, and not as a cornerstone of mainstream computing environments.

A multicomputer cannot be treated entirely as a parallel processor because processors do not share a common memory, and may not even share a common architecture. On the other hand, it should not be treated entirely as a distributed system because processors need not be considered wholly autonomous, arbitrarily dissimilar, and randomly dispersed. Moreover, many multicomputer networks, such as the Paragon 2D-mesh [Rat92a], the CM-5 fat-tree [Ric92a], and even ATM-based LANs [Rid89a], can be assumed to be nearly 100% reliable, both in message delivery and ordering. These characteristics of multicomputers – neither an entirely parallel system *nor* an entirely distributed one, introduce new constraints and requirements for the software that will be used to manage them.

Effective multicomputing imposes three requirements on software systems. These are:

- A parallel programming environment that allows the programmer to balance programmability and efficiency.
- Low-overhead software paths between the applications and the network.
- A set of operating system services distributed across many nodes in the multicomputer layered on top of a portable microkernel.

In the rest of this paper, I discuss each of these requirements for multicomputer software systems. I use the term "software systems" rather than "operating systems" to imply a scope broader than that implied by just the latter. Effective multicomputing is not simply an operating systems problem, restricted to virtual memory management and communication protocols, but is instead an entire systems problem, ranging from the operating system kernel layer (local resource management) to the network (communications management) to the parallel-distributed programming environment (applications management) available on the system.

2. The Parallel Programming Environment

Parallel processing is the *raison d'être* of multicomputers. Consequently, its requirements should serve as the driving force of all multicomputer system software. In other words, a highly parallelizable, network transparent, fault tolerant `ioctl` interface is not going to make the task of parallel programming on multicomputer any easier.

The absence of a coherent, logically centralized shared memory is what makes a multicomputer both attractive and hard to deal with. The lack of a global shared memory enables a distributed memory multicomputer to scale with little cost, making it an attractive computing base. In contrast, the few large-scale truly shared memory multiprocessors such as Tera and the KSR machine [Bel92a] have a substantially higher "per-node" cost and have not yet demonstrated themselves to cost-effective. On the other hand, it is the largely absence of a single shared memory that makes a multicomputer difficult to program.

2.1. Memory Consistency Models

In recent years, the architecture and operating systems communities have been experimenting with an array of memory consistency models which provide memory semantics that are less stringent than those intuitively expected from a memory system. These models are intended to reduce the overhead of providing a consistent distributed shared memory. Intuitively, programmers expect a memory system to be *sequentially consistent* [Lam79a]. In such a system, all processors perceive all memory accesses in exactly the same order. Trivially, a uniprocessor is sequentially consistent. A small-scale shared memory multiprocessor with strict memory buffering (reads do not bypass writes) is also sequentially consistent. Although it matches the programmer's base expectation of memory system behavior, sequentially consistent systems incur far greater communications overhead than is generally needed by a shared memory parallel program.

Memory access performed by a parallel program can be divided into two classes: regular accesses and synchronizing accesses. Regular accesses are those that are performed on a program's data structures. Synchronizing accesses are those that are used to schedule regular accesses, and are performed on locks and semaphores. In a distributed shared memory system, synchronizing accesses can be used to control the behavior of the memory system, as well as processors, in order to reduce the overhead of ensuring a consistent shared memory. Simply put: a synchronizing access by a processor is a signal to the memory system that previous updates performed by that processor should become visible to other processors in the network. Until the synchronizing access occurs, other processors should be unable to access the data for which the synchronization is occurring (presumably that being written) anyway, so any transmission of new values is unnecessary.

There are a large class of memory consistency models which exploit synchronizing accesses. These include: weak consistency [Dub86a], release consistency [Gha90a], and entry consistency [Ber91a]. Each model requires progressively more information from the program with respect to synchronizing accesses, and each, in order, provides a narrower range of guarantees about the consistency of memory with respect to such accesses. In essence, as the consistency model is weakened, the programmer must provide more information to the memory system describing the type and scope of a synchronizing access. In return, the memory system can reduce the amount of communication it performs to provide the level of consistency guaranteed by the model.

A parallel programming environment for a multicomputer should have the following attributes with respect to the memory consistency model:

- *Support for a distributed shared memory that can support a range of memory consistency models.* The "best" consistency model for a parallel program depends on the network, processor speed, and a program's sharing patterns. It makes no sense to dictate a highly-efficient but semantically restrictive model which makes writing a parallel program difficult when the program itself would function just as efficiently under a less strict model. For example, a two-processor producer/consumer program is naturally pipelined, and would not be any better served by a memory system which delays operations until synchronization time – a simple asynchronous pipeline is sufficient. On the other hand, it also makes no sense to deny the programmer influence over the communication patterns through the use of a sequentially consistent memory model simply to maintain an aura of transparency.

- *Support for converting "dusty-deck" parallel programs written in C or Fortran that are based on a strongly consistent memory model into programs that can use a weaker consistency model.* A programming environment must provide a graceful migration path by which dusty-deck programs written for a strongly consistent shared memory multiprocessor can be ported to a weakly consistent distributed shared memory. Without this ability, it can be a substantial task to convert a shared memory parallel program written to use one consistency model into one that uses another.
- *Support for isolating and monitoring memory "hot spots" during the execution of a parallel program.* By identifying which memory accesses result in the greatest degree of consistency overhead, the programmer can restructure the code which accesses those hot spots to rely on a less consistent (and less communications-intensive) memory model.

These attributes allow a parallel programmer to begin with a sequentially consistent parallel program and algorithm and to then selectively modify the program to use a weakly consistent memory model where such use would improve performance.

2.2. A Consistent Operating System Interface

From the perspective of the program developer, a parallel program consists of two disjoint sets of code: that which the programmer writes to compute a solution to some problem, and that which has been written by somebody else to connect the program to the outside world. Although not entirely accurate, programmers generally consider this latter set of code to be "the operating system."

Whether or not the code is truly the operating system, or is simply a set of standard libraries linked into the program, code not written by the parallel programmer must present reasonable semantics in the context of a multicomputer application. Some examples of unreasonable semantics are:

- A standard I/O library which does not properly interleave I/O to the same descriptor simply because the I/O was performed from different processors.
- A dynamic memory allocator which does maintain consistency of address space utilization across processors.
- A file system interface which does not permit a file descriptor returned by a single "open" call to be used on any processor except the one on which the open occurred.

These unreasonable semantics can arise when the parallel processing runtime system fails to provide completely a single-system image. In essence, a parallel program expects consistency semantics along two axes: memory, which is how the program communicates with itself, and the system interface, which is how the program communicates with the outside world. Failure to provide reasonable consistency semantics along either axis can result in a software system which is difficult to program.

To address this difficulty, the operating system, the runtime, or both must cooperate with an application to present the image of a single underlying operating system.

3. Low-Overhead Paths to the Network

A multicomputer is communications-intensive. A parallel program running on multiple nodes communicates with itself, and even a sequential program communicates heavily with services distributed throughout the multicomputer and with other services distributed throughout the network. High communication throughput and low latency are therefore critical to the effectiveness of the multicomputer.

3.1. Communication Protocols as Application-Level Libraries

While protocols such as UDP and TCP should be considered an operating system service, they are a service in much the same vein as the UNIX standard I/O (stdio) service. They export one I/O interface to clients and import a second I/O interface from the operating system. I/O is buffered and formatted in the client address space to improve performance by reducing the frequency of operating system calls. When buffering results in a page-sized I/O transfer, it can also eliminate a copy operation. Similar benefits can be achieved by implementing communication protocols as application-level libraries.

A communication protocol implemented in the the client's address space makes the following possible:

- There is a minimal latency path from one endpoint of the protocol to the network interface. Procedure calls, rather than system calls or cross-address space remote procedure calls, are used to invoke protocol services.
- Needless copy operations, even for small packets, can be eliminated. By-reference parameters can truly be by-reference.
- Protocol implementations can be tuned for performance on a per-application basis. Options such as buffer size, window size, time-out length, etc. can all be manipulated by the application since the protocol's algorithms and data structures are all directly accessible.

This approach is a departure from current practices where the protocol implementations reside in the operating system kernel or in a dedicated process. The key to this new approach is to separate the protocol's implementation from the operating system's, and to provide a complementary interface between them that allows the two to coexist. Further, the interface between the operating system and the protocol must be designed so that it presents a cohesive interface to applications, which can transparently take advantage of the new protocol's structure as though it were implemented in the operating system. For example, the UNIX socket interface, because it is accessible through UNIX file descriptors, affects many other interfaces such as ioctl and select, and these interfaces must behave as though they were tightly coupled with the protocol code.

3.2. A Mapped Communications Interface

With communication protocols running in application-level libraries, the only barrier between an application and the network is the device driver to the network interface. In order to achieve minimal latency for network access, it is necessary to map the network interface directly into applications' address spaces. In this way, applications can communicate with the network as though it were an extended memory system, sending and receiving packets through direct memory accesses to I/O space.

4. The Multicomputer Operating System

An operating system for a multicomputer must satisfy many of the same requirements as a large scale distributed file system [Sat85a]. It must be scalable, reliable, secure, and have predictable performance. The solutions appropriate to large-scale distributed file systems are therefore also appropriate to general purpose multicomputer operating systems:

- The semantics of the operating system interface must not diverge too greatly from a standard which has become acceptable in single node systems. For the short term, at least, this means it must provide a UNIX interface. As other multitasking interfaces begin to take hold in the marketplace, other interfaces may become acceptable.
- There must be no single point of failure for the entire system, although individual applications may fail due to the failure of one server or another. In other words, while it may be the case that one or more programs may fail to complete due to a component failure, it must never be the case that all programs fail to complete.
- Services must be replicable with fine granularity. Replicated services increase scalability and performance, and improve fault-tolerance. For example, an NFS-style directory and file management mechanism, whereby directories and files are one-to-one with servers, is inappropriate for a multicomputer. Many parallel programs are I/O intensive (scatter a multi-megabyte data set across many processors, process, gather a multi-megabyte result data set) and have higher bandwidth requirements than can be delivered by a single file server.
- Trusted services must run on trusted, secure, nodes. In a multicomputer configured out of a network of workstations, for example, it makes little sense to run a global authentication service on a machine in an employee's office. While this requirement seems obvious, arbitrary process migration mechanisms could result in a failure to abide by it.
- Clients should cache results to reduce network and server loads. While this requirement clearly applies to file system reads and writes, it can also be applied to other operating system accesses. For example, client nodes can keep files "open" longer than they really ought to be.
- Clients should use a pipelined interface to mask network and server latencies. While pipelining (write-behind) is relevant to file system writes, it can also be used for other operations which affect system state, but for which the outcome is not immediately required, such as opens and (pre-fetching) reads [Gib92a].
- Services should support an arbitrary call-back mechanism to relay changes in service state. Hauser [Hau92a] suggests the use of client-specific cache-invalidation facilities to allow arbitrary client-side caching. This facility should be generalized in the server interfaces so that clients can specify arbitrary pieces of operating system state, such as number of nodes in the system,

number of files in a directory, etc, which, if changed, result in client notification.

These requirements demand a "multiserver" architecture, in which a flat operating system interface, such as UNIX, is implemented by a collection of servers running on different nodes. Necessary services include process management, processor management, namespace management, filing, authentication, tty management, bidirectional pipes, device management, and a blackboard mechanism to allow these different services to share state in an anonymous fashion. Some of these services, such as namespace management, should be fault-tolerant to survive node failures. Others, such as process management, need only be replicated to ensure availability (that is, if one process manager fails, others can still be accessed, although the state maintained in the failed manager might be lost).

4.1. Kernel Facilities

With network protocols implemented at user-level, and the remaining high-level operating system services distributed throughout the network, one question remains: what sort of kernel should be running on every node? One answer is to use a minimalist, custom-built traphandler which simply vectors requests for system operations off to a controlling processor. This approach is attractive in that the traphandler consumes few processor resources, which is important for memory-impoverished systems. Unfortunately, the approach reflects the position that the nodes in a multicomputer are not first-class processing elements, and therefore do not need many of the kinds of processor-local services such as multiprogramming and virtual memory that are useful in more conventional operating systems.

The custom-built approach also fails to take advantage of an emerging generation of commodity microkernel technology such as Amoeba [Mul90a], Chorus [Roz88a], Mach [Acc86a] and Microsoft's NT operating system. These systems provide a portable microkernel base that exports an abstract machine interface which deals with critical services such as scheduling, protected interprocess communication, and virtual memory management. By using one of these systems for a kernel base, a multicomputer system can easily track new functionality and performance improvements which occur to support other, non-multicomputer systems. Moreover, by standardizing on a microkernel platform, rather than the privileged interface dictated by a specific processor, multicomputer operating systems can be more readily adapted to new processor architectures.

5. Satisfying the Requirements

At CMU, we are working to satisfy these requirements for multicomputer system software. In the Midway project [Ber91a], we are building a distributed shared memory parallel programming environment which supports a range of memory consistency models, and provides a graceful migration path away from sequentially consistent programs. In the context of the Mach project, we are building a suite of IP-based protocols which execute as application-level libraries, yet which also provide UNIX socket semantics. At the device level, we have already demonstrated the effectiveness of mapped network interfaces [For91a]. Wᴇ are now concentrating on implementing a software interface to ATM networks which allows the interface card to be mapped simultaneously into multiple non-privileged applications. Finally, we are continuing our development efforts on the CMU multiserver, which is a collection of low-level operating system services such as those described in Section 4 running on top of the Mach 3.0 microkernel [Jul91a]. These services can be composed to implement a higher-level systems interface such as UNIX.

References

[Acc86a] Michael J. Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young, "Mach: A New Kernel Foundation for Unix Development," *Proceedings of the Summer 1986 USENIX Conference*, pp. 93-113 (July 1986).

[Bel92a] Gordon Bell, "Ultracomputers: A Teraflop Before Its Time," *Communications of the ACM* **35**(8), pp. 27-47 (August 1992).

[Ber91a] Brian N. Bershad and Matt Zekauskas, *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors,* cmucs (September 1991).

[Dub86a] Michel Dubois, Christoph Scheurich, and Faye Briggs, "Memory Access Buffering in Multiprocessors," *Proceedings of the 13th Annual Symposium on Computer Architecture*, pp. 434-442 (June 1986).

[For91a] Alessandro Forin, David B. Golub, and Brian N. Bershad, *An I/O System for Mach 3.0*, Proceedings of the Second Usenix Mach Workshop, November 1991.

[Gha90a] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors," *Proceedings of the 17th Annual Symposium on Computer Architecture*, pp. 15-26 (May 1990).

[Gib92a] Garth A. Gibson, R. Hugo Patterson, and M. Satyanarayanan, *Disk Reads with DRAM Latency*, Proceedings of the Third Workshop on Workstation Operating Systems (WWOS-III), April 1992.

[Hau92a] Carl Hauser, *A Plea for Interfaces that Support Caching*, Proceedings of the Third Workshop on Workstation Operating Systems (WWOS-III), April 1992.

[Jul91a] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy, *Generalized Emulation Services for Mach 3.0: Overview, Experiences and Current Status*, Proceedings of the Second Usenix Mach Workshop, November 1991.

[Lam79a] Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers* **C-28**(9), pp. 241-248 (September 1979).

[Mul90a] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robert van Renesse, and Hans van Staveren, "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer Magazine* **23**(5), pp. 44-54 (May 1990).

[Rat92a] Justin Rattner, "The Paragon System," *Proceedings of the First DARPA Workshop on High Performance Software*, pp. 28.1-28.30 (January 1992).

[Ric92a] John Richardson, "The CM-5 System," *Proceedings of the First DARPA Workshop on High Performance Software*, pp. 28.1-29.20 (January 1992).

[Rid89a] M.J. Rider, "Protocols for ATM Access Networks," *IEEE Network* (January 1989).

[Roz88a] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Giend, M. Guillemont, F. Herrmann, P. Leonard, S. Langlois, and W. Neuhauser, "The Chorus Distributed Operating System," *Computing Systems* **1**(4) (1988).

[Sat85a] M. Satyanaranyanyan, J. Howard, D. Nichols, R. Sidebotham, and A. Spector, "The ITC Distributed File System: Principles and Design," *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 35-50 (December 1985).

Light-Weight Process Groups

Bradford B. Glade Kenneth P. Birman
Robert C. B. Cooper Robbert van Renesse

Computer Science Department
Cornell University, USA
{ glade | ken | rcbc | rvr }@cs.cornell.edu

Abstract

ISIS is a toolkit for building applications consisting of cooperating processes in a distributed system. Group management and group communication are two basic building blocks provided by ISIS. This approach has proven very successful, and ISIS' large user community is putting very high demands on these mechanisms. To accommodate these demands a complete redesign of the system, called HORUS, is being done to build a simpler and faster system that scales well. Of particular concern is the support and management of hundreds of thousands or more process groups. This paper describes a key component of HORUS known as *light-weight process groups* that addresses this scaling issue.

1. Introduction

With the advent of millions of PCs becoming powerful networked workstations, the support for distributed programming is sadly lagging. Many local area networks are becoming large due to the cheap price of personal computers. In such environments, failures within the network are quite commonplace. Users treat a networked PC much like a stand-alone machine, turning the machine off and rebooting when an application program fails. This behavior can quickly lead to chaos for the remaining networked computers that may depend on the machine for a source of input or service. By the very nature of such networks we are then forced to consider fault-tolerance not as a luxury for the few, but as a necessity required by all.

The ISIS toolkit is a collection of algorithms and tools that can be used to build fault-tolerant distributed applications in an environment such as the above. A description of ISIS can be found in [Bir91a]. In this paper we describe a fundamental element of a new system called HORUS[†] being built at Cornell. HORUS has evolved from ISIS after much experience with building practical fault-tolerant distributed systems.

This re‿arch was supported under DA.\PA/NASA grant NAG-2-593 and by grants from IBM, HP, Siemens, GTE, and Hitachi.

[†] In Egyptian mythology, HORUS is the son of ISIS.

This work was motivated by a trend in the use of ISIS process groups that has emerged over the last eight years. The process group paradigm has become tremendously popular with ISIS applications programmers; almost every major application written using ISIS makes extensive use of process groups. In their original design, process groups were intended as a coarse grain transport mechanism for communicating with multiple processes. Process groups were used to represent a replicated service. However, the paradigm has proven popular for more fine grain uses. Over the last few years applications written using ISIS have used process groups to represent *objects* rather than services. This trend has impacted the original design in several ways and has lead us to focus our attention on providing *light-weight process groups*.

The architecture of HORUS was influenced by microkernel design concepts, in which several light-weight mechanisms are provided in user space. The most obvious of these is the light-weight process or thread abstraction. Another well-known, older abstraction is memory allocation. These abstractions not only allow easier resource management by sharing most of a core environment, but also provide a portable interface across different environments.

The basic idea behind the light-weight process group (LWG) abstraction is that many LWGs will be mapped to a single core group (or set of core groups) as implemented by the kernel of HORUS. Thus, these LWGs will share the same security environment (much like threads share the same address space), and the same failure model, while their messages will be multiplexed over a single core group transport. The benefit of this approach is that membership changes to the core group automatically affect large numbers of LWGs, amortizing the cost of maintaining membership information over what the application considers a large number of independent groups. The ISIS system lacks such a facility, forcing many application programmers to develop equivalent mechanisms.

