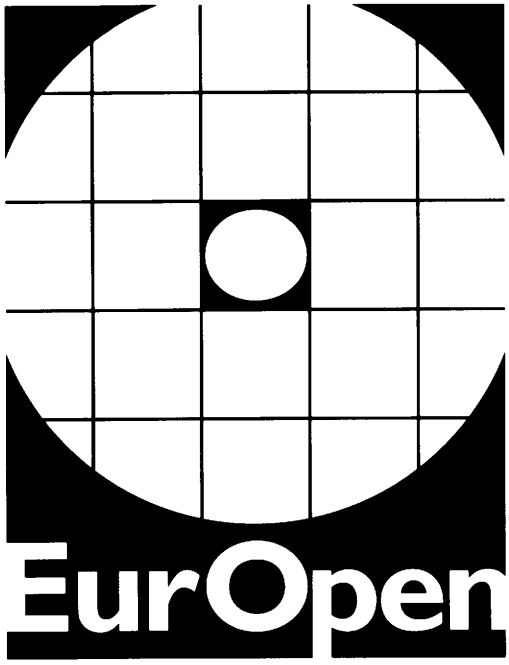
EurOpen

Spring 1991 Conference Proceedings

Distributed Open Systems in Perspective

20 — 24 May
Kulturhuset
Tromsø, Norway



UNIX
Distributed Open Systems
In Perspective

Proceedings of the
Spring 1991 EurOpen Conference

May 20–24, 1991
Kulterhuset,
Tromsø, Norway

ISBN 1 873611 00 5

Further copies of the proceedings may be obtained from:

EurOpen Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

These proceedings were typeset in Times Roman and Courier on a Linotronic L300 PostScript phototypesetter driven by a white swan. PostScript was generated using **refer**, **tt**, **pic**, **psfig**, **tbl**, **sed**, **eqn**, **troff**, **pm** and **psdit**. Laser printers were used for some figures.

ACKNOWLEDGEMENTS

The first conference to be held by EurOpen,† the European Forum for Open Systems, has become fact. The programme addresses one particularly interesting and important theme to our organisation: distributed open systems. All important technical issues have been covered: operating systems, architecture, scheduling, monitoring and control, language issues, and server-based solutions. What the keynote speakers have to say will remain a surprise until the very last moment: make sure you're there to listen. All other presentations have been included in the proceedings.

As the most important part of a conference is the programme, those who have spent time and energy in preparing their papers deserve sincere thanks from the audience, the reader and EurOpen. We should be also very grateful to all others who submitted proposals: the programme committee was faced with more than five times as many proposals as could be accepted. The programme committee has done it's work well: Dag Johanssen, Hans Strack-Zimmerman, Donal Daly and Ernst Janich have invested considerable time and effort in compiling the programme of this conference.

The hosting organisation the NUUG (the Norwegian UNIX Users Group), has been actively involved in the organisation of all aspects of the conference. The staff and students of the Department of Computer Science at the University of Tromsø, and in particular Magnar Antonsen as local chair, have shown tremendous dedication to the success of this event.

Ernst Janich, as EurOpen conference executive, Neil Todd, as EurOpen tutorial executive and the EurOpen Secretariat have again shown that the organisation of conferences as this can work in cooperation with the local organisation. Stuart McRobert, Jan-Simon Pendry and Dave Edmondson in London have again not only provided resources for the production of the proceedings, they have also the necessary perseverance.

EurOpen is grateful to all for their efforts. The programme and papers are most promising: the discussions and presentations should be lively, interesting and educating.

Frances Brazier

These proceedings were specially produced for EurOpen at the Department of Computing, Imperial College, London, using resources generously provided by the Computing Support Group. Thanks to Alexios Zavras for his help and assistance whilst visiting us in January.

– jsp, sm, dme.

† EurOpen, the European Forum for Open Systems has evolved from the former EUUG.

Table of Contents

Author Index

UNIX Conferences in Europe 1977–1991

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

Experiences with Amoeba

Sape J. Mullender

University of Twente, Netherlands
mullender@cs.utwente.nl

Abstract

The Amoeba distributed operating system has been in use now for a few years. It has been used in experiments with parallel algorithms, as a distributed UNIX-like system, in real-time applications, and in event-processing for proton-scattering high-energy physics experiments.