We have built a prototype of LWGs on top of ISIS V3.0.6 and the initial results show significant improvements in performance. In particular, the LWG subsystem allows LWGs to share the same failure detection protocol execution thereby resulting in faster reaction to member failures and reduced network load. Execution times for typical group operations are also improved: the initial prototype has a speed-up factor of 9 for the group create operation (the resulting speed is about 30 ms), and even higher speed-ups for group joins and leaves.

To motivate the problem, we present several examples of how fine grain process groups help solve problems present in distributed applications. We then briefly present the architecture of the HORUS system with particular attention to the light-weight group subsystem. We follow this with a discussion of the key aspects of light-weight process groups and present the basic portions of an interface to our subsystem. We conclude with some initial performance results.

2. Trends in the use of Process Groups

In this section we look at the use of ISIS process groups in three major applications written on top of the ISIS system. By looking at these and other applications we gained insight into how to improve the performance and functionality of process groups.

The Deceit File System

Our first example of a practical fault-tolerant distributed application is the Deceit file system [Sie92a]. Deceit is an NFS-compatible file system that replicates its exported file system across a collection of servers. The system provides flexible support for fault-tolerance. A set of parameters attached to each file controls the replication level, and update semantics of that file. As the system is used, file replicas migrate to form working sets on the servers that are currently receiving requests. Deceit's file system therefore exists as a whole across all of the servers yet no one server need contain the whole file system. A key aspect of Deceit is its ability to maintain one-copy serializability in the event of server failures and distributed requests and updates. To manage the inherent complexity of achieving such a property, Deceit uses an ISIS process group to represent the replicas of a file; each member of the group actively maintains a replica of the file. This set of servers changes dynamically as replicas migrate and as servers crash and recover.

Logically, an update to a file need only be multicast to the collection of servers maintaining replicas of that file using the ISIS process group as the transport mechanism. The initial design of this system was built in the obvious way; a single process group was associated with each file's set of replicas. It became quite apparent however that this was not the correct approach for using ISIS process groups; the system suffered greatly from performance problems. Too many process groups that were created (one for each file in the file system) and the algorithms that provide the ordering semantics of group communication were greatly affected by this (we will discuss this later).

A few observations about the collection of process groups lead us to the design decisions that contribute to the good performance of today's Deceit and to the foundation of light-weight process groups. First, good fault-tolerance was obtained with a relatively small collection of file servers. Three to five servers provide good availability, reliability, and performance. Second, even though many (thousands of) process groups were desired, the number of unique process groups, in terms of their membership, was quite small. By using a single process group for the collection of files that had the same replica set, the number of process groups was dramatically reduced with a corresponding improvement in performance. In this new design, when replicas migrated they needed to change process groups, by orchestrating the change through a coordinator in the group. Deceit was able to use the inexpensive CBCAST protocol [Bir91b] while maintaining the consistency of the file's replicas.

The ISIS Transaction Tool

The ISIS Toolkit includes a tool for distributed transactions. A transaction is represented by a process group comprising all the servers which have an interest in the outcome of the transaction (the participants). The implementation of the tool in ISIS is very straightforward. Reliable group multicast is used to implement the commit protocol, and group monitoring facilities are used to detect the failure of transaction participants and to trigger a transaction abort. To ensure that the state of a transaction persists even when all participants fail, transaction state is logged to disk, and transaction outcomes are logged to the transaction recovery manager, itself implemented by a process group.

While the semantics of ISIS process groups and reliable multicast greatly simplified the implementation of the transaction tool, performance was poor. The transaction tool needed only anonymous groups, but ISIS required every group to have a name. The transaction tool generates a known-to-be-unique name derived from the transaction identifier. ISIS incurs unnecessary costs verifying the name's uniqueness by multicasting to the ISIS servers on the network when the group is created, and searching for the name during subsequent join operations. This deficiency is fixed in HORUS, which directly supports anonymous groups and leaves naming to an external service.

More serious than group naming was the cost of a group join. The critical path of a transaction included one group join for every participant and a single group deletion at transaction end. A join involves synchronizing all the current members of the group, and possibly the authentication of the new member. One common scenario in the use of the transaction tool is for a client to issue a series of transactions to the same set of servers. After each transaction the group is torn down only to be built again by the following transaction. This creates unnecessary work when the group transport could be saved.

META

META [Woo91a, Mar91a] is a system for distributed management. It provides a mechanism for instrumenting programs with sensors and actuators and allows creating sophisticated reactive control systems in a distributed network. META makes use of ISIS for its group communication and fault-tolerance. Process groups in META are used both to maintain *aggregates* and as a convenient naming mechanism. Aggregates are used to represent a collection of machines that satisfy some property (e.g., a set of machines with a light load). This collection is maintained (determined) by a set of replicas which detect changes in the aggregate set. An ISIS process group is used to manage this replica set. Aggregates are a fundamental piece of META and are intended for heavy use by META applications, and consequently, META shows similar characteristics to Deceit: a relatively small set of replicas can be responsible for a large number of coincident process groups. Like the initial design of Deceit, the failure of a replica can trigger a flood of distributed agreement protocol invocations.

3. Analysis of Performance Problems

In general we have found that good performance can be obtained from group communication in ISIS provided that the programmer has a solid knowledge of the protocol semantics and knows the details of the implementation well enough to make optimizations. Each of the authors in the above systems are sophisticated ISIS programmers that took the semantics of the ISIS communication system and knowledge of the internal protocols into account when designing their software. In general one cannot expect typical applications programmers to be (or want to be!) as knowledgeable about ISIS as these authors. It is this that has motivated us to consider light-weight process groups as a necessary piece of the HORUS system. LWGs should allow applications programmers to use the process group paradigm in a manner which fits the logical structure of their application and which yields good performance.

We now look at why the original process group mechanism in ISIS performed poorly for these applications. The performance problems are mainly a result of the process group algorithms being too closely coupled with the interface provided to the applications builder. Three major performance problems illustrate this point.

Failure detection

ISIS provides a strong guarantee of consistency for group membership changes. A group's membership history can be characterized by a total order on the join and leave/failure events on the group. Each group member observes the membership in an order consistent with this history. In addition, ISIS provides the strong guarantee of failure atomicity; messages are delivered in the same view of the group's membership at all of its destinations. This allows the recipients to make efficient local decisions about the global state of the system without the need for extra communication. [Ric91a] and [Bir91b] present the semantics of ISIS process groups and group communication.

Figure 1 shows an example of communication with and without failure atomicity. Failure atomicity and serialized membership greatly impact the performance of process groups when failures occur. Consider the fault-tolerant NFS file server described above if it made a naive use of process groups (by creating one process group per file). At some point during the normal operation there might be a thousand or more process groups representing the files actively in use that are being maintained on three servers. If one of these servers should fail, ISIS would trigger the invocation of a failure recovery protocol on each of these one thousand groups, forcing failure atomicity on the outstanding messages, delivering them in consistent views across their recipients. Each of these protocols would force an expensive flush of the group's commu-

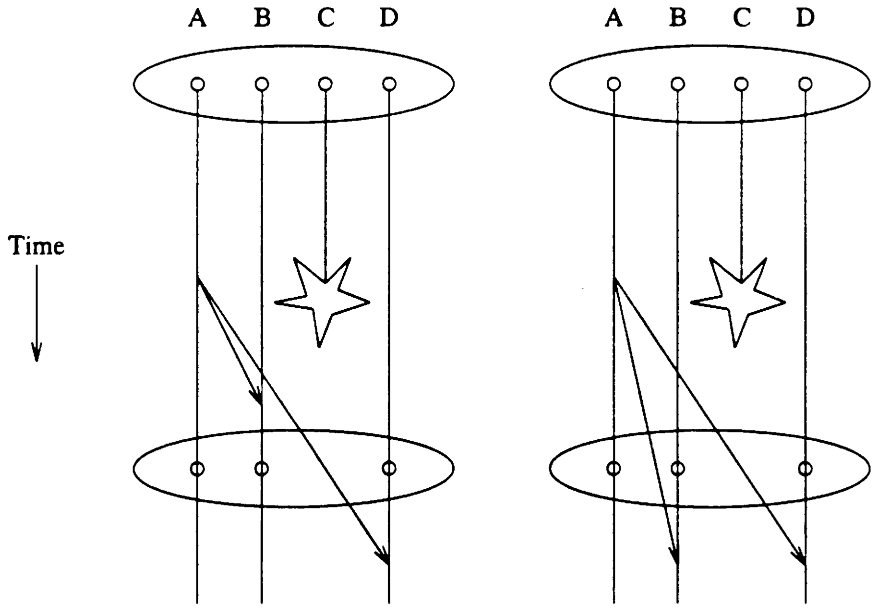


Figure 1: *Diagrams (a) and (b) show four processes, A-D, joined to a single process group, denoted by the encompassing oval. C crashes at around the same time that A sends a message to the group. (a) shows multicast communication that does not respect failure atomicity; B and D receive the message in different views of the group. The multicast in (b) respects failure atomicity.*

nication. Unfortunately this would have the disastrous impact of flooding the network with protocol messages, which can lead to very bad congestion and the ultimate "failure" of other processes in the system, causing a "domino" effect.

Overlapping Groups

ISIS provides strong causality guarantees for group communication. This guarantee applies to communication that spans groups. This is an important property of the ISIS system because it allows for less constrictive communication and allows groups to be used flexibly. [Bir91b] discusses the ramifications of this property on the algorithms that must implement it. Currently the ISIS system uses a conservative protocol. In order to send a message *m* to a group *G*, *G* must be the only "active" group. A group is active for a process *p* if there is some message *m'* to *G'* that has been transmitted by *p* or delivered to *p* and which *p* considers unstable. A process considers a message stable if it learns that the message has been received at all of its destinations. If there is more than one group active for a process, it must block the transmission of a message *m* until it all other groups become inactive (i.e. until their messages become stable). This delay may require waiting for the receipt of acknowledgements from all members of a previous multicast, and potentially for stability information from other groups. In Figure 2(a), *C* must delay its multicast to *B* and *D* until it

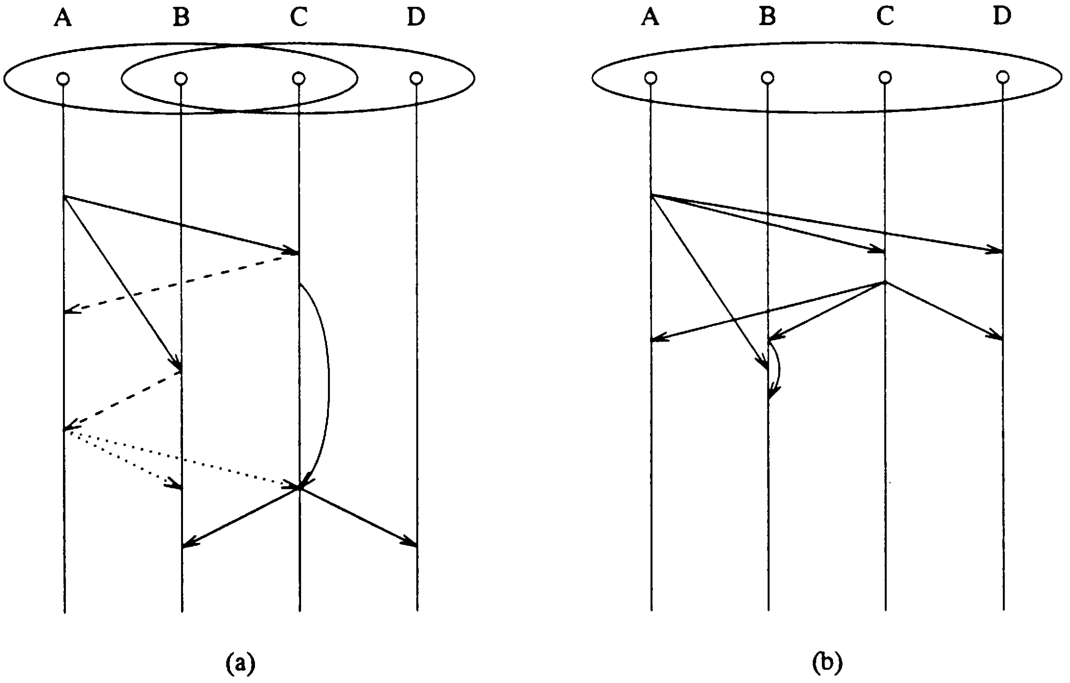


Figure 2: *Diagram (a) shows two process groups (represented by ovals) and the messages sent by the system during when communication switches from group {A,B,C} to group {B,C,D}. The solid arrows represent the application multicasts, the dashed arrows represent low-level acknowledgements, and the dotted arrows represent messages containing message stability information. Diagram (b) shows the message traffic for the same pair of application multicasts, but with the two groups merged into one. The arced arrows represent delayed messages, in (a) by the sender, and in (b) by the receiver.*

learns that the causally preceding message from A has been stably received. This delay is denoted by the arc. An application that continuously alternates communication between two groups by sending messages asynchronously, will in fact see no advantage to the asynchronous call, since each communication context switch will essentially force synchrony on the previous message send.

Named groups

Previous implementations of ISIS have incorporated the naming service into the same server process that manages the group membership protocols. This process, historically known as *protos* (for protocol server), resides on every ISIS site. (For scaling reasons ISIS V3.0 allows for remote connections that are less fault-tolerant and do not run the protocol server directly but instead connect to a "mother" ISIS site.) The implementation of the name service ensures one-copy consistency of the name space mappings among all of the protos processes. This has a great impact on the cost of creating a named group as indicated in the transaction tool discussion above.

4. Overall Design

The following observations about the common uses of process groups guided us in our design to combat these problems. We have found that many applications use

- Many process groups.
- Heavily overlapping groups.
- Both small groups and large groups.
- Unnamed groups.

With the number of groups far exceeding the number of processes in the system, high overlap and coincidence of groups is unavoidable. We observed that by combining overlapping process groups so that they share a single "core" process group, we could obtain several distinct advantages. A careful look at the performance problems shown above revealed that for the common case of identical overlapping groups, the protocols being exercised were largely unnecessary. Consider the failure reaction protocol: if a single core process group were used instead of a thousand identical groups, only a single flush would be necessary to ensure failure atomicity and instantiate the new group view. Similarly, using only a few core groups can reduce transmission delays (for obtaining stability) and thus increase truly asynchronous message sends. Much of the state maintained by the ISIS transport system to maintain causality and other ordering semantics can be shared by these light-weight groups, reducing the resource requirements of the system.

Thus there is much to be gained by separating the protocols underlying the process group implementation from the interface provided to the applications programmer. As was the solution in the above distributed systems examples, we manage a large collection of light-weight process groups by mapping them onto relatively small sets of "core" process groups. These core groups are the groups provided by the VSync (for virtually synchronous) kernel in Figure 3.

Experience with ISIS has identified the major components of the system and has allowed us to reorganize the system in a more layered and

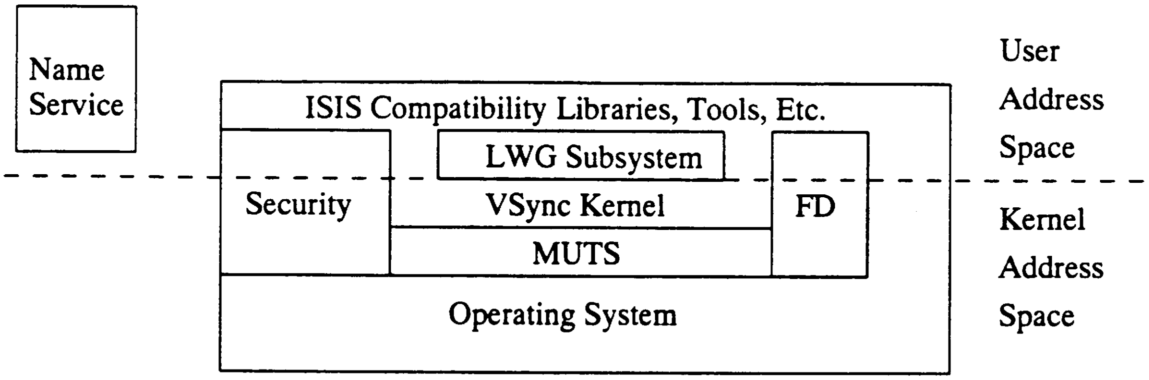


Figure 3: *The HORUS Architecture*

modular fashion in order to take advantage of the microkernel architectures being offered by modern operating systems. Figure 3 shows the design of our new architecture. The lowest layer of HORUS called MUTS (MUlticast Transport Service) [Ren92a, Ren92b], provides a portable abstraction of the underlying operating system to the higher layers. The operating system specific code is isolated in configuration dependent source files within MUTS. This foundation allows for easy porting of the system to operating systems such as Mach, Chorus, and Amoeba. A key component of MUTS is the abstraction it provides of a multicast transport service. MUTS isolates the higher layers from the details of the interprocess communication mechanism to a collection of groups, yet provides important feedback information to the higher layers so that they may deal with communication failures in a consistent, well-defined manner. Above MUTS, the VSync kernel provides ordering semantics on multicasts, and provides the basic process group abstraction with strong semantics on the ordering of group events with respect to multicasts. These two layers define the portion of the architecture that is appropriate to put in the system space of an operating system. While this is not necessary, it will probably yield more efficient communication. The layers above this are most appropriately placed in a user space library. This is where the light-weight process group subsystem lives. The subsystem provides an interface to applications through this library and is used by many of the other tools within the library itself. The library also contains tools for managing replicated data and distributed computations.

5. Design Issues

In this section we examine a number of the issues which we faced during the design of the LWG subsystem. We wanted a flexible, efficient, portable, and simple interface to the subsystem. The interface had to allow for tight control of the light-weight to core group mapping for use as a research tool and by sophisticated users, yet also allow the subsystem itself to manage this mapping in an intelligent way for ordinary users of the system. Efficiency was paramount; to be useful, the system had to optimize the critical path. In the next few sections we discuss the major issues in designing the LWG subsystem.

Mapping LWGs to Core Groups

To address the goals of flexibility and simplicity we introduced the notion of *core group sets* which can be managed by the subsystem or the user. A core group set is a collection of ISIS process groups which are used as the communication transports for light-weight groups. Light-weight groups are allocated out of a core group set and are always mapped to exactly one core group in the set. By providing routines that manipulate these sets along with options in the LWG interface, we allow for tight control of the mapping between a light-weight group and its core group. Core group sets can also be managed completely by the LWG subsystem. In this case the subsystem will add and change core groups in the set dynamically as the mapping needs of the LWGs change over time.

Core group sets allow us to address several issues at once. First, they provide flexibility. By providing support for multiple sets, varying levels of mapping control may be used within the same application. This allows different mapping policies to be enforced for different types of objects. For example, one policy might mandate that the membership of a light-weight group exactly match the membership of its core group, while another might allow LWG members to be a subset of the members of the core group. These policies will have different impacts on the performance of the system. Second, by providing policies for self-management together with a default core set, the system provides much of the functionality of light-weight groups with a simple interface. Third, by constraining LWGs to map only to those core groups within their core group set, we improve the efficiency of self-management policies by reducing the search space for core groups.

In Figure 4 we show a mapping of 3 different light-weight groups onto a common core group. It is important to note that the membership of the core group need not match the membership of the light-weight group exactly; it can be larger. However, there are tradeoffs with such mappings. If hardware multicast is not available, the cost of sending a multicast message may be greater due to the increased number of recipients. In Figure 2(b) we see that processes *A* and *D* receive extra messages which the light-weight group subsystem will need to filter out. However, these extra messages must be weighed against the acknowl-

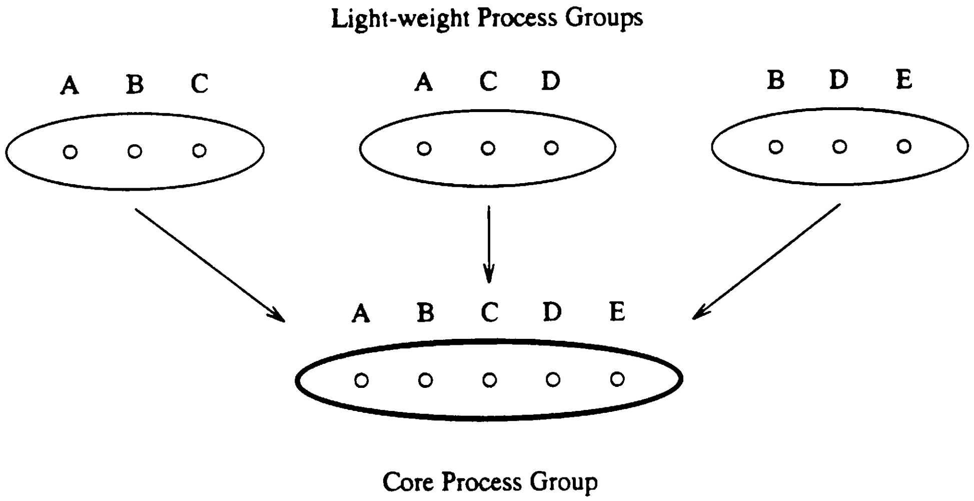


Figure 4: *A mapping of three light-weight groups onto a common core group*

edgement and stability information messages sent in diagram (a). If hardware multicast is in use, the extra members do not add to the cost of sending a message, but the extra members themselves still pay a cost for handling the receipt of the message. On the other hand, supporting "subset mappings" yields a number of advantages. First, the number of core groups that are needed is reduced since they can encompass more light-weight groups. This reduces the amount of state that is needed to support causality, reduces the number of communication context switches that occur, and reduces the size of the space that must be searched when creating a new mapping for a LWG. Second, with fewer core groups, better use can be made of Ethernet multicast addresses. This can be a critical performance factor since most Ethernet devices support a fairly limited number of multicast addresses before they go into "software" mode. Third, the cost of adding a member already in the core group to the LWG is cheaper since much of the state of the member has already been set up by the core group.

Over time core groups will have a number of different LWGs mapped to them and at some point a core group may have no LWGs that map to it. To avoid consuming too much memory, such core groups have to be garbage collected periodically. This collection could occur at the instant the set of mapped LWGs becomes empty, but leaving the core group around for some grace period can be advantageous in the event that a subsequent LWG mapping appears soon. In the transaction tool this would do well on the common scenario where a client issues a series of transactions to the same set of servers, if the grace period is longer than the time between transactions. Thus we could exploit temporal, as well as spatial, locality of transactions.

Under high load conditions the LWG subsystem can be faced with a potentially large search problem. Upon the creation of a LWG with an initial set of members, it must map this group to an existing core group, if possible. Determining the best mapping can, without using good search techniques, lead to a linear search of the core group set, which in the worst case can be quite expensive (for n processes, there are potentially $2^n - 1$ unique core groups). In practice such a large number of core groups never exists since the presence of subset mappings eliminates the need for many of these groups. In any case, the LWG subsystem manages this search by using a hash index scheme keyed on the membership of the group. This enables the search to quickly narrow in on a core group containing the right members.

Added Functionality

Rewriting ISIS gives us the opportunity to consider providing different forms of group semantics. ISIS provides a broad range of ordering semantics for its communication (MBCAST, FBCAST, CBCAST, ABCAST, and GBCAST) [Gro91a], yet only one set of semantics is provided for the process group mechanism. While it can rightfully be argued that too many choices only leads to the confusion of the programmer, it is nonetheless interesting to consider the use of this subsystem as tool for research into a spectrum of process group semantics. An example clearly establishes the validity of this argument. We have observed that while many applications benefit greatly from the strong semantics of ISIS process groups, there are nonetheless a number of applications for which these semantics are too strong and which would benefit from the performance improvements obtained by using weaker semantics. Consider a collection of sensor processes responsible for periodically sensing the temperature of a room and reporting on these

Function	*Arguments*	*Result*	*Description*
lwg_create	initial members	lwg	Create light-weight group.
lwg_add	members	–	Add members to a group.
lwg_remove	members	–	Remove members.
lwg_destroy	lwg	–	Destroy group.
lwg_send	lwg, msg	send_id	Post message to a group.
lwg_receive	lwg	msg, recv_id	Wait for next message.
lwg_reply	recv_id, reply msg	–	Send a reply.
lwg_get_next_reply	send_id	msg	Wait for next reply.
lwg_discard_replies	send_id	–	No more replies wanted.

Figure 5: *The light-weight group interface*

values to a collection of reader processes. For fault-tolerance multiple sensors are used, and the reader processes collect the sensor data to determine an average for the room's temperature. Here an ISIS process group may be used as the group communication transport. The sensor processes would, on initialization, join the group and start broadcasting data. Notice, however, that the sensors themselves use the group for sending only; they do not need to obtain state from other members and are not concerned about the order in which they join the group. In this situation ISIS would completely order the joins when in fact this is not needed.

Large Numbers of Process Groups

Just as light-weight threads share their state within the address space of their encompassing process, light-weight groups share their causality context and group data structures within their core group. The reduced memory resource needs combined with the sharing of the core group protocols for failure detection and causality allow HORUS to efficiently support many more light-weight process groups than core groups.

6. Interface

Figure 5 shows the interface to the light-weight group subsystem. This interface provides asynchronous results to enable the application to take advantage of pipelining to improve its efficiency and yet retain a simple model of execution.

7. Initial Performance Results

As a proof of concept, we built a prototype of the light-weight group subsystem on top of ISIS V3.0.6. Doing so allowed us to proceed with our research testing in parallel with the building of the HORUS system, which is being built bottom up. The lowest layers of HORUS are almost now complete and the building of the light-weight group subsystem on top of HORUS is just beginning. Building the prototype on top of ISIS V3.0.6 allows us to make measurements of the impact of the LWG subsystem on the performance of the system. Happily, the prototype showed significant improvements in performance and the results supported our initial suspicions.

Initial measurements of the performance of our light-weight process group subsystem are encouraging. The following measurements were taken on Sun 4c/60 Sparc 1+ workstations running Sun OS 4.1.1 using ISIS V3.0.6.

Our measurements of the cost of obtaining message stability confirmed our initial expectations. Switching communication from one core group to another core group costs the application approximately one synchronous multicast. For applications that change contexts frequently with respect to message sends, this overhead can be significant. For example, a process that repeatedly switches between to coincident core groups runs roughly twice as long as the equivalent program sending to only one core group. Asynchronously CBCASTing 400 byte messages to 4 members (3 remote, 1 local) costs 18.0 ms per multicast in the strictly alternating case, and only 10.4 ms in the single group streaming case. For 2 members (1 remote, 1 local), the cost of alternating CBCASTs is 10 ms, for streaming it is 3.2 ms. The tuning of the transport layer plays an important factor in the cost of obtaining stability. For efficiency the transport layer will attempt to determine if the sending application is in a streaming or "interactive" mode. In the former, the transport layer will delay acknowledgements in order to send as few ack messages as possible, in the latter case the transport layer is aggressive about sending acks, so that the cost of the context switch is as small as possible.

To measure the effect of light-weight groups on reducing the costs of a join, we compared creating bursts of 100 LWGs vs. core groups. The prototype LWG subsystem makes use of a group view manager which replaces the role of "protos" for managing views and group names. We ran these tests with the creating process both local and remote to the view manager. In the local case, a LWG create took 45 ms compared to 60 ms. In the remote case, a LWG create took 29 ms compared to 200 ms for the core group. Contention for the processor may partially explain why the LWG create with the local view manager is more expensive than the remote case, but this is still curious. These results are preliminary and only serve as proof of concept. The LWG subsystem on HORUS will not use a group view manager and will use a separate name service for named groups.

We measured the time of a light-weight group leave event for both the local and remote view manager cases. Under both situations the cost of a light-weight group leave was 9 ms. The cost of a core group leave for the remote case was 197 ms, and for the local leave it was 80 ms.

8. Conclusion

It is interesting to draw analogies with the evolution of some other common system paradigms. Memory allocation is an excellent example. Before the advent of standard library routines like malloc, programmers were forced to implement their own memory allocator routines which usually had the effect of reducing the portability of their software, since their memory allocators were often OS and machine specific. Today, malloc is widely available, and the mechanisms by which memory is allocated are hardly a concern to most programmers. Much like malloc, light-weight process groups abstract away the details of the implementation, yet provide added functionality and improved performance.

Similarly, threads have become an attractive mechanism for improving the performance of processes. Threads reduce the heavy-weight context switching of processes by sharing an address space among the threads of control. The sharing of resources seems to be a common theme to providing light-weight mechanisms. We are encouraged by the initial results of our prototype and are actively incorporating these ideas into HORUS.

Currently, we are actively experimenting with prototype and are building the light-weight process subsystem and user-level libraries on top of the VSync kernel in HORUS. We hope to have a release of this system available by the end of 1992.

Acknowledgements

The authors of this paper gratefully acknowledge the contributions of Sue Honig, Keith Marzullo, Aleta Ricciardi, Alex Siegel, Patrick Stephenson, and Mark Wood to ISIS and its applications. We also greatly appreciate the helpful comments of Cliff Krumvieda and Mike Reiter on an earlier draft of this paper.

References

[Bir91a] Kenneth Birman, *The Process Group Approach to Reliable Distributed Computing,* Dept. of Computer Science, Cornell University (July 1991). Submitted to Communications of the ACM

[Bir91b] Kenneth Birman, Andre Schiper, and Patrick Stephenson, "Lightweight Causal and Atomic Group Multicast," *ACM Transactions on Computer Systems* (August 1991).

[Gro91a] The ISIS Group, *The ISIS Distributed Toolkit Version 3.0 User Reference Manual,* Dept. of Computer Science, Cornell University, Ithaca, NY (May 1991).

[Mar91a] Keith Marzullo, Robert Cooper, Mark Wood, and Ken Birman, "Tools for Distributed Application Management," *Computer* **24**(8), pp. 42-51 (August, 1991).

[Ren92a] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson, "Reliable Multicast between Microkernels," *Proc. of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, Washington, pp. 269-283 (April 27-28, 1992).

[Ren92b] R. van Renesse, R. Cooper, B. Glade, and P. Stephenson, "A RISC Approach to Process Groups," *Proc. of the Fifth ACM SIGOPS European Workshop*, Le Mont Saint-Michel, France (September 21-23, 1992).

[Ric91a] Aleta Ricciardi and Kenneth Birman, "Using Process Groups to Implement Failure Detection in Asynchronous Environments," *Proc. of the Eleventh ACM Symposium on the Principles of Distributed Computing*, Montreal, pp. 341-353 (August 1991).

[Sie92a] Alexander Siegel, "Performance in Flexible Distributed File Systems," PhD Thesis, 92-1266, Dept. of Cc .nputer Science, Cornell University, Ithaca, NY (February 1992).

[Woo91a] Mark Wood, "Fault-Tolerant Management of Distributed Applications Using the Reactive System Architecture," PhD Thesis, 91-1252, Dept. of Computer Science, Cornell University, Ithaca, NY (December 1991).

Fast Group Communication for Standard Workstations

Werner Vogels
INESC[†]
Lisboa, Portugal
werner@inesc.pt

Luis Rodrigues   Paulo Veríssimo
Technical University of Lisboa
INESC, Lisboa, Portugal
{ ler | paulov }@inesc.pt

Abstract

This paper presents a Group Communication Service suitable for standard workstations. The communication service is designed to take advantage of the technology offered by modern standard Local Area Networks and offers a very versatile multi-primitive interface to its users. The authors focus on the design and implementation of the communication service, and of the software modules necessary to exploit specific network and operating system properties. Additionally performance results are given and evaluated in the context of comparable systems.

1. Introduction

Increasing use of distributed systems, with the corresponding decentralization of activities, stimulates the need for structuring those activities around groups of participants, for reasons of consistency, user-friendliness, performance and dependability. The concept appears intuitively in all flavors of distributed actions: when participants cooperate in an activity (e.g. management of a partioned database, shared document processing or distributed process control), compete for a given activity (e.g. distributed use of a resource), or execute a replicated activity for performance or fault-tolerance reasons (e.g. replicated database server, replicated actuator).

The group paradigm is widely accepted as being an excellent method of structuring these distributed activities. From the pioneering projects

[†] Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 – 6° – 1000 Lisboa – Portugal, Tel.+351-1-3100281. This work has been supported in part by JNICT, through Programa Ciência.

in the past [Bir91a, Coo85a, Che85a], a large number of research projects in areas related with group structuring and reliable group communication have emerged [Bir91b, Cri90a, Gar89a, Her89a, Pet89a, Pow91a]. The Distributed Systems and Industrial Automation group at INESC has contributed to the evolution of the group paradigm by focusing on the development of highly responsive group communication and management protocols [Ver92a].

To support the development of systems and applications that rely on distributed paradigms, we have developed a Group Communication Service. Originally designed and developed as part of the Delta-4† ESPRIT project [Pow91a], an effort has been undertaken to make the same service available for standard workstations. The results of this effort have yielded a group communication module suitable for integration in UNIX kernels. Prototype implementations have been made for the SunOS 4.1.1 and the Mach 2.5 kernels.

In this paper we discuss the global design and implementation of the Group Communication Service and related modules. The next section describes our approach to group communication in general, followed by a section on the actual design of the service. The different modules that found the basis of the service are each described in separate sections after the section on design. Section 10 will deal with the formal specification and verification of the protocols. In Sections 11 and 12 we present the performance of our protocols and evaluate these results by comparing them to other group communication systems.

2. The Group Communication Service

The need for support of group activity is based on the assumption, shown correct by a number of real examples, that in a distributed architecture processes frequently get together to achieve a common goal. The set of such processes can be called a *group*. A communication service can be said to support groups when it provides services that facilitate the design and the execution of distributed software running on such a group of distributed processes in cooperation, competition or replication [Ver92a].

The Group Communication Service described in this paper is based on three essential services [Rod92a]:

- The first services required in a group communication service are, naturally, the *group membership* services. Powerful support for groups is given to allow the dynamic creation – and reconfiguration – of process groups. During the lifetime of a group, processes may join or leave the group and the communications service provides primitives to perform these operations. The failure of a group member is also detected and an indication of the event is provided to the remaining members.
- The second goal of the group communication service is to provide efficient and versatile support for exchange of information between group members. To start with, a *multicast* communication service avoids the need to explicitly perform point-to-point transfers to execute a multicast operation. The service accepts a list of addresses, what we call a *selective address*, as a valid destination address for a multicast message and – transparently –

† Delta-4, ended in December 1991, was a CEC Esprit II consortium, formed by Ferranti-CSL (GB), Bull (F), Credit Agricole (F), IEI (I), IITB (D), INESC (P), LAAS (F), LGI (F), MARI (GB), NCSR (GB), Renault (F), SEMA (F), Un. of Newcastle (GB), designing an open, dependable, distributed architecture.

Consistent Group View

Px1 Each change to group membership is indicated by a message obeying total order, to all correct group participants within a known and bounded time T_g.

Addressing

Px2 **Selective addressing:** The recipients of any message are identified by a pair (g,sl), where g is a group identification and sl is a *selective address* (a list of physical addresses).

Px3 **Logical addressing:** For each group g there is a mapping between g and an address A_g, such that A_g allows all correct members of g to be addressed without the knowledge by the sender of their number or physical identification.

Validity

Px4 **Non-triviality:** Any message delivered, was sent by a correct participant.

Px5 **Accessibility:** Any message delivered, was delivered to a participant correct and accessible for that message.

Px6 **Delivery:** Any message is delivered, unless the sender fails, or some participant(s) is(are) inaccessible.

Synchronism

Px7 The time between any service invocation and the (eventual) subsequent indication at any recipient (T_e), as well as the time between any two such (eventual) indications (T_i), are:
– *Loose synchronism:* ΔT_e and ΔT_i may be not negligible, in relation to max T_e.
– *Tight synchronism:* ΔT_e and ΔT_i are negligible, in relation to max T_e

Agreement

Px8 **Unanimity:** Any message delivered to a participant, is delivered to all correct addressed participants.

Px9 **At-least-N:** Any message delivered to a recipient, is delivered to at least N correct recipients.

Px9.1 **At-least-To:** Given a subset P_{addr} of the recipients, any message delivered to a recipient, is delivered to all correct recipients in P_{addr}.

Px10 **Best-effort-N:** Any message delivered to a recipient, is delivered to at least N correct recipients, in absence of sender failure.

Px10.1 **Best-effort-To:** Given a subset P_{addr} of the recipients, any message delivered to a recipient, is delivered to all correct recipients in P_{addr}, in absence of sender failure.

Order

Px11 **Total order:** Any two messages delivered to any correct recipients, are delivered in the same order to those recipients.

Px12 **Causal order:** Any two messages, delivered to any correct participants of any group, are delivered in their "precedes" order.

Px13 **FIFO order:** If any two messages from the same participant, are delivered to any correct recipient, they are delivered in the order they were sent.

Table 1: *Group communication properties*

delivers the message to the intended recipients. Additionally, a *logical address* can be associated with a multicast group, allowing all group members to be addressed through a *logical name*. This frees the programmer from having to deal explicitly with selective address lists. Note that a logical name can be seen as a pre-defined address list, containing the addresses of all group members, and being constantly updated upon every group change.

- The third goal of the group communication service is to provide an execution environment that applies algorithms to ensure a given set of desirable properties.[†] These properties are summarized in Table 1. Validity and synchronism properties (Px4, Px5, Px6 and Px7) are desirable in most communication systems. They usually state that the user can trust the system in the sense that messages are not corrupted, arbitrarily lost or spontaneously generated. Synchronism properties assure that the service is provided within known time bounds. Timely behavior of the protocol is of major relevance in real-time systems. Agreement properties describe when, and to whom, a multicast message must be delivered. The strongest property in this set is *unanimity* (Px8). Unanimity states that a message, if delivered to a correct participant, will be delivered to all other correct participants despite the occurrence of faults. This may be stronger than usually required. For instance, queries to replicated servers need only reach one of the replicas, since all responses would be the same. Quorum-based protocols are another example where unanimity is not required. This raised the need to provide different agreement properties (Px9 and Px10). Finally, order properties specify which ordering disciplines the protocol should impose on the messages exchanged between group members. The stronger property, *total order* (Px11) assures that the messages are delivered in the same order to different participants. Causal (Px12) and FIFO (Px13) are different, less costly, ordering disciplines that can provide better performance for those applications not requiring total order.

Clearly, all these different requirements cannot be provided in an efficient manner by a single communication primitive. That is why the Group Communication Service provides several qualities of service [Rod91a]:

- **Best-Effort**. Acknowledged datagrams with retries to reach a certain quorum. Quorum can be set by either a number of members or a subset of addressees.
- **Reliable**. Acknowledged datagrams with retries and Quorum specification like in Best-effort but with guarantee of delivering even if the sender fails.
- **Causal**. Reliable quality of service respecting the "happened before" order.
- **Atomic**. Datagrams delivered to all members (including the sender) or none with total order within the group.
- **Tight**. Total order datagrams within a group with queue re-ordering for priority handling of messages and approximate same time delivery.
- **Delta**. Support for total order of messages based on global time (achieved by synchronized virtual clocks).