We have discovered many of the strong and the weak points of Amoeba. On the positive side, we are very pleased with our RPC-based communication, with capability-based protection, with a free-standing naming service, the bootstrap service and with the process-management facilities.

On the negative side, we are not happy with our incomplete, non-binary-compatible UNIX functionality, with the flat port name space which limits the ability to scale, with an immutable-file service, and with a kernel implementation of user threads.

In this paper, I will present a brief overview of Amoeba, discuss our experiences with Amoeba, and present some of my current work on the design of a new distributed system.

1. Introduction

The Amoeba Distributed Operating System Project started in 1980 at the Vrije Universiteit in Amsterdam. Its goal was the design and implementation of a true distributed operating system, a system with no single points of failure, a system that would scale to arbitrary size, and a system that would provide at least the functionality and performance of current UNIX systems on a similar equipment basis.

The provision of a UNIX interface was an explicit non-goal, even though we do now run many UNIX applications on Amoeba. In the design of Amoeba, we wanted to be unhampered by existing operating system interfaces in order to address the more fundamental question of what functionality is appropriate to provide and how can one provide it efficiently.

The project began very small: two people worked on it part time for several years. It was only around 1983 that more people started working on Amoeba. In 1984, CWI joined in on the Amoeba project, making it a distributed distributed systems project. In 1990, a full dozen or so people were working full time in the project.

Amoeba will soon be available as a distributed systems research platform for universities and research laboratories. Inquiries should be directed to Prof. Andrew S. Tanenbaum at the Vrije Universiteit (ast@cs.vu.nl).

The remainder of the paper is structured as follows. Section 2 gives a short description of the Amoeba distributed system which makes the rest of this paper readable by itself. Sections 3 and 4 then discuss our positive and negative experience with the system in view of the original goals. Finally, Section 5 draws conclusions and describes the current systems research plans in Twente.

2. The Amoeba Distributed System

Amoeba [Mul90a, Tan90a] is an *object-based system*. Client processes use remote procedure calls to send requests for carrying out operations to objects. Each object is both identified and protected by a *capability*. Capabilities have the set of operations that the holder may carry out on the object coded into them and they contain enough redundancy and cryptographic protection to make it infeasible to guess an object's capability. Thus, keeping capabilities secret is the key to protection in Amoeba.

Objects are implemented in terms of server processes that manage them. Capabilities have the identity of the object's server encoded into them so that, given its capability, the system can easily find a server process for an object. The RPC system guarantees that requests and replies are delivered at most once and only to authorized processes.

Amoeba's communication model is that of a client thread making remote procedure calls [Bir84a] on objects to manipulate them. The model is implemented in terms of the client sending a *request* message to the *service* that manages the object. A server thread will carry out the request and return a *reply* message back to the client.

Conceptually, clients communicate with active objects. An active object is implemented as a set of (multithreaded) server processes that manage the (passive) representation of the object – as well as the representations of many other objects, usually. A set of server processes that jointly manages a collection of objects of the same type is referred to as a *service*.

The interface for manipulating a type of object is called the object type's *class*. Classes can be composed hierarchically; that is, a class may contain the operations from one or several more primitive classes. This *multiple inheritance* mechanism allows many services to inherit the same interfaces for simple object manipulations, such as for changing the protection properties on an object, or deleting an object. It also allows all servers manipulating objects with file-like properties to inherit the same interface for low-level file I/O: read, write, append. The mechanism resembles the file-like properties of UNIX pipe and device I/O: the UNIX *read* and *write* system calls can be used on files, terminals, pipes, tapes and other I/O devices. But for more detailed manipulation, specialized calls are available (*ioctl*, *popen*, etc.). Interfaces for object manipulation are specified in a notation, called the Amoeba Interface Language (AIL) [Ros89a], which resembles the notation for procedure headers in C with some extra syntax added. This allows automatic generation of client and server stubs.

AIL generates the code for marshalling and unmarshalling the parameters of remote procedure calls into and out of message buffers and then

calls on Amoeba's transport mechanism for the delivery of request and reply messages. Messages consist of two parts, a *header* and a *buffer*. The header has a fixed format and contains addressing information (among which, the capability of the object that the RPC refers to), an operation code which selects the function to be called on the object, and some space for additional parameters. The size of the buffer may range from 0 and 32K bytes and is mostly used to hold the variable-length arguments of RPCs. A file read or write call, for instance, uses the message header for the operation code plus the length and offset parameters, and the buffer for the file data. With this set-up, marshalling the file data takes zero time, because the data can be transmitted directly from and to the arguments specified by the program.