3. Design

The design of the Group Communication Service was driven by a number of goals:

- Exploitation of technology offered by the network infrastructure.
- Offer a versatile set of primitives that can satisfy all application requirements regarding group communication.

[†] For a more detailed study the reader is referred to [Ver89a].

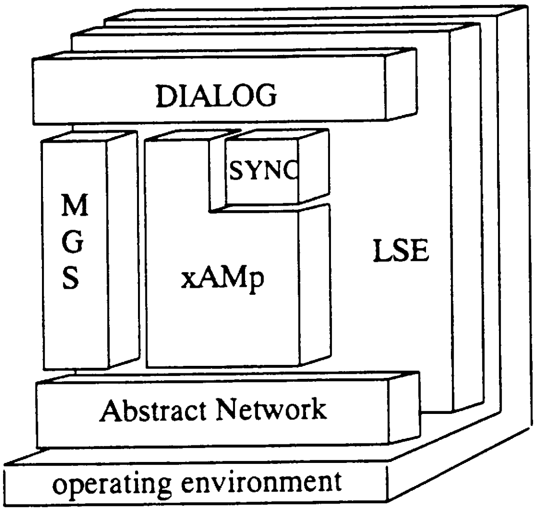


Figure 1: *Modules in the Group Communication Service*

- Highly responsive behavior of all primitives.
- Entry points at all layers are accessible by the user of the service.

Another goal was to design the Group Communication System as a highly portable software system suitable for integration in several different operating environments. User level programming environments are getting more and more standardized, and it is becoming easier to develop software that is portable at this level. At the operating system level the situation is the contrary, manufactures are moving away from the original base (often BSD UNIX) making it more difficult to built portable kernel modules. We have gone through considerable effort to design our Group Communication Service in such a way that the core part of the system is highly portable, and is surrounded by a number of well defined modules that implement the environment dependencies.

The following modules are part of the Group Communication Service (see also Figure 1):

- **Local Support Environment** – Offers an environment independent interface to system specific functions, such as memory allocation, timers, buffer management and event handling [Fon90a].
- **Abstract Network** – This modules implements all network properties common to networks that need to support group communication [Ruf91a]. It handles address management, the sending of messages, filtering of incoming messages, and supplies support for algorithms that are based on properties like *bounded execution time*.
- ***x*AMp** – The core protocol kernel, implementing the Qualities of Service described in Section 2 [Rod92a].
- MGS – The Multicast Group of Stations protocol. This is a low level processor group membership protocol designed to support membership and addressing techniques [Rod92b].
- SYNC The Clock Synchronization Service. Algorithms are implemented to achieve synchronized virtual clocks, creating a global time base [Rod91b].
- **Dialog** This is the interface module to make the system work with each of the three standard UNIX communication interfaces *socket*, *streams* and *device driver*.

Each of these modules is described in detail in the following sections. In nearly all sections we provide some implementation details. These details are related to the SunOS and MACH 2.5 ports. Ports to other environments have also been made but are out of the scope of this paper as the are not considered as "standard workstations".

4. Local Support Environment

The Group Communication Service is designed to be environment independent, resulting in a highly portable protocol core that has no dependencies to a particular operating environment and a well described interface that covers all possible environment specific interactions. Minimal porting efforts are needed to bring the service to another environment.

To be able to shield the protocols from all environment dependencies a *Local Support Environment* (LSE) has been developed [Fon90a]. This LSE is not only a product of theoretical design, but it reflects our experiences with porting the Group Communication Service to different platforms. Especially in the areas of timer and buffer management the design of the LSE modules have undergone substantial changes through the years, to arrive at a point where they have become generic packages, usable by designers of any protocol, offering more functionality then the underlying operating system provides.

In addition to the modules that implement the interface to the environment dependent system parts, a number of generic data structure handling routines have been integrated into the LSE, adding easy to use *pools, lists, etc* to the protocol development environment. There is an overhead in making these data structure handling routines generic, but during the design phase it shortens the prototyping path. If during profiling it turns out that the generic manipulation introduces a substantial performance penalty, dedicated implementations of the data structure handling routines are built.

When porting to a different operating system environment, the dependent parts of the LSE need to be re-implemented to match the new environment. The environment dependent modules include:

4.1. Buffer Management

How to construct and manipulate messages is of extreme importance when designing high-performance protocols [Bir84a, Che88a, Hut89a, Dru92a, Sch89a, Ber89a]. Former research pointed out that operations on message buffers are often bottlenecks in the performance of network software. Especially the copy operations are to be avoided.

Although an effort has been made to design the LSE buffer management as efficient as possible, the *avoid to copy* rule dominates the design, resulting in that the operating system specific buffer management scheme is left intact as much as possible.

In the UNIX kernel and the MACH macrokernel versions this resulted in using the LSE buffer management as a frontend to operations on *mbuf's*. The only need for copying is from user to kernel space and from kernel to device space. These are, given the current structure of UNIX, the minimum number of operations that you have to apply. The new mbuf scheme in SunOS 4.1.1, which makes it possible to add private manipulation functions to arbitrary sized mbuf's, looks promising for designing dedicated buffer management. But for the time being it is inadequate because the network devices will still use the old scheme for assigning frames to buffers. To convert to the new style mbuf's an extra copy operation is needed.

The LSE buffer management is implemented as a regular mbuf chain in which the first mbuf has data size zero (0) and contains only administrative data for buffer manipulation. When passing the mbuf chain to standard kernel routines, these routines will discard this first mbuf.

The user of the buffer management is presented a contiguous buffer in which read and write operations can be done at any desired location and headers and tails can be added and removed without caring about the mbuf representation.

4.2. Memory Management

This module presents an interface to allocating and releasing pieces of memory. The routines in this module are merely function calls to the system routines that perform memory allocation. No direct translation can be made because often the *malloc* and *free* calls have different semantics when used in different environments. The SunOS kernel version of *free* for example expects the number of released bytes to be given as an parameter, while most higher level versions of this routine can be satisfied with just an address of the memory block. Some clever tricks have to be used if one wants to keep the interface as efficient as possible. In the UNIX kernel the memory is taken of the kernel heap, which turns out to be out an expensive operation. This justified the design of a local memory management package that would overcome all these difficulties, but adding this type of complexity does not outweigh the advantage. When designing the protocols care has been taken only to use dynamic allocation in startup phases and when there is not critical impact on protocol performance.

4.3. Timer Management

A module with standard timer operations is based on operations on a delta list of timers using the kernel *timeout* function to fire a timer interrupt function.

Timers can be created, destroyed, started and stopped. Two types of timers are available:

- *A-synchronous* timers which execute a registered function at the moment they expire.
- *Synchronous* timers which will send a timeout message to a message queue once they expire.

Using synchronous timers can enhance the simplicity of the protocol code as there is no need for complex interrupt handling of timer triggered routines. Concurrency is locked out of the design of the protocol state machines to simplify the state transition mechanisms.

4.4. Debug and Logging Management

This module offers convenient routines for printing warnings, errors and debug statements. It also provides interface for time measurement to enable intra-kernel performance management.

When operating normally or with a small number of debug messages the system makes use of the syslog facility or writes directly to the console device. When the number of debug messages is expected to become high, the messages can be send to a special *xamp-debug* device that handles the messages very efficiently. Extreme care has to be taken when writing verbose debug messages to the console of a SUN workstation, as this console is so slow that it can effectively block the execution of protocols when printing for longer periods, making the protocol deaf and dumb for unacceptable periods.

4.5. Generic Data Structures

As described earlier the LSE also has a small set of generic data structure handling routines:

- **Pools** A pool of a certain type of data structures can be created, data units can be requested and returned to the pool, (re-) initialization routines can be specified to be called each time a data unit is returned to the pool.
- **Queues** A collection of generic single and double linked lists routines.
- **Plists** A list of data structures that reside in a pool linked by a list.

5. Abstract Network

At the basic to the design of the Group Communication Service is the strategy to take advantage of Local Area Network (LAN) technology, using different types of LANs like 8804-4 token-bus [ISO85a], 8802-5 token ring [ISO85b], FDDI [X3T86a] and Ethernet [ISO85c]. Although these LANs are quite different in their use of technology, one can determine a general set of properties that are to be offered by every LAN [Ver91a]. The *Abstract Network* is used to hide the LAN specific details from the protocol environment [Ruf91a], exporting a number of helpful (Table 2) properties that are used to implement the properties of the group communication protocols. These abstract network properties are partially provided by the LAN technology and is complemented by additional software.

The properties Pn1 and Pn2 guarantee detection of erroneous delivery by the LAN in case of the broadcast/multicast case. Properties Pn4 and

Pn1 *Broadcast:* Destinations receiving an uncorrupted frame transmission, receive the same frame.

Pn2 *Error detection:* Destinations detect any corruption by the network in a locally received frame.

Pn3 *Bounded omission degree:* In a network with *N* nodes, in a known interval, corresponding to *(k+1)* series of unordered transmissions, such that each of the *N* access points transmits one frame per series, all transmissions are indicated in all destination access points, in at least one series.

Pn4 *Full duplex:* Indication, at a destination access point, of frame reception, during transmission by the local source access point, may be provided, on request.

Pn5 *Network order:* Any two frames indicated in two different destination access points, are indicated in the same order.

Pn6 *Bounded transmission delay:* Every frame queued at a source access point, is transmitted by the network within a bounded delay.

Table 2: *Network Properties*

Pn5 are the foundation for the ordering properties of the group communication protocols. Pn3 and Pn6 define the behavior in the time domain, Pn3 denotes a *bounded omission degree*, based on failure detection and fault treatment, Pn6 depends on the particular network, its sizing, parameterizing and loading conditions. The *Abstract Network*, in a sense, extends the concept of LLC,[†] the LAN independent sublayer of the IEEE, and later ISO 802 standard [ISO85d].

5.1. Abstract Network Primitives

The user of the abstract network service has a number of primitives available for interaction with the network [Ruf91b].

- *Data Request primitives.* These primitives request the transmission of a frame.
    - ♦ **Group request** – multicast this frame to all members of a given group.
    - ♦ **Selective request** – multicast this frame to a given subset of the members of a group.
    - ♦ **Individual request** – send the frame to a specified station.
- *Data Receive primitives* – Indications of data from the network, or confirmations from the network interface if a given data request has been served or not.
- *Network Management primitives* – These primitives provide interaction with manageable objects in the abstract network like addresses, network sizing, load control, traffic monitoring, protocol characterization. All objects can be read, some can be set to new or predefined values.
- *Station Management primitives* – a number of primitives manipulate the stations presence on the network and the management of the multicast address space it will receive frames on.
    - ♦ **Stations** can be inserted or removed from the network. The routines initialize or shutdown the internal abstract network protocols and control the presence of the station on the network.
    - ♦ **Groups** can be opened and closed. This is the management of the multicast address space using either hardware or software selection.
    - ♦ **Selective** addresses can be set or removed. These are the identifiers used as the station selective address, used in subset addressing.
    - ♦ **Fault injection** mechanisms like making the station *deaf* or *dumb* to introduce faults for protocol testing.
- *Notification primitives* for flow control, network failure detection and station management.

5.2. Abstract Network Implementation

The Abstract Network presents the user with an interface to the real network network, through use of the primitives from the previous section. But not all network controller give the designer the same set of mechanisms to implement the Abstract Network, often additional software is needed to implement all properties of the Network correctly.

[†] Logical Link Control sublayer.

Implementations have been made for token-bus, token-ring and Ethernet, an experimental implementation for FDDI is in progress.

The abstract network is implemented independent from the *x*AMp protocol suite. Within SunOS the only abstract network implemented are of the Ethernet type (and LAN class, see [Ver92a]). Within each station a number of abstract networks instances is available to which a higher level protocol can connect, either directly from within the kernel or from user space through a device driver interface. This way the abstract network is not only available for the group communication service for can be used for other types of protocol development as well.

Binding of a protocol to an abstract network is done dynamically, and after this binding the abstract network instance is initialized to use a specified network interface (corresponding to its type, only Ethernet in the SunOS case) and to use a specified network type identifier (the protocol field in the Ethernet frame header).

An important aspect of the abstract network is the management of the multicast address space [Vog91a, Vog92a]. To all extend one should avoid using *broadcast* or *all-multicast* modes of the network controller, as one looses the advantage of hardware multicast address filtering. If this can not be avoided there are two possible schemes;

- All stations receive all messages from all other stations participating in the conversation, and address filtering is done by software.
- Messages are send using multiple point-to-point messages.

In both cases the real advantages of hardware multicast are nullified.

Per network interface a module handles the multicast address management for all abstract networks connected to that interface. For the Ethernet case there is a mapping between group identifiers and the multicast address, in contrast with the token-bus implementation made for the SPART/UE real-time environment where the *selective* address is part of the hardware address filtering scheme.

In the Ethernet version the selective address filtering is done by software. It are simple, inexpensive bit masking manipulations. Although the selective address for a particular abstract network can be altered, it is implicitly connected to the selective address assigned by the MGS to this station.

For each interface a number of statistics are kept to be able to identify load, sizing, error rate etc. This information is used to compute round trip estimates, omission timeouts, transmission delays, etc.

6. The Group Communication Protocol

The core of the Group Communication Service is the *eXtended Atomic Multicast Protocol* (*x*AMp) [Rod92a], which offers a number of qualities of service as described in Section 2 [Rod91a]. The selection of these QOS's was driven by user requirements put by diverse classes of distributed applications. These requirements arisen from the literature and largely from the needs of the group replication and membership protocols of Delta-4 architecture.

In the following Sections we describe the basic transmission procedure and its use by a number of the qualities of service.

```
0   // tr − w − resp ($m$, ord, send, $P_r$, $n_r$, $M_r$)
1   // "m" is a message to be sent. ($D_{(m)}$ is the set of recipients).
2   // "ord" is a boolean specifying if network order is relevant.
3   // "send" allows the first transmission to be skipped.
4   // $P_r$ is a set of processors from which a response is expected.
5   // $n_r$ is the number of responses expected.
6   // (usually $n_r = \#P_r$ ; $P_r = D_{(m)}$)
7   // $M_r$ is a bag of responses
8
9   retries := 0;
10  do // while
11     if(retries = 0 | ord) then
12        $P_w := P_r$ ; $n_w := n_r$ ; $M_r := 0$ ;
13     fi
14     if(retries > 0 |send) then send(m); fi
15     timeout := 0; start a timer; // wait responses
16     while ($n_w > 0$ & ¬ timeout) do
17        when response ($r_m$) received from $p$ & $p \in P_r$ do
18           add($r_m$) to $M_r$;$n_w := n_w - 1$;
19           remove $p$ from $P_w$; od
20        when timer expires do
21           timeout := 1; od
22     od
23     retries := retries + 1;
24  while(retries < MAX & $n_w > 0$)
25
26  if($n_w > 0$) then check membership fi
```

Figure 2: *Transmission with response (tr-w-resp) procedure*

6.1. The Transmission with Response Procedure

Basic to the *x*AMp is the use of the abstract network service which offers an *unreliable multicast* service. In absence of faults the *broadcast* (Pn1), *full duplex* (Pn4) and *network order* (Pn5) properties of the abstract network provide message delivery at all connected stations in the same order. However, although errors can be considered rare in LANs, the occasional loss of messages – or omissions – cannot be prevented. Thus, the communication service must be able to recover from such errors. In the *x*AMp, omission errors are detected and recovered using a transmission with response procedure: it uses acknowledgments to confirm the reception of the message and detects omission errors based on the *bounded omission degree* property of the abstract network.[†]

The *tr-w-resp* procedure[‡] is depicted in Figure 2. It consists of a loop, where the data message is sent over the network and responses are awaited for. The procedure waits during a pre-defined time interval for the responses (l.15), which are then inserted in a response bag (l.17) and exits when the desired number of responses is collected. If some responses are missing, the response bag is re-initialized (l.12) and the message re-transmitted. The main loop finishes when all the intended

[†] The detailed technique, as well as its advantages over other approaches such as diffusion based masking is discussed in detail in [V.r91a].

[‡] It is a modified version of the procedure given in [Ver90a].

responses are received or when a pre-defined retry value is reached (1.23).

To preserve *network order*, the procedure re-transmits the message until it is acknowledged by all recipients in a same transmission. When order is not required, the procedure can be optimized by keeping responses in the bag from one re-transmission to the other (response messages are inserted only once in the response bag). For some omission patterns, this would allow the bag to be filled faster. To activate this mode, the flag **ord** must be set to false. Finally, the boolean variable **send** allows the user to specify that the message should be sent over the network on the first cycle of the procedure. This parameter is useful to allow another processors to collect responses – and execute the procedure – on behalf of the sender without immediately re-transmitting the message. In later sections we explain how this feature is used to provide some of *x*AMp qualities of service.

Several transmissions with response can be executing simultaneously, on the same or on different machines. We assume that messages can be uniquely identified. Different re-transmissions of the same message can also be distinguished. It is thus possible to route any response to the appropriate *tr-w-resp* instantiation (also called an *emitter-machine*).† To make a protocol tolerant to sender crashes, several emitter-machines may be activated concurrently, at recipients sites, for a same message transmission (in this case, responses must be also broadcasted). See *atLeast* agreement for an example.

6.2. *BestEffort* and *atLeast* Qualities of Service

A number of distributed applications do not need communication primitives that provide very strong order and agreement primitives, but do want to use the efficient dissemination of messages to a group of stations. To give support for this type of application demands the *x*AMp offers the *bestEffort* and *atLeast* primitives.

BestEffort is used to simply send a message to a group of stations. The user can specify the number of responses needed (n_r), or which named subset of addressees (P_r) needs to acknowledge the message. If the number of requested responses is zero the service is equivalent to an *unreliable multicast* service.

The *bestEffort* quality of service is not able to assure delivery in case of sender failure. In order to provide assured delivery, in the presence of sender failures, we make every recipient responsible for the termination of the protocol. In consequence, *tr-w-resp* is invoked both at the sender and at the recipients, as depicted in the figure (1.7). However, to avoid superfluous re-transmissions of the data message, recipients skip the first step of the *tr-w-resp* procedure, using the **send** boolean parameter). In the no fault case, the data message will be acknowledged by all intended recipients, these acknowledgments will be seen by the all the participants and no retransmission takes place.‡ As with *bestEffort* several variants on agreement are possible by choosing the set of stations that need to respond (P_r) or the number of responses (n_r) needed.

† Since several emitter-machines can run in parallel, the protocol implementation is able to execute several user requests at the same time. However, since a node usually has limited resources (memory and cpu), the implementation may restrict the number of simultaneous transmissions, for instance keeping a fixed size pool of emitter machines. Some qualities of service may impose additional restrictions on parallelism.

‡ This algorithm can be improved to avoid multiple retransmissions when a single omission occurs, by making the recipients use slightly different timeout values, and making the protocol refraining from re-sending when a retransmission from other participant is detected before the timeout expires.

If the number of responses needed is smaller then the set of addressed stations ($n_r \le \#D_{(m)}$), the primitive will assure that *at least* that number of the addressees receive the frame even if the sender fails. This is satisfactory to implement quorum based protocols.

In the case where the number of responses required is equal to the number of members of the group, the primitive is also called *reliable* multicast. Reliable multicast is used as the base of two other qualities of service: *causal* and *delta*.

6.3. The Atomic Quality of Service

The *atomic* quality of service, in relation to the other qualities of service previously described, introduces the assurance of total order. This can be achieved exploiting the properties of the abstract network: in fact messages are naturally ordered as they cross the LAN medium (abstract network property Pn5). To preserve network order, a mechanism must be implemented to ensure that the messages are delivered to the user respecting the order they have crossed the network and, when a message crosses the network several times, that a unique re-transmission is used to establish this order. This requires extra work both at the sender and at the recipient sides, as described below.

In each recipient, is maintained a *reception queue*, where messages are inserted by the order they cross the network. Since at the moment of reception, a recipient as no way to know if the message was also received by the other recipients, the message cannot be delivered immediately to the user. Instead, it is stamped as *unaccepted* and kept in the queue until there is an assurance that it was inserted in the same relative position in *all* recipient's queues. If meanwhile, a re-transmission is received, the message is moved to the end of the queue. On its side, the sender invokes *tr-w-resp* activating the "ord" flag, thus requiring the re-transmission of the message until all recipients acknowledge the same retry. When a successful re-transmission is detected, the sender issues an *accept* frame, committing the message. When the accept frame is received, the recipients mark the associated message as *accepted* and deliver it as soon as it reaches the top of the queue.

If a receiver is not able to process the message[†] due to lack of resources like buffer space or scheduling guarantees it notifies the sender by returning a *not-ok acknowledgement* to the sender. The sender reacts on the receipt of such a negative acknowledgement by issuing a *reject* instead of an *accept* message. Upon receipt of the reject all recipients discard the corresponding data message.

The atomic service consists of a **two-phase accept** protocol (see Figure 3) that resembles a commit protocol where the *sender* coordinates the protocol: In the dissemination phase the data message is sent to all recipients, who have to respond if they will be able to process the message. In the second phase (decision phase) the sender decides to send either an *accept* or a *reject* message. To increase performance the accept message is sent using a *negative* acknowledgement scheme: If a recipient has not received a decision message due to an omission, it will detect this through a timeout mechanism and send a *Request-Decision* frame. Using this scheme a second round of acknowledgements is avoided increasing the performance. In the scenario where there is an omission of an *accept* message, termination of the protocol

[†] More related work on *inaccessibility* can be found in [Ruf92a, Ruf92b].

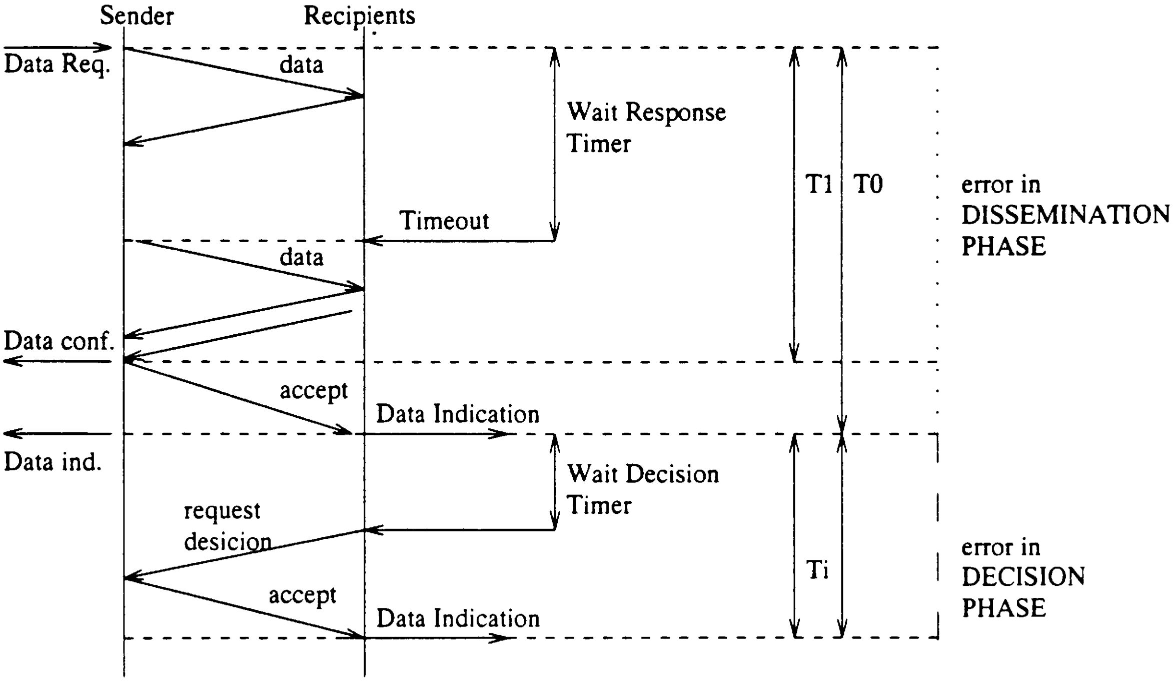


Figure 3: *Structure of the multi-phase Atomic Multicast protocol*

is delayed but due to the low error rate expected in local area networks, throughput is significantly improved.

Since in the two-phase accept the sender coordinates the protocol, some exception mechanism must be implemented to overcome its failure. In the *atomic* quality of service, protocol execution is carried on, in the event of sender failure, by a termination protocol. This termination protocol is executed by an *atomic monitor* function. There is no permanent monitor activity however – so to speak, a monitor only exists when needed. The monitor impersonates the failed sender but never re-transmits a data message on its behalf. It just collects information about the state of the transmission and disseminates an decision (reject or accept) accordingly [Ver90b].

7. Clock Synchronization

A number of classes of distributed applications require access to a global time base, for implementation of coordinated decentralized actions in the time domain, sensoring, performance measurement or timestamping of events.

It is possible to provide such a timebase by using a centralized time service, resident in a single node of the system. This solution is not fault-tolerant, exhibits poor performance if clocks need to be frequently read, and errors are introduced due to variation of transmission delays. The common solution for the clock synchronization problem lies on using the processor hardware clock to create a virtual clock at each node, which is locally read. All virtual clocks are synchronized by a *clock synchronization algorithm*. Surveys of existing clock synchronization algorithms can be found in [Sch87a, Ram90a, Kop89a]. Of the available software algorithms the convergence-non-averaging algorithms are attractive because they use the convergence function both to generate the re-synchronization event and to adjust virtual clocks.

However, the existing algorithms of this class have a major disadvantage: the precision of their convergence function is limited by the maximum message transmission delay in the system.

Verissimo and Rodrigues [Ver92b] have developed an algorithm which overcomes the limitation caused by the uncertain message delays, by using the properties of broadcast networks. The algorithm is implemented within the Group Communication Service using dedicated services available to the *x*AMp protocol suite [Rod91b].

The global time is available to the user through library functions that read the virtual clock value.

8. Processor Management

To provide efficient management of stations, a low-level processor membership protocol [Rod92b] is developed that deals with availability information on the nodes in the network. This information is not static: during the lifetime of the system, stations will join, leave and, possibly, fail. The protocol runs directly on top of the LAN (Abstract Network) to achieve improved performance and to offer a service that can be used by other protocol layers.

The processor membership has two major goals:

- It keeps a complete, and updated, list of a selected group of stations, participating in the multicast traffic (target systems typically include up to 32 nodes). This group is called the *Multicast Group of Stations* or simply MGS. The MGS protocol assures that the membership view is updated consistently in the presence of joins, leaves and failures. Changes in the MGS membership are indicated to the protocol users.
- It implements a mapping function that translates unique node identifiers into short-addresses. To enable run-time reconfiguration, the mapping is not statically pre-defined and new stations are able to, at any time, obtain a short-address. This mapping is *universal* and *stable*, meaning that, in all stations, the same short-address corresponds to the same station and that correspondence remains unchanged during the lifetime of the system.

The use of short addresses, as also exploited in Autonet [Sch90a], is to provide fast address manipulation based on bitmasking. These operations provide a significant performance improvement and, when the maximum number of multicast stations is small, allows the recognition of selective addresses to be implemented in hardware, by the chipset of the underlying network.[†]

8.1. Protocol Service

Our group membership protocol provides the mapping function referred above by maintaining a table with information about all stations participating in the multicast traffic. For efficiency and fault-tolerance, the table is replicated at every group member. The table includes an array of state entries, each entry storing information about a given member of the group: an entry contains, at least, the node unique identifier and a boolean stating if the node is alive. Additionally, the

[†] For instance, the MC68824 token-bus controuer has a *group address mask* which can be set to filter messages in function of a bit value.

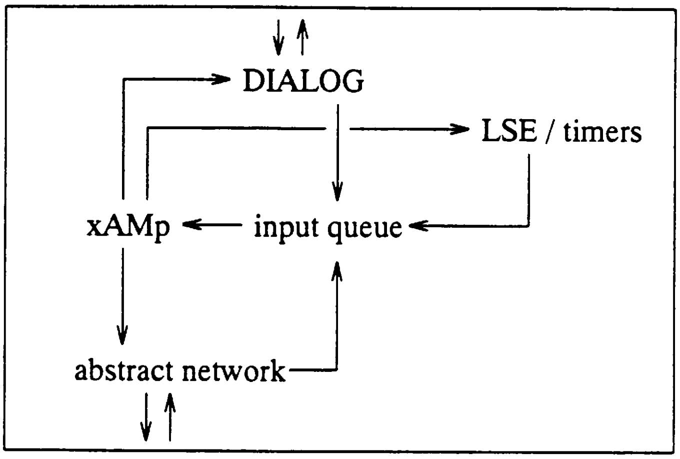


Figure 4: *Interaction between several modules*

entry may store user related data. The short-address associated with each MGS member is stored implicitly: it corresponds to the index of the associated entry in the table.

A station may be connected to the network without participating in the group membership protocol. In order to join the MGS group it must execute a *MgsJoin* operation. The join operation requires exchange of messages with the other MGS members to acquire the state table, insert itself and obtain a short-address. Upon an insertion in the MGS, a station is informed of any change in the MGS membership by an *MgsChange* indication. A station may leave the MGS by executing an *MgsLeave* operation. The MGS membership is checked at every execution of a Join or Leave or when a specific *MgsCheck* operation is explicitly invoked. The *MgsCheck* can be called periodically or upon the detection of an event that raises suspicion about the failure of a MGS member.

When a station joins the MGS, it acquires a short-address which will remain associated with that station. Even if the station fails or leaves the MGS group, the short-address remains assigned to the station, such that the remaining stations can refer to it by the associated short-address. If the station recovers and executes a new join, it obtains its old short address. A dedicated operation, *MgsDelete* is used to remove a station from the MGS table and to release the associated short-address. Since there is a local copy of the MGS table available at every station, translation between unique identifiers and short-address is a purely local operation.

Once the MGS protocol has inserted the station into the group it makes use of the *x*AMp primitives to assure the detection of failed stations. The MGS protocol joins an *x*AMp group that includes all available stations, within this group *keep-alive* message are sent to trigger the *Group Monitor* in case of failure of a station. The Group Monitor will automatically call the MGS protocol primitives to assure a consistent view of the available stations.

9. Dialog

One of the goals in the design of the UNIX kernel version of the Group Support Service was that the service should be available through the standard UNIX network interfaces like *streams* and *sockets*. Although the structure of both BSD and System V style network protocols didn't match the structure of the *x*AMp protocol core we wanted to make an effort of offering the service through these interfaces. The interface between the *x*AMp and the socket and streams endpoint environment is named **Dialog**.

The *x*AMp is structured as a state machine with an input channel on which messages from user, network and synchronous timers arrive and a collection of routines that are called as result of changes in the state machine, resulting in confirmation and indication messages to the user, interaction with the abstract network, including sending of messages, and manipulation of timers (see Figure 4).

The *x*AMp runs as *light weight process* (thread) in the kernel, sleeping on the input queue channel. If a message is placed in the input queue a wakeup of the *x*AMp thread is generated. If the *x*AMp outputs messages to the user it calls a routine from a predefined collection of *dialog* routines. These routines handle the two types of user messages generated by the *x*AMp:

- **Confirmation**: The requested operation has completed. A confirmation can be positive or negative regarding the result of the operation.
- **Indication**: data output is produced for the user, this could be a new group or processor view, a time synchronization message or data received for a group member.

The user can specify through control operations which types of confirmations and indications he does want to receive.

Confirmations are necessary as the caller is not blocked in the submitting routine until the operation is successful. For the socket version this requires some additional mechanisms in the dialog module to maintain the blocking semantics of the *send* and *write* system calls.

The Dialog module for the *streams* version was expected to link up better with the *x*AMp, as *streams* are also model after a submit/confirmation/indication model. Already in the prototype phase it became noticeable that using streams in SunOS is a very expensive method of designing network software, it added almost a 80 msec overhead for interaction with the abstract network driver. This scale of delay was unacceptable for our goals, and we stopped with the development of the streams driver.

As an alternative to the streams environment we built a *Dialog* interface to a regular UNIX device driver that has the same user semantics as a streams driver. This implementation turned out to have good performance with low overhead, as all message handling is tuned for this specific environment.

The driver, as well as the socket code, supports all UNIX type operations like select, signals, non-blocking read, etc.

10. Formal Specification and Verification

There is now a general agreement that protocols must be validated. We have chosen to do a formal design specification (as opposed to simulation) because this will give you insight in possible errors in your protocol design. As an approach to formal verification we decided to use model checking instead of deductive proof methods for the same reason: it is of great help for the detection of errors. In order to apply these techniques, one needs the description of a complete system consisting of a fixed number of communicating entities and their interaction environment. Such a complete system is called a *scenario* Practically, validation comes down to the construction of a certain number of critical scenarios and their formal verification by using a tool.

For the verification of AMp we have used the verification tool Xesar [Ric87a, Gra89a]. This tool evaluates properties given by formulas of temporal logic on a *model* generated from a scenario to be verified. The *model* represents the complete state graph obtained automatically from a scenario written in Estelle/R, a variant of Estelle (communication is modeled by *rendez-vous*). The basis for the verification is the complete Estelle/R design specification of the AMp. Since a *closed system* (for each message, the sender and the receiver must be described) is needed for the verification, a description of the environment is also needed, i.e. the modeling of the adjacent protocol layers: the network layer and the user layer.

The work on the verification of the AMp has been very successful, a number of possible errors has been found, and the results of the verification have given us great confidence in the correctness of the protocol [Bap90a].

The implementation was also subject to a validation effort: a fault injection campaign is in course, with the aim of forecasting faults and assisting in its removal, with the help of a specialized tool [Arl90a].

11. Performance Measurement

Throughout this paper we have stated that achieving a responsive service was one of the main goals to achieve. In this section we will describe some of the performance measurements we have executed. The next section will focus on comparing these results with those of comparable systems.

The measurements have been performed in two different operating system environments:

- **SunOS 4.1.1** – running on SPARCstations I, IPC's and SLC's
- **Mach2.5** – running on 33Mhz i486 machines of Taiwanese origin.

The environments differ most in the implementation of the *Abstract Network*. We did not have the source code for the SunOS operating system available and used the *ether_family* mechanism to insert are protocols in the de-multiplexing process. For the Mach2.5 port we were able to implement the Abstract Network exactly as we designed, having access to all functionality of the lowest layers.

Measurements were done by using the special performance device driver, which allows us to make timestamps at different stages of the frame manipulation process, collecting these timestamps afterwards.

The main application of the Group Communication Service is in the area of responsive and real-time systems. In this context we are more interested in the timely execution of the primitives and in the exceptions in the execution times. We recognize the importance of throughput of large batches of messages, but the protocols are tuned towards single message handling and guaranteed timely termination of the protocols.

The first set of measurements are to determine the latency caused by the Abstract Network and the physical transport over the network. Relevant is the size of the buffer used. The number of stations used is not

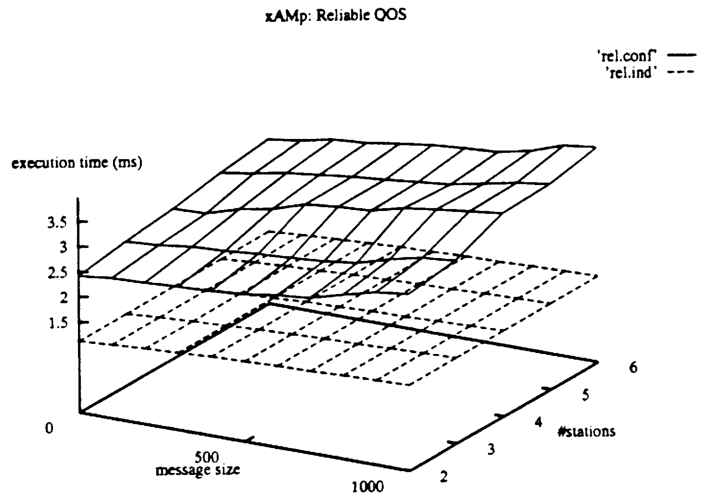


Figure 5: *Performance of the reliable quality of service*

of any influence as all message are transmitted using hardware multicast.

When selecting only the i486 machines the latency dropped significantly with 30 to 35%, this is caused by the more optimal abstract network implementation.

The variance of the large buffer transfers is larger because of the collisions on the network. When we repeated the tests on an isolated Ethernet and the variance approaches that of the smaller buffers.

The second series of tests involved measurements of the *bestEffort, atLeast* and *reliable* primitives. As these primitives involve exchanges of acknowledgements the number of stations plays a role in the performance of the protocols. Two points of measurement are taken:

1. The moment the data is indicated to the user.
2. The moment the sender is confirmed of the termination of the protocol.

All three primitives are not concerned with ordering properties and indicate the data as soon at it arrives at the Group Communication Service. The protocols terminate after the requested number of the specified subset of group members have acknowledged the message (see Figure 5).

Abstract network roundtrip time (msec)		
frame size (bytes)	i486 & SPARC stations	i486 stations
1	1.04	0.78
100	1.13	0.82
200	1.27	0.87
500	1.32	0.93
1000	1.39	1.01
1450	1.48	1.12

Table 3: *Abstract network latency*

Group membership primitives						
# stations	0	1	2	3	4	5
join	2.5	3.2	4.1	4.9	5.6	6.8
leave	1.2	2.0	2.4	2.6	2.9	3.2

Table 4: *group join performance*

In case of the *causal* ordered primitive, there is a small overhead for handling of the ordering protocol which resides on top of the reliable primitive. But we have noticed that in the case when all related messages already have been received the overhead is in the order of 170 microseconds.

The *atomic* primitive is a **two-phase accept** protocol (see Figure 3) that confirms the user about the result of the operating after the *dissemination* phase and indicates the data after the *decision* phase (see Figure 6).

As the last results we want to report on the performance of the group membership primitives. In Table 4 the costs of joining and leaving a group is presented.

12. Performance Comparison

From all published research in the area of group communication we will discuss our results in comparison with the results of ISIS, Amoeba, and Consul/Psync. We have chosen these three systems because they all represent a different main stream in group communication.