The transport mechanism itself consists of the server calls *get_request* and *put_reply*, usually arranged in a loop of a server thread as follows:

```
/* allocate a request buffer */
do {
        get_request(port, reqheader, reqbuffer, reqbuflen);
        /* Unmarshal the request parameters */
        /* Call the implementation routine */
        /* Marshal the reply parameters */
        put_reply(repheader, repbuffer, repbuflen);
} while (1);
```

Get_request blocks until a request comes in. *Put_reply* blocks until the header and buffer parameters can be reused. A client sends a request and waits for a reply by calling

```
do_operation(reqheader, reqbuffer, reqbuflen,
        repheader, repbuffer, repbuflen);
```

These three calls are implemented as system calls of the Amoeba kernel. The protocol for the transport of messages is network dependent. Over wide-area networks, standard protocols, such as IP or X.25 are used. Over local networks, specialized protocols, designed for fast response and high throughput are used.

Before a request for an operation on an object can be delivered to a server thread that manages the object, the location of such a thread must be found. Capabilities consist of 3 parts, a *port*, which identifies the service that manages the object that the capability refers to, a *location hint* which can be used to provide a clue for the location of the object, and an *object part* that identifies the object further within the service. The structure of a capability is shown in Figure 1.

When a server thread makes a *get_request* call, it provides its service port to the system. When a client thread calls *do_transaction*, it is the system's job to find a server thread with an outstanding *get_request* that matches the port in the capability provided by the client.

We call the process of finding the address of such a server thread *locating*. If objects and processes did not move, locating servers would be easy, but they do so it isn't. The current technique for locating servers for a service is to *broadcast* for them: When the location of a service is not know, the client's kernel broadcasts a "where-are-you" packet. To this, the servers for the service react and the first reaction is selected. Kernels cache location information to avoid broadcasting as much as possible.

When Amoeba is run over wide-area networks, which do not support broadcast, *port publishing* is used: Wide-area services announce their

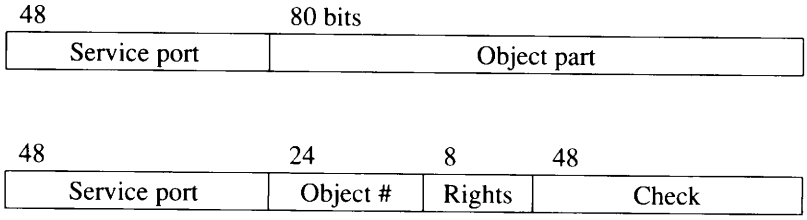


Figure 1: *Structure of a Capability*

Above: Structure of a capability as defined by Amoeba.
Below: Detailed structure of a capability as defined by many services. The *service port* identifies the service that manages the object. The *object part* identifies the object within the service. *Object number* is usually an index into a table of objects within the service; *rights* define what operations the client holding the capability is allowed to invoke and *check* is a cryptographic checksum that prevents forgery.

presence by sending a message to the wide-area network *gateway* process, which broadcasts the message to all other gateways. The gateways then answer where-are-you broadcasts from clients on the local network.

When a client process sends requests to the file server, one wants to make quite certain that they go to the file server and not to an intruder process. When the reply comes back, one also wants to be absolutely sure that the reply was sent by the file server and no other process. Authentication mechanisms are needed that prevent one process from impersonating another and prevent unauthorized processes from looking at other processes' messages.

The way it was described earlier, the interface for message exchange is insecure. A malicious client in possession of the capability of a file knows the port of the file service and it can therefore do *get_request*s on the file server's port in order to intercept read and write requests from unsuspecting clients of the file server. Fortunately, the interface does not work exactly as described before.

In order to receive requests, a server has to know the *get port* of the service it implements. In order to send requests to a service, the client has to know (as part of a capability) the *put port* of that service. To make it feasible for the system to deliver requests addressed with a put port to a process asking for requests on a get port, it has to know the relationship between get ports and put ports.

This relationship is provided by a one-way function F. One-way functions are functions that take an argument from a very large domain and compute a value in a range of approximately (or exactly) the same size. The special property of these functions