The *ISIS* [Bir91b] toolkit offers a versatile set of communication primitives combined with higher level implementations of distributed algorithms. The ISIS protocols run in user space using the standard communication channels. The authors have put the emphasis on throughput sacrificing some of the responsive behavior. As the toolkit relies on the

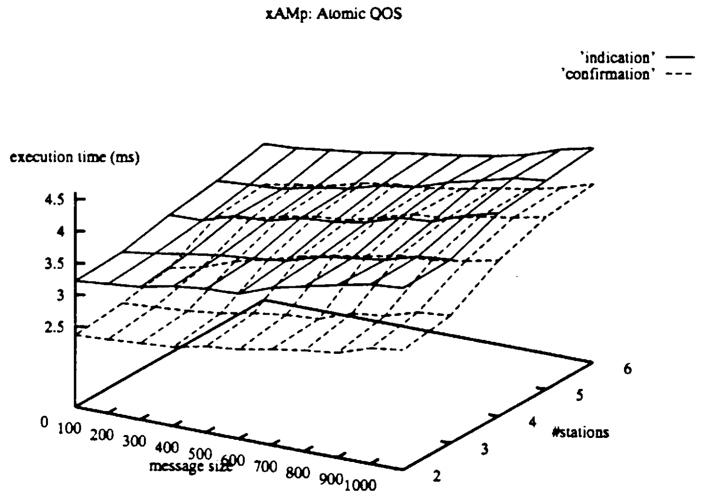


Figure 6: *Performance of the atomic quality of service*

standard Internet protocols to transport their messages they are not bound by the scope of a Local Area Network, but do have to deal with the sometimes unpredictable behavior of the UDP/TCP/IP layers. Obviously these transport methods make the toolkit very portable but a significant performance penalty is paid to achieve this.

The ISIS CBCAST primitive is comparable with the reliable/causal primitive offered by the *x*AMp protocol suite. The ABCAST can be compared with the atomic two phase accept protocol described earlier, although ABCAST is not able to distinguish *inaccessibility* from communication or processor failure. Although the complexity of the protocols is comparable the performance of the *x*AMp primitives is much better, a 0 bytes CBCAST (6 stations) takes about 17.8 msec, while the *x*AMp causal primitive takes 2.96 msec (confirmation). For 1K packet the costs are 21.1 and 3.9 msec respectively.

The performance analyses given in [Bir91c] show that more than 75% of the measured latency in the ISIS system is caused by the operating system layers. Our Group Communication Service gains in performance by locating its service as close to the network as possible, bypassing as much system layers as possible. Another reason for the improved performance is the use of hardware multicast by the abstract network, minimizing the message traffic. At Cornell a total redesign of ISIS is in progress which will result in a system that will approach the perform of the *x*AMp primitives.

In *Amoeba* [Tan90a, Ren88a] the group communication service offers only one primitive: total order within a group [Kaa89a]. The protocol uses sequencer sites to regulate the order of the messages. The protocols simpleness results in very high performance but lacks the ability to be used in more complex environments with different application requirements. Some of the major criticisms are the inability of the protocol to support overlapping groups and the lack of timely omission detection. Recently the protocol has been adjusted to make use of the Fast Local Internet Protocol (FLIP) [Kaa92a] which provides reliable multicasting. Ongoing research at the Vrije Universiteit is focused on extending the current protocols.

When comparing performance figures it is clear that the complexness of our service makes it not competitive with the less versatile service of Amoeba. On a lightly loaded Ethernet the Amoeba protocol makes an atomic broadcast to 10 station within 1.5 msec. There are no acknowledgements involved, and there is little overhead from the network interface modules.

Consul [Mis92a] is a communication substrate for building fault-tolerant systems, the system relies heavily on the services offered by Psync, a group communication protocol designed to preserve causality based on the use of a context graph. The systems are build within the *x*-kernel [Hut90a], a protocol development environment from the University of Arizona. The performance of this dedicated environment is almost comparable to our implementation, both causal and total order service have a latency that is 0.5 to 1.5 msec higher than the *x*AMp protocols. We believe the slightly worse performance of the total order service can be related to the fact that this service is build on top of the causal order primitive.

13. Lessons Learned

Some conclusion from the practical side are:

- Having multiple qualities of service helps the builder of distributed applications to minimize communication cost.
- Exploiting network properties makes building of group communication easier and more responsive.
- Integration inside the operating system has yielded good performing service.
- Integration into a operating system without having access to the source code should be avoided.
- Implementing responsive protocols using SunOS streams is not possible.
- Using formal verification techniques has improved the confidence in the correctness of the protocols.
- The different addressing modes have improved the usefulness of the service.
- The short addresses have improved the address manipulation enormously but do not scale well.
- Portability is possible even within between operating system code if at design time an effort has been made to locate system dependencies.

The experiences with the design of the group support service form the basis of a report on requirements for building group support systems, see [Vog92a].

14. Future Directions

Our current research continues to focus on integration of the group concept in different areas of distributed computing. Our main goal is to achieve a high performance group service that can be used in real-time and responsive systems. We will also focus on how to build responsive group support for large scale distributed systems, especially in the area of CSCW. Another main line is the development of group management protocols [Ver92a].

In the *Navigators* project we are focusing on a total redesign of the *x*AMp protocol suite to to incorporate new ideas on responsive systems, dedicated support by micro-kernels, low-level high-performance transport mechanisms, multi-level failure detectors, etc. In the same environment we try to incorporate internetworking support for group communication at MAN and WAN scale [Vog91b].

Our new environment is being developed for the Mach 3.0 microkernel and a prototype is planned for the end of 1992. Also cooperation between newly designed ISIS modules and Navigators protocols are foreseen.

Formal verification and specification techniques will be more integrated into the design process as they have shown in our case to improve the quality of the protocols built.

15. Summary

In this paper we have presented the implementation of a Group Communication Service aimed at achieving high-performance to support distributed applications that have responsive requirements. Scalability has been traded for timely protocol execution of the protocols.

The main concepts are described as well as the actual implementation, and design decisions have been motivated. For more details on specific parts of the service the reader is referred to [Ver90b, Rod92a].

We have described the performance of our protocols and compared these to three other popular group communication services. When looking at the performance figures is becomes clear that the different protocols that form the core of our service can compete with any other know group communication system in both performance and quality of the offered services.

Acknowledgements

Most of this work has been developed in the scope of the Delta-4 architecture. We wish to thank M. Baptista, A. Casimiro, M. Chereque, J. Etheringhton, H. Fonseca, B. Guilmore, R. Ribot, C. Rodriguez, and J. Rufino for their valuable contribution to the evolution and engineering of *x*AMp. The authors are indebted to several other people in the Delta-4 project for the many criticisms and suggestions made during the design and implementation of *x*AMp, particularly to P. Bond, D. Powell, JL. Richier, D. Seaton and J. Voiron. We also thank the fruitful exchanges of ideas with Ken Birman, Larry Peterson and Rick Schlichting.

References

[Arl90a] J. Arlat, M. Aguera, Y. Crouzet, J. Fabre, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: a Methodology and some Applications," *IEEE Transactions on Software Engineering*, IEEE (February 1990). Special Issue of Experimental C.Sc.

[Bap90a] M. Baptista, L. Rodrigues, P. Veríssimo, S. Graf, J. L. Richier, C. Rodriguez, and J. Voiron, "Formal Specification and Verification of a Network Independent Atomic Multicast Protocol," *Third International Conference on Formal Description Techniques (FORTE 90)*, Madrid, Spain, IFIP (November 1990).

[Ber89a] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "Lightweight Remote Procedure Call," *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 102-113, ACM (1989).

[Bir91c] Kenneth Birman, Andre Schiper, and Pat Stephenson, "Lightweight causal and Atomic Group Multicast," *ACM Transactions on Computer Systems* **9**(3) (August 1991).

[Bir91a] Kenneth P. Birman, "The Process Group Approach to Reliable Distributed Computing.," TR 91-1216, Cornell University, Ithaca, USA (July 1991).

[Bir91b] Kenneth P. Birman, R. Cooper, and B. Gleeson, "Design Alternatives for Process Group Membership and Multicast," TR91-1185, Cornell University, Ithaca, USA (december 1991).

[Bir84a] Andrew D. Birrell and Bruce Jay Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* 2(1) (February 1984).

[Che85a] D. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V-Kernel," *ACM Transactions on Computer Systems* 3(2) (May 1985).

[Che88a] Greg Chesson, "XTP/PE Overview," *13th Local Computer Network Conference*, Minneapolis-USA (October 1988).

[Coo85a] Eric C. Cooper, "Replicated Distributed Programs," *10th ACM Symposium on Operating Systems Principles*, Berkeley, California 94720, USA, ACM (November 1985).

[Cri90a] Flaviu Cristian, Robert D. Dancey, and Jon Dehn, "Fault-Tolerance in the Advanced Automation System," *Digest of Papers, The 20th International Symposium on Fault-Tolerant Computing*, Newcastle-UK, IEEE (June 1990).

[Dru92a] Peter Druschel and Larry L. Peterson, "High-Performance Cross-Domain Data Transfer," TR 92-11, University of Arizona, Tucson, USA (March 1992).

[Fon90a] H. Fonseca, L. Rodrigues, J. Rufino, and P. Veríssimo, "Local Support Environment: User Specification," RT/50-90, INESC, Lisboa, Portugal (August 1990).

[Gar89a] H. Garcia-Molina and Annemarie Spauste, "Message Ordering in a Multicast Environment," *9th Internacional Conference on Distributed Computing Systems*, pp. 354-361, IEEE (June 1989).

[Gra89a] S. Graf, J. L. Richier, C. Rodriguez, and J. Voiron, "What are the Limits of Model Checking Methods for the Verification of Real Life Protocols?," pp. 275-285 in *Automatic Verification Methods for Finite State Systems*, ed. J. Sifakis, Springer-Verlag (June 1989).

[Her89a] A. J. Herbert, J. Monk, and R. van der Linden, *The ANSA Reference Manual,* Architecture Projects Management, Ltd, Cambridge, UK (July 1989).

[Hut89a] Norman C. Hutchinson, Shivakant Mishra, LArry L. Peterson, and Vicraj T. Thomas, "Tools for Implementing Network Protocols," *Software – Practice and Experience* (September 1989).

[Hut90a] Norman C. Hutchinson and Larry L. Peterson, *The x-Kernel: An Architecture for Implementing Network Protocols,* University of Arizona, Tucson, USA (1990).

[ISO85a] ISO, "Token Passing Bus Access Method," DIS 8802/4-85 (1985).

[ISO85b] ISO, "Token Ring Access Method," DP 8802/5-85 (1985).

[ISO85c] ISO, "Carrier Sense Multiple Access with Collision Detection," DIS 8802/3-85, ISO (1985).

[ISO85d] ISO, "Logical Link Control," DIS 8802/2-85, ISO (1985).

[Kaa89a] Frans M. Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal, "An Efficient Reliable Broadcast Protocol," *ACM Operatings Systems Review*, pp. 5-19 (October 1989).

[Kaa92a] Frans M. Kaashoek, Robert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum, "FLIP: an Internetwork Protocol for Supporting Distributed Systems," *ACM Transactions on Computer Systems* (1992).

[Kop89a] H. Kopetz, G. Grunsteidl, and J. Reisinger, "Fault-tolerant Membership Service in a Synchronous Distributed Realtime System," *Int. Working Conference on Dependable Computing for Critical Applications*, Sta Barbara – USA, IFIP WG10.4 (August 1989).

[Mis92a] Shivakant Mishra, Larry L. Peterson, and Richard Schlichting, *Consul: A Communication Substrate for Fault-Tolerant Distributed Programs,* University of Arizona, Tucson, USA (1992).

[Pet89a] Larry L. Peterson, Nick C. Buchholdz, and Richard D. Schlichting, "Preserving and Using Context Information in Interprocess Communication," *ACM Transactions on Computer Systems* 7(3) (August 1989).

[Pow91a] D. Powell, *Delta-4 – A Generic Architecture for Dependable Distributed Computing,* Springer Verlag (November 1991).

[Ram90a] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler, "Fault-Tolerant Clock Synchronization in Distributed Systems," *Computer*, pp. 33-42, IEEE (October 1990).

[Ren88a] Robert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum, "The Performance of the Worlds's Fastest Distributed Operating System," *ACM Operatings Systems Review*, pp. 25-34 (October 1988).

[Ric87a] J. L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron, *XESAR: a Tool for Protocol Validation – User Manual,* Laboratoire de Genie Informatique, Grenoble, France (1987).

[Rod91b] L. Rodrigues, P. Veríssimo, and A. Casimiro, "xAMp Time Service Implementation Specification," RT/-91, Delta-4 Project, INESC, Lisboa, Portugal (October 1991).

[Rod92a] L. Rodrigues and P. Veríssimo, "*x*AMp: a Multi-primitive Group Communications Service," *11th Symposium on Reliable Distributed Systems*, Houston, Texas, IEEE (October 1992).

[Rod92b] L. Rodrigues, P. Veríssimo, and J. Rufino, "A low-level processor group membership protocol for LANS," RT/-92, INESC, Lisboa, Portugal (1992).

[Rod91a] Luís Rodrigues and Paulo Veríssimo, "*x*AMp: A Versatile Group Communications Service," *ERCIM Workshop on Distributed Systems*, Lisboa, Portugal (November 1991).

[Ruf92b] Jose Rufino and Paulo Veríssimo, "Minimizing token-bus inaccessibility through network planning and parameterizing," *EFOC/LAN92 Conference*, Paris, France, IGI (June 1992).

[Ruf91a] J. Rufino, P. Veríssimo, and L. Rodrigues, "Abstract Network Specification," RT/-91, INFSC, Lisboa, Portugal (October 1991).

[Ruf91b] J. Rufino and P. Verı́ssimo., "Design Requirements of the Abstract Network User Interface," RT/-91, INESC, Lisboa, Portugal (January 1991).

[Ruf92a] J. Rufino and P. Verı́ssimo, "A study on the inaccessibility characteristics of ISO 8802/4 Token-Bus LANs," *IEEE INFOCOM'92 Conference on Computer Communications*, Florence, Italy, IEEE (May 1992).

[Sch87a] Fred B. Schneider, *Understanding Protocols for Byzantine Clock Synchronization,* Cornell University, Ithaca, New York (October 1987).

[Sch89a] Michael D. Schroeder and Micheal Burrows, "Performance of Firefly RPC," *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 83-90, ACM (1989).

[Sch90a] Michael D. Schroeder, Andrew D. Birrell, Micheal Burrows, Edwin H. Satterthwaite, and Charles P. Thacker, "Autonet: a High-Speed, Self-configuring, Local Area Network Using Point-to-point Links," 59, Digital, Systems Research Center, Palo Alto, California (April 1990).

[Tan90a] Andrew S. Tanenbaum, Robert van Renesse, Hans van Staveren, G. J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum, "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM* (December 1990).

[Ver90b] Paulo Verı́ssimo, "Group Communications Support," in *Delta-4 A Generic Architecture for Dependable Distributed Computing*, ed. D. Powell, Springer Verlag (1990).

[Ver89a] P. Verı́ssimo and L. Rodrigues, "Order and Synchronism Properties of Reliable Broadcast Protocols," RT/66-89, INESC, Lisboa, Portugal (December 1989).

[Ver90a] P. Verı́ssimo and J. A. Marques, "Reliable Broadcast for Fault-Tolerance on Local Computer Networks," *Ninth Symposium on Reliable Distributed Systems*, Huntsville, Alabama, USA, IEEE (Oct 1990).

[Ver91a] P. Verı́ssimo, J. Rufino, and L. Rodrigues, "Enforcing Real-Time behaviour of LAN-based protocols," *10th IFAC Workshop on Distributed Computer Control Systems*, Semmering, Austria, IFAC (September 1991).

[Ver92a] P. Verı́ssimo and L. Rodrigues, "Group Orientation: a Paradigm for Distributed Systems of the Nineties," *3rd Workshop on Future Trends of Distributed Computing Systems*, Taipe, Taiwan (April 1992).

[Ver92b] P. Verı́ssimo and L. Rodrigues, "A posteriori Agreement for Fault-Tolerant Clock Synchronization on Broadcast Networks," in *Digest of Papers, The 22th International Symposium on Fault-Tolerant Computing* (July 1992).

[Vog91a] Werner Vogels and Paulo Verı́ssimo, "Process Groups and reliable multicast communication in extended LANs, MANs and Internetworks," RT/-91, INESC, Lisboa, Portugal (August 1991).

[Vog91b] Werner Vogels and Paulo Verı́ssimo, "Supporting Process Groups in Internetworks with Lightweight Reliable Multicast Protocols," *ERCIM Workshop on Distributed Systems*, Lisboa, Portugal (November 1991).

[Vog92a] Werner Vogels, Luís Rodrigues, and Paulo Veríssimo, "Requirments for High-Performance Group Support.," *5th ACM SIGOPS European workshop*, Mont Saint-Michel, ACM (September 1992).

[X3T86a] X3T9.5, "FDDI documents: Media Access Layer, Physical and Medium Dependent Layer, Station Mgt.," FDDI, X3T9.5 (1986).

Group Communication in Amoeba and its Applications

M. Frans Kaashoek
Andrew S. Tanenbaum Kees Verstoep
Vrije Universiteit
Amsterdam, The Netherlands
kaashoek@cs.vu.nl

Abstract

Unlike many other operating systems, Amoeba is a distributed operating system that provides group communication (i.e., one-to-many communication). We will discuss design issues for group communication, Amoeba's group system calls, and the protocols to implement group communication. To demonstrate that group communication is an useful abstraction, we will describe a design and implementation of a fault-tolerant directory service. We discuss two versions of the directory service: one with Non-Volatile RAM (NVRAM) and one without NVRAM. We will give performance figures for both implementations.

1. Introduction

Most current distributed operating systems provide only *Remote Procedure Call* (RPC) [Bir84a]. The idea is to hide the message passing, and make the communication look like an ordinary procedure call (see Figure 1). The sender, called the *client*, calls a *stub routine* on its own machine that builds a message containing the name of the procedure to be called and all the parameters. It then passes this message to the driver for transmission over the network. When it arrives, the remote driver gives it to a stub, which unpacks the message and makes an ordinary procedure call to the *server*. The reply from server to client follows the reverse path.

Although RPC is a very useful communication paradigm, many applications need something more. RPC is inherently point-to-point communication, but what often is needed is 1-to-*n* communication. Consider, for example, a parallel application. Typically in a parallel application a

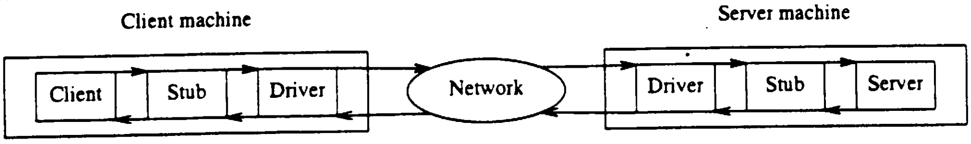


Figure 1: *Remote procedure call from a client to a server*

number of processes cooperate to compute a single result. If one of the processes finds a partial result (e.g., a better bound in a parallel branch-and-bound program) it is often necessary that this partial result is communicated immediately to the other processes, so that they do not waste cycles on computing something that is not interesting anymore, given the new partial result. What is needed here is a way to send a message from 1 process to n processes. This abstraction is called *group communication*.

Now consider a second application: a fault-tolerant storage service. A reliable storage service can be built by replicating data on multiple processors each with their own disk. If a piece of data needs to be changed, the service either has to send the new data to all processes or invalidate all other copies of the changed data. If only point-to-point communication were available, then the process would have to send $n - 1$ reliable point-to-point messages. In most systems this will cost at least $2(n - 1)$ messages (one packet for the actual message and one packet for the acknowledgement). If the message sent by the server has to be fragmented into multiple network packets, then the cost will be even higher. This method is slow, inefficient, and wasteful of network bandwidth.

In addition to being expensive, building distributed applications using only point-to-point communication is often difficult. If, for example, two servers in the reliable storage service receive a request to update the same data, they need a way to order the updates, otherwise the data may become inconsistent. The problem is illustrated in Figure 2. The copies of variable x become inconsistent because the messages from Server 1 and Server 2 are not ordered. What is needed is that all point-to-point messages sent by one server precede all point-to-point messages sent by the other server.

Many network designers have realized that group communication is an important tool for building distributed applications; broadcast communication is provided by many networks, including LANs, geosyn-

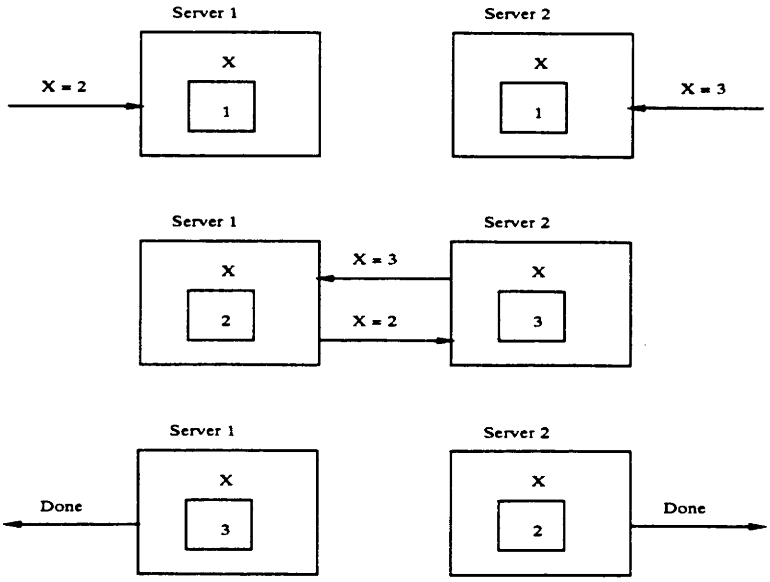


Figure 2: *Inconsistency due to lack of message ordering*

chronous satellites, and cellular radio systems [Tan89a]. Several commonly used LANs, such as Ethernet and some rings, even provide multicast communication. Using multicast communication, messages can be sent exactly to the group of processes that are interested in receiving them. Future networks, like Gigabit LANs, are also likely to implement broadcasting and/or multicasting to support high-performance applications such as multimedia [Kun92a].

The protocol presented in this paper for group communication uses the hardware multicast capability of a network, if one exists. Otherwise, it uses broadcast messages or point-to-point messages, depending on the size of the group and the availability of broadcast communication. Thus, Amoeba makes the hardware support for group communication available to application programs.

The outline of the rest of the paper is as follows. In Section 2, we will discuss design issues in group communication. In Section 3, we will discuss the Amoeba group system calls. In Section 4, we will give an overview of the protocols that implement group communication. In Section 5, we will describe the design and implementation of a distributed application using group communication: a fault-tolerant directory service. In Section 6, we will give performance figures for two implementations of the directory service. In Section 7, we will draw some conclusions.

2. Design Issues in Group Communication

Few existing operating systems provide application programs with support for group communication. To understand the differences between these existing systems, six design criteria are of interest: addressing, reliability, ordering, delivery semantics, response semantics, and group structure (see Figure 3). We will discuss each one in turn.

Four methods of *addressing* messages to a group exist. The simplest one is to require the sender to explicitly specify all the destinations to which the message should be delivered. A second method is to use a single address for the whole group. This method saves bandwidth and also allows a process to send a message without knowing which processes are members of the group. Two less common addressing methods are *source addressing* [Gue85a], and *functional addressing* [Hug88a]. Using source addressing, a process accepts a message if the source is a member of the group. Using functional addressing, a process accepts a message if a user-defined function on the message evaluates to true. The disadvantage of the latter two methods is that they are hard to implement with current network interfaces.

Issue	Description
Addressing	Addressing method for a group (e.g., list of members)
Reliability	Reliable or unreliable communication
Ordering	Order among messages (e.g., global ordering)
Delivery semantics	How many processes must receive the message successfully
Response semantics	How to respond to a broadcast message
Group structure	Semantics of a group (e.g., dynamic versus static)

Figure 3: *The main design issues for group communication*

The second design criterion, *reliability*, deals with recovery from communication failures, such as buffer overflows and garbled packets. Because reliability is more difficult to implement for group communication than for point-to-point communication, a number of existing operating systems provide *unreliable* group communication, whereas almost all operating systems provide *reliable* point-to-point communication, for example, in the form of RPC.

Another important design decision in group communication is the *ordering* of messages sent to a group. Roughly, there are 3 possible orderings: no ordering, causal ordering, and global ordering [Bir91a]. The first ordering is easy to understand and implement, but unfortunately makes programming harder. The causal ordering guarantees that all messages that are related are ordered. More specifically: if a member after receiving message A sends a message B, it is guaranteed that all members will receive A before B. In the global ordering, all messages are ordered. The last method is stronger than the second and makes programming easier, but is harder to implement.

To illustrate the difference between causal and global ordering, consider a service that stores records for client processes. Furthermore, assume that the service replicates the records on each server to increase availability and reliability and that it guarantees that all replicas are consistent. If a client may only update its own records, then it is sufficient if all messages from the same client will be ordered. Thus, in this case a causal ordering can be used. If a client, however, may update any of the records, then a global ordering on the updates is needed to ensure consistency among the replicas. To see this, assume that two clients, C_1 and C_2 resp., send an update for record X at the same time. As these two updates will be globally ordered, all servers either (1) receive first the update from C_1 and then the update from C_2 or (2) receive first the update from C_2 and then the update from $Csub1$. In either case, the replicas will stay consistent, because every server applies the updates in the same order. If in this case causal ordering would have been used, it might have happened that the servers applied the updates in reverse order, resulting in inconsistent replicas.

The fourth item in the table, *delivery semantics* relates to when a message is considered delivered successfully to a group. There are 3 choices: k-delivery, quorum delivery, and atomic delivery. With k-delivery, a broadcast is successful when k processes have received the message for some constant k. With quorum delivery, a broadcast is defined as being successful when a majority of the current membership has received it. With atomic delivery either all processes receive it or none do. Atomic delivery is the ideal semantics, but is harder to implement if processors can fail.

Item five, *response semantics* deals with what the sending process expects from the receiving processes [Hug89a]. There are four broad categories of what the sender can expect: no responses, a single response, many responses, and all responses. Operating systems that integrate group communication and RPC completely support all four choices [Che85a].

The last design decision specific to group communication is *group structure*. Groups can be either closed or open [Lia90a]. In a *closed* group, only members can send messages to the group. In an *open* group, nonmembers may also send messages to the group. In addition, groups can be static or dynamic. In static groups processes cannot leave or join a group, but remain a member of the group for the lifetime of the process. Dynamic groups may have a varying number of members over time.

To make these design decisions more concrete, we briefly discuss two systems that support group communication. Both systems support open dynamic groups, but differ in their semantics for reliability and ordering. In the V system [Che85a], groups are identified with a group identifier. If two processes concurrently broadcast two messages, A and B, respectively, some of the members may receive A first and others may receive B first. No guarantees about ordering are given. Reliability in the V system means that at least one of the members must have replied. A more reliable primitive can be built by waiting for a reply from all members. However, this is a very inefficient way of doing reliable broadcasting. For a completely reliable broadcast, n packets are needed (1 for the actual message and $n - 1$ for the acknowledgements).

In the Isis system [Bir87a], messages are sent to a group identifier or to a list of addresses. When sending a message, a user specifies how many replies are expected. Messages can be globally ordered. Reliability in Isis means that either *all* or *no* members of a group will receive a message, even in the face of processor failures. Because these semantics are hard to implement efficiently, Isis also provides primitives that give weaker semantics, but better performance. It is up to the programmer to decide which primitive is required.

3. Group Communication in Amoeba

Amoeba is a distributed operating system based on the client/server model [Tan90a, Mul90a]. Services in Amoeba are addressed by *ports*, which are large random numbers. When a service is started, it generates a new port and registers the port with the directory service. A client can look up the port using the directory service and ask its own kernel to send a message to the given port. The kernel will map the port on a network address. If multiple servers listen to the same port, only one (arbitrary) server will get the message.

Ports are also used to identify groups. When a group is created, a user specifies a port. Other processes can use this port, for example, to join the group or to send a message to the group. Thus, in Amoeba all entities, processes and groups, are addressed in a uniform way.

Groups in Amoeba are closed. A process that is not a member and that wishes to communicate with a group can use RPC (or it can join the group). The reason for doing so is that a client need not be aware whether a service consists of multiple servers which perhaps broadcast messages to communicate with one another, or a single server. Also, a service should not have to know whether the client consists of a single process or a group of processes. This design decision is in the spirit of the client-server paradigm: a client knows what operations are allowed, but should not know how these operations are implemented by the service.

The primitives to manage groups and to communicate within a group are given in Figure 4. We will discuss the most important one: *SendToGroup*. This primitive guarantees that *hdr* and *buf* will be delivered to all members, even in the face of unreliable communication and finite buffers. Furthermore, when the *resilience degree* of the group is r (as specified in *CreateGroup*), the protocol guarautees that even in the event of a simultaneous crash of up to r members, it will either deliver the message to all remaining members or to none. Choosing a large value for *r* provides a high degree of fault tolerance, but extracts a penalty in performance. The tradeoff chosen is up to the user.

In addition to reliability, the protocol guarantees that messages are delivered in the same order to all members. Thus, if two members (on two different machines), simultaneously broadcast two messages, *A* and *B*, the protocol guarantees that either

1. All members receive *A* first and then *B*, or
2. All members receive *B* first and then *A*.

Random mixtures, where some members get *A* first and others get *B* first, are guaranteed not to occur. Application programs can count on it.

Figure 5 lists the design issues and the choices for Amoeba. To summarize, the group primitives provide an abstraction that enables programmers to design applications consisting of one or more processes running on different machines. It is a simple, but powerful, abstraction. All members of a group see all events in the same order. Even the events of a new member joining the group, a member leaving the group, and recovery from a crashed member are globally ordered. If, for example, one process calls *JoinGroup* and a member calls *SendToGroup*, either all members first receive the join and then the broadcast or all members first receive the broadcast and then the join. In the first case the process that called *JoinGroup* will also receive the broad-

Function(parameters) → result	**Description**
`CreateGroup(port, resilience, max_group, nr_buf, max_msg) → gd`	Create a group. A process specifies how many member failures must be tolerated without loss of any message.
`JoinGroup(hdr) → gd`	Join a specified group.
`LeaveGroup(gd, hdr)`	Leave a group. The last member leaving causes the group to vanish.
`SendToGroup(gd, hdr, buf, bufsize)`	Atomically send a message to all the members of the group. All messages are globally ordered.
`ReceiveFromGroup(gd, &hdr, &buf, bufsize, &more) → size`	Block until a message arrives. *More* tells if the system has buffered any other messages.
`ResetGroup(gd, hdr, nr_members) → group_size`	Recover from processor failure. If the newly reset group has at least *nr_member* members, it succeeds.
`GetInfoGroup(gd, &state)`	Return state information about the group, such as the number of group members and the caller's member id.
`ForwardRequest(gd, member_id)`	Forward a request for the group to another group member.

Figure 4: *Primitives to manage a group and to communicate within a group*

cast message. In the second case, it will not receive the broadcast message. A mixture of these two orderings is guaranteed not to happen. This property makes reasoning about a distributed application much easier. Furthermore, the group interface gives support for building fault tolerant applications by choosing an appropriate resilience degree.

4. Implementation of Group Communication

The protocol to be described runs inside the kernel and is accessible through the primitives described in the previous section. It assumes that *unreliable* message passing between processes is possible; fragmentation, reassembly, and routing of messages are done at lower layers in the kernel [Kaa91a]. The protocol performs best on a network that supports hardware multicast. Lower layers, however, treat multicast as an optimization of sending point-to-point messages; if multicast is not available, then point-to-point communication will be used. Even if only point-to-point communication is available, the protocol is in most cases still more efficient than performing *n* RPCs. (In a mesh interconnection network, for example, the routing protocol will only use ln *n* instead of *n* messages.)

Each kernel running a group member maintains information about the group (or groups) to which the member belongs. It stores, for example, the size of the group and information about the other members in the group. Any group member can, at any instant, decide to broadcast a message to its group. It is the job of the kernel and the protocol to achieve reliable broadcasting, even in the face of unreliable communication, lost packets, finite buffers, and node failures. We assume, however, that Byzantine failures (in which a kernel sends malicious or contradictory messages) do not occur.

Without loss of generality, we assume for the remainder of this section that the system contains one group, with each member running on a separate processor. All machines run exactly the same kernel and application software. However, when the application starts up, the machine on which the group is created is made the *sequencer*. If the sequencer machine subsequently crashes, the remaining members elect a new one. The sequencer machine is in no way special – it has the same hardware and runs the same kernel as all the other machines. The only difference is that it is currently performing the sequencer function.

Issue	Choice
Addressing	Group identifier (port)
Reliability	Reliable communication; fault tolerance if specified
Ordering	Global ordering
Delivery semantics	All or none
Response semantics	None (RPC is available)
Group structure	Closed and dynamic

Figure 5: *Important design issues of Figure 3 and the choices made for Amoeba*

Basic Protocol

A brief description of the protocol is as follows (a complete description and comparison with other protocols is given in [Kaa92a]). When a group member calls *SendToGroup* to send a message, *M*, it hands the message to its kernel and is blocked. The kernel encapsulates *M* in an ordinary point-to-point message and sends it to the sequencer. When the sequencer receives *M*, it allocates the next sequence number, *s*, and broadcasts a packet containing *M* and *s*. Thus all broadcasts are issued from the same node, the sequencer. Assuming that no packets are lost, it is easy to see that if two members concurrently want to broadcast, one of them will reach the sequencer first and its message will be broadcast first. Only when that broadcast has been completed will the other broadcast be started. Thus, the sequencer provides a global time ordering. In this way, we can easily guarantee the indivisibility of broadcasting per group.

When the kernel that sent *M*, itself receives the message from the network, it knows that its broadcast has been successful. It unblocks the member that called *SendToGroup*.

Although most modern networks are highly reliable, they are not perfect, so the protocol must deal with errors. Suppose some node misses a broadcast packet, either due to a communication failure or lack of buffer space when the packet arrived. When the following broadcast packet eventually arrives, the kernel will immediately notice a gap in the sequence numbers. If it was expecting *s* next, and it receives $s + 1$ instead, it knows it has missed one.

The kernel then sends a special point-to-point message to the sequencer asking it for a copy of the missing message (or messages, if several have been missed). To be able to reply to such requests, the sequencer stores broadcast messages in the *history buffer*. The sequencer sends the missing messages to the process requesting them as point-to-point messages. The other kernels also keep a history buffer, to be able to recover from sequencer failures and to buffer messages when there is no outstanding *ReceiveFromGroup* call.

As a practical matter, a kernel has only a finite amount of space in its history buffer, so it cannot store broadcast messages indefinitely. However, if it could somehow discover that all members have received broadcasts up to and including *m*, it could then purge the first *m* broadcast messages from the history buffer.

The protocol has several ways of letting a kernel discover this information. For one thing, each point-to-point message to the sequencer (e.g., a broadcast request), contains, in a header field, the sequence number of the last broadcast received by the sender of the message (i.e., a piggybacked acknowledgement). This information is also included in the message from the sequencer to the other kernels. In this way, a kernel can maintain a table, indexed by member number, showing that member *i* has received all broadcast messages up to T_i (and perhaps more). At any instant, a kernel can compute the lowest value in this table, and safely discard all broadcast messages up to and including that value. For example, if the values of this table are 8, 7, 9, 8, 6, and 8, the kernel knows that everyone has received broadcasts 0 through 6, so they can safely be deleted from the history buffer. If a node does not do any broadcasting for a while, the sequencer will not have an up-to-date idea of which broadcasts it has received. To provide this information, nodes that have been quiet for a certain interval, Δt, send the sequencer a special packet acknowledging all received broadcasts. The sequencer can also request this information when it runs out of space in its history buffer.

PB Method and BB Method

There is a subtle design point in the protocol; there are actually two ways to do a broadcast. In the method we have just described, the sender sends a point-to-point message to the sequencer, which then broadcasts it. We call this the *PB method* (Point-to-point followed by a Broadcast). In the *BB method*, the sender broadcasts the message. When the sequencer sees the broadcast, it broadcasts a special *accept* message containing the newly assigned sequence number. A broadcast message is only "official" when the *accept* message has been sent.

These methods are logically equivalent, but they have different performance characteristics. In the PB method, each message appears on the network twice: once to the sequencer and once from the sequencer. Thus a message of length *n* bytes consumes 2*n* bytes of network bandwidth. However, only the second message is broadcast, so each user machine is interrupted only once (for the second message).

In the BB method, the full message appears only once on the network, plus a very short *accept* message from the sequencer. Thus, only about *n* bytes of bandwidth are consumed. On the other hand, every machine is interrupted twice, once for the message and once for the *accept*. Thus the PB method wastes bandwidth to reduce interrupts and the BB method minimizes bandwidth usage at the cost of more interrupts. The protocol switches dynamically between the PB method and BB method.

Processor Failures

The protocol described so far recovers from communication failures, but does not guarantee that all surviving members receive all messages that have been sent before a member crashed. For example, suppose a process sends a message to the sequencer, which broadcasts it. The sender receives the broadcast and delivers it to the application, which interacts with the external world. Now assume all other processes miss the broadcast, and the sender and sequencer both crash. Now, the effects of the message are visible but none of the other members will receive it. This is a dangerous situation that can lead to all kinds of disasters, because the "all-or-none" semantics have been violated.

To avoid this situation, *CreateGroup* has a parameter *r*, the *resilience degree* that specifies the resiliency. This means that the *SendToGroup* primitive does not return control to the application until the kernel knows that at least *r* other kernels have received the message. To achieve this, a kernel sends the message to the sequencer point-to-point (PB method) or broadcasts the message to the group (BB method). The sequencer allocates the next sequence number, but does not officially accept the message yet. Instead, it buffers the message and broadcasts the request for broadcasting to the group. On receiving such a request with a sequence number, the *r* lowest-numbered kernels buffer the message in their history and send acknowledgement messages to the sequencer. After receiving these acknowledgments, the sequencer broadcasts the *accept* message. That way, no matter which *r* machines crash, there will be at least one left containing the full history, so everyone else can be brought up to date after the recovery. Thus, an increase in fauit tolerance is paid for by a decreace in performance. The tradeoff chosen is up to the user.

5. An Application of Group Communication: a Fault-tolerant Directory Service

The group communication primitives have been used in parallel applications [Bal90a, Tan92a], and in a fault-tolerant implementation of the Orca programming language [Kaa92b]. In this section, we discuss a fault-tolerant design and implementation of Amoeba's directory service. The directory service exemplifies distributed services that provide high reliability and availability by replicating data.

The directory service is a vital service in the Amoeba distributed operating system [Ren89a]. It provides among other things a mapping from ASCII names to capabilities. In its simplest form a directory is basically a table with 2 columns: one storing the ASCII string and one storing the corresponding capability. Capabilities in Amoeba identify an object (e.g., a file). The set of capabilities a user possesses determines which objects it can access and which not. The directory service allows the users to store these capabilities under ASCII names to make life easier for them.

The previous design and implementation of the directory service is based on RPC [Ren89a]. The RPC directory service is duplicated and recovers therefore only from one processor failure. Furthermore, it cannot tolerate network partitions. We will now discuss the design and implementation of a directory service based on group communication. A comparison of the two directory services can be found in [Kaa92c].

The group directory service is triplicated (though four or more replicas are also possible, without changing the protocol) and uses active replication. Also, it allows network partitions. To keep the copies consistent, it uses a modified version of read-one write-all policy, called *accessible copies* [Abb85a]. Recovery is based on the protocol described by Skeen [Ske85a]. The main purpose of this section is to describe a fault-tolerant service based on group communication. Other projects have implemented similar services [Mar88a, Sat90a, His90a, Mis89a, Blo87a, Lis91a].

The organization of the group directory service is depicted in Figure 6. The directory service is currently built out of three directory servers, three Bullet file servers [Ren89b], and three disk servers. A Bullet server and a disk server share one disk. Each directory server stores a copy of a directory.

The directory servers form a group with a resilience degree, r, of 2. This means that if *SendToGroup* returns successfully, it is guaranteed, even if two processors fail, that the message still will be delivered to the third one. Furthermore, it is guaranteed even in the presence of communication and processor failures that each server will receive all messages in the same order. The strong semantics of *SendToGroup* make the implementation of the group directory service simple.

The service stores the administrative data on a raw disk partition of n fixed-length blocks. Block 0 contains information needed during recovery (see below). Blocks 1 to $n - 1$ contain a table of capabilities, indexed by object number. The capability in the object table points to a Bullet file that stores the directory, random number for access protection, and the sequence number of the last change.

Default Operation

Each server in the directory service consists of several threads: multiple server threads and one group thread. The server threads are waiting for requests from a clients. The group thread is waiting for an internal message sent to the group. There can be multiple server threads, but there is only one group thread. A server thread that receives a request and initiates a directory operation is called the *initiator*.

The initiator first checks if the current group has a majority (i.e., at least two of the three servers must be up). If not, the request is refused; otherwise the request is processed. The reason why even a read request requires a majority is because the network might become partitioned. Consider the following situation. Two servers and a client are on one side of the network partition and the client deletes the directory *foo*. This update will be performed, because the two servers have a majority. Now assume that the two servers crash and that the network partition is repaired. If the client asks the remaining server to list the directory *foo*, it would get the contents of a directory that it had successfully deleted earlier. Therefore, read requests are refused if the group of servers does not have a majority. (There is an escape for system administrators in case two servers lose their data forever due to, for example, a head crash.)

Read operations can be handled by any server without the need for communication between the servers. When a read request is received, the initiator checks if the kernel has any messages buffered using *GetInfoGroup*. If so, it blocks to give the group thread a chance to process the buffered messages; before performing a read operation, the initiator has to be sure that it has performed all preceding write operations. If a client, for example, deletes a directory and then tries to read it back, it has to receive an error, even if the client requests were processed at different directory servers. As messages are sent using $r = 2$, it is sufficient to see if there are any messages buffered on arrival of the

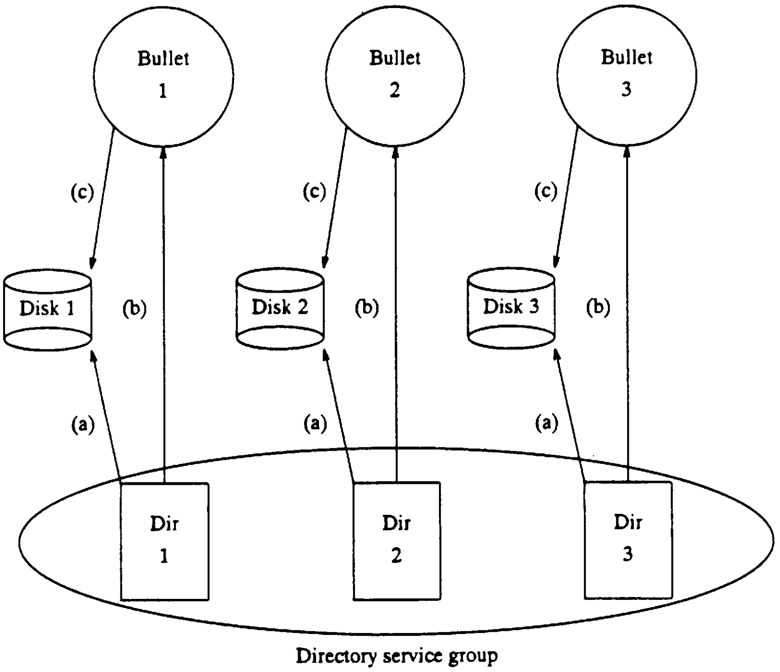


Figure 6: *Organization of the service (a) Administrative data; (b) Directories; (c) Files*

read request. Once these buffered messages are processed, the initiator can perform the read request.

Write operations require communication among the servers. First, the initiator generates a new capability, because all the servers must use the same capability when creating a new directory. Otherwise, some servers may consider a directory capability invalid, whereas others consider it valid. The initiator broadcasts the request to the group using the primitive *SendToGroup* and blocks until the group thread received and executed the request. Once it is unblocked, it sends the result of the request back to the client.

The group thread is continuously waiting for a message sent to the group (i.e., it is blocked in *ReceiveFromGroup*). If *ReceiveFromGroup* returns, the group thread first checks if the call to *ReceiveFromGroup* returned successfully. If not, one of the servers must have crashed. In this case, it rebuilds the group by calling *ResetGroup*, updates its commit block, and calls *ReceiveFromGroup* again. If it does not succeed in building a group with a majority of the members of the original group, the server enters recovery mode.

If *ReceiveFromGroup* returns successfully, the server creates the new directories on its Bullet server, updates its cache, updates its object table, and writes the changed entry in the object table to its disk. As soon as one server writes the new entry to disk, the operation is committed. If no server fails, each server will receive all requests and service all requests in the same order and therefore all the copies of the directories stay consistent. There might be a small delay, but eventually each server will receive all messages.

When the client's RPC returns successfully, the user knows that one new copy of the directory is stored on disk and that at least two other servers have received the request and stored the new directory on disk, too, or will do so shortly. If one server fails, the client can still access its directories.

Let us analyze the cost of a directory operation in terms of communication cost and disk operations. Read operations do not involve communication or disk operations (if the requested directory is in the cache). Write operations require one group message sent with $r = 2$, a Bullet operation to store the new directory, and one disk operation to store the changed entry in the object table.

Recovery Protocol

Block 0, the commit block, contains information that is needed during recovery and is shown in Figure 7. It contains the *configuration vector*. The configuration vector is a bit vector, indexed by server number. If server 2, for example, is down, bit 2 in the vector is set to 0.

During recovery, the sequence number is computed by taking the maximum of all the sequence numbers stored with the directory files and the sequence number stored in the commit block. At first sight it may seem strange that a sequence number is also stored in the commit block, but this is needed for the following case. When a directory is deleted, the Bullet file containing the sequence number is deleted, but the server

1 up?	2 up?	3 up?	Sequence number	Recovering?

Figure 7: *Layout of the commit block*

must store somewhere that it performed an update. The sequence number in the commit block is used for this case. It is only updated when a directory is deleted.

The *recovering* field is needed to keep track if a server crashed during recovery. If this field is set, the server knows that it crashed during recovery. In this case, it sets the sequence number to zero, because its state is inconsistent. It may have recent versions of some directories and old versions of other directories. The sequence number is set to zero to ensure that other servers will not try to update their directories from a server whose state is inconsistent.

A server starts executing the recovery protocol when it is a member of a group that forms a minority or when it comes up after having been down. Two conditions have to be met to recover:

1. The new group must have a majority to avoid inconsistencies during network partitions;
2. The new group must contain the set of servers that possibly performed the latest update.

It is the latter requirement that makes recovery of the group service complicated. During recovery the servers need an algorithm to determine which servers failed last.

Such an algorithm exists; it is due to Skeen [Ske85a] and it works as follows. Each server keeps a *mourned set* of servers that crashed before it. When a server starts recovering, it sets the new group to only itself. Then, it exchanges with all other alive servers its mourned set. Each time it receives a new mourned set, it adds the servers in the received *mourned set* to its own *mourned set*. Furthermore, it puts the server with whom it exchanged the mourned set in the new group. The algorithm terminates when all servers minus the *mourned* set are a subset of the new group.

The complete recovery protocol is as follows. When a server enters recovery mode, it first tries to join the group. If this fails, it assumes that the group is not created yet and it creates the group. If after a certain waiting period, an insufficient number of members joined the group, it leaves the group and starts all over again. It may have happened that two servers recreated the group (e.g., two servers on each side of the network partition) and that they both cannot acquire a majority of the members.

Once a server has created or joined a group that contains a majority of all directory servers, it executes Skeen's algorithm to determine the set of servers that crashed last, the *last set*. If this set is not a subset of the new group, the server starts all over again, waiting for servers from the *last set* to join the group. If the *last set* is a subset of the new group, the new group has the most recent version of the directories. The server determines who in the group has them and gets them. Once it is up-to-date, it writes the new configuration to disk and enters normal operation.

The recovery protocol can be improved. Skeen's algorithm assumes that network partitions do not occur. To make his algorithm work for our assumption, we forced the servers that have a minority to fail. Now the recovery protocol will fail in certain cases in which it is actually possible to recover. Consider the following sequence of events. Server 1, 2, and 3 are up; server 3 crashes; server 1 and 2 form a new group; server 2 crashes. Now as we want to tolerate network partitions correctly, we forced server 1 to fail. However, this is too strict. If server 1 stays alive and server 3 is restarted, server 1 and 3 can form a new group, because server 1 must have performed all the updates that server 2 could have performed. The rule in general is that two servers can recover, if the server that did not fail has a higher *sequence number*, as in this case it is certain that the new member has not formed a group with the (now) unavailable member in the meantime. We will incorporate this improvement in our directory service in the near future.

6. Performance of the Directory Service

The directory service has been used in an experimental environment for several months. It runs on machines comparable to a Sun3/60 connected by 10 Mbit/s Ethernet. The Bullet servers run on Sun3/60s and are equipped with Wren IV SCSI disks.

We have measured the performance of three kinds of operations. The results are shown in Figure 8. The first experiment measures the time to append a new (name, capability) pair to a directory and delete it subsequently (e.g., appending and deleting a name for a temporary file). The second experiment measures the time to create a 4-byte file, register its capability with the directory service, look up the name, read the file back from the file service, and delete the name from the directory service. This corresponds with the use of a temporary file that is the output of the first phase of a compiler and then is used as an input file for the second phase. Thus, the first experiment measures only the directory service, while the second experiment measures both the directory and file service. The third experiment measures the performance of the directory server for read operations.

For comparison reasons, we ran the same experiments using Sun NFS; the results are listed in the second column. The measurements were run on SunOS4.1.1 and the file used was located in `/usr/tmp/`. NFS does not provide any fault tolerance or consistency (e.g., if another client has cached the directory, this copy will not be updated consistently when the original is changed). Compared to NFS, providing high reliability and availability costs a factor of 2.1 in performance for the "append-delete" test and 1.9 in performance for the "tmp file" test.

The dominant cost in providing a fault-tolerant directory service is the cost for doing the disk operations. Therefore, we have implemented a third version of the directory service, which does not perform any disk operations in the critical path. Instead of directly storing modified directories on disk, this implementation stores the modifications to a directory in a 24Kbyte Non Volatile RAM (NVRAM). When the server is idle, it applies the modifications logged in NVRAM to the directories stored on disk. Because NVRAM is a reliable medium, this implementation provides the same degree of fault tolerance as the other implementations, while the performance is much better. A similar optimization has been used in [Dan87a, Lis91a, Har92a].

Operation	Group	Sun NFS	Group +NVRAM
Append-delete	184	87	27
Tmp file	215	111	52
Directory lookup	5	6	5

Figure 8: *Performance of 3 kinds of directory operations (times in msec)*

Using NVRAM, some sequences of directory operations do not require any disk operations at all. Consider the use of `/tmp`. A file written in `/tmp` is often deleted shortly after it is used. If the append operation is still logged in NVRAM when the delete is performed, then both the append and the delete modifications to `/tmp` can be removed from NVRAM without executing any disk operations at all.

We have implemented and measured a version of the directory service that uses NVRAM. Using group communication and NVRAM, the performance improvements for the experiments are enormous (see third column in Figure 8). This implementation is 6.8 and 4.1 times more efficient than the pure group implementation. The implementation based on NVRAM is even faster than Sun NFS, which provides less fault tolerance and has a lower availability.

7. Conclusion

Six design issues are important in group communication: addressing, reliability, ordering, delivery semantics, response semantics, and group structure. We have described the choices that have been made for Amoeba. Amoeba groups are addressed by a port and provide reliable globally-ordered communication. Furthermore, users can trade performance for fault tolerance.

To implement group communication, Amoeba uses a centralized negative acknowledgement protocol. The global ordering is enforced by a centralized machine, called the sequencer. Instead of acknowledging every messages, members of the group piggyback the sequence number for the latest received messages on messages sent to the sequencer. The result is two simple and efficient protocols: the PB and BB protocols. If no failures occur, both protocols need on average only slightly more than two messages per reliable globally-ordered group message.

To illustrate the usage of group communication, we discussed the design and implementation of Amoeba's directory service. To achieve high availability and high reliability, the directory service replicates directories on three machines, each with their own disk. The replicas of a directory are kept consistent using group communication. We described two implementations of the directory service: one using NVRAM and one without NVRAM. NVRAM is used to avoid disk operations in the critical path.

Acknowledgements

Henri Bal, Leendert van Doorn, Greg Sharp, and Mark Wood provided comments on drafts of this paper, which improved its content and presentation substantially.

References

[Abb85a] A. El Abbadi, D. Skeen, and F. Cristian, "An Efficient, Fault-Tolerant Algorithm for Replicated Data Management," *Proc. Fifth Symposium on Principles of Database Systems*, Portland, OR, pp. 215-229 (March 1985).

[Bal90a] H. E. Bal, *Programming Distributed Systems,* Silicon Press, Summit, NJ (1990).

[Bir87a] K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. Comp. Syst.* **5**(1), pp. 47-76 (Feb. 1987).

[Bir91a] K. P. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Comp. Syst.* **9**(3), pp. 272-314 (Aug. 1991).

[Bir84a] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.* **2**(1), pp. 39-59 (Feb. 1984).

[Blo87a] J. J. Bloch, D. S. Daniels, and A. Z. Spector, "A Weighted Voting Algorithm for Replicated Directories," *Journal of the ACM* **34**(4), pp. 859-909 (Oct. 1987).

[Che85a] D. R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V kernel," *ACM Trans. Comp. Syst.* **3**(2), pp. 77-107 (May 1985).

[Dan87a] D. S. Daniels, A. Z. Spector, and D. S. Thompson, "Distributed Logging for Transaction Processing," *Proc. ACM SIGMOD 1987 Annual Conference*, San Francisco, CA, pp. 82-96 (May 1987).

[Gue85a] R. Gueth, J. Kriz, and S. Zueger, "Broadcasting Source-Addressed Messages," *Proc. Fifth International Conference on Distributed Computing Systems*, Denver, CO, pp. 108-115 (1985).

[Har92a] S. Hariri, A. Choudhary, and B. Sarikaya, "Architectural Support for Designing Fault-Tolerant Open Distributed Systems," *IEEE Computer* **25**(6), pp. 50-61 (June 1992).

[His90a] A. Hisgen, A. D. Birrell, C. Jerian, T. Mann, M. Schroeder, and C. Swart, "Granularity and Semantic Level of Replication in the Echo Distributed File System," *IEEE TCOS Newsletter* **4**(3), pp. 30-32 (1990).

[Hug88a] L. Hughes, "A Multicast Interface for UNIX 4.3," *Software Practice and Experience* **18**(1), pp. 15-27 (Jan. 1988).

[Hug89a] L. Hughes, "Multicast Response Handling Taxonomy," *Computer Communications* **12**(1), pp. 39-46 (Feb. 1989).

[Kaa91a] M. F. Kaashoek, R. van Renesse, H. van Staveren, and A. S. Tanenbaum, "FLIP: an Internetwork Protocol for Supporting Distributed Systems," IR-251, Vrije Universiteit, Amsterdam (June 1991).

[Kaa92a] M. F. Kaashoek and A. S. Tanenbaum, "Efficient Reliable Group Communication for Distributed Systems," IR-295, Vrije Universiteit,, Amsterdam (July 1992).

[Kaa92b] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum, "Transparent Fault-tolerance in Parallel Orca Programs," *Proc. Symposium on Experiences with Distributed and Multiprocessor Systems III*, Newport Beach, CA, pp. 297-312 (March 1992).

[Kaa92c] M. F. Kaashoek, A. S. Tanenbaum, and K. Verstoep, "An Experimental Comparison of Remote Procedure Call and Group Communication," *Proc. Fifth ACM SIGOPS European Workshop*, Le Mont Saint-Michel, France (Sept. 1992).

[Kun92a] H. T. Kung, "Gigabit Local Area Networks: a Systems Perspective," *IEEE Communications Magazine* **30**(4), pp. 79-89 (April 1992).

[Lia90a] L. Liang, S. T. Chanson, and G. W. Neufeld, "Process Groups and Group Communication: Classification and Requirements," *IEEE Computer* **23**(2) (Feb. 1990).

[Lis91a] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the Harp File System," *Proc. Thirteenth Symposium on Operating System Principles*, Pacific Grove, CA, pp. 226-238 (Oct. 1991).

[Mar88a] K. Marzullo and F. Schmuck, "Supplying High Availability with a Standard Network File System," *Proc. Eighth International Conference on Distributed Computing Systems*, San Jose, CA, pp. 447-453 (June 1988).

[Mis89a] S. Mishra, L. L. Peterson, and R. D. Schlichting, "Implementing Fault-Tolerant Replicated Objects Using Psync," *Proc. Eighth Symposium on Reliable Distributed Systems*, Seattle, WA (Oct. 1989).

[Mul90a] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer* **23**(5), pp. 44-53 (May 1990).

[Ren89a] R. van Renesse, "The Functional Processing Model," Ph.D. Thesis, Vrije Universiteit, Amsterdam (1989).

[Ren89b] R. van Renesse, A. S. Tanenbaum, and A. Wilschut, "The Design of a High-Performance File Server," *Proc. Ninth International Conference on Distributed Computing Systems*, Newport Beach, CA, pp. 22-27 (June 1989).

[Sat90a] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer* **23**(5), pp. 9-22 (May 1990).

[Ske85a] D. Skeen, "Determining the Last Process to Fail," *ACM Trans. Comp. Syst.* **3**(1), pp. 15-30 (Feb. 1985).

[Tan89a] A. S. Tanenbaum, *Computer Networks 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ (1989).

[Tan90a] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. J. Mullender, A. Jansen, and G. van Rossum, "Experiences with the Amoeba Distributed Operating System," *Commun. ACM* **33**(12), pp. 46-63 (Dec. 1990).

[Tan92a] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal, "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer* **25** (Aug. 1992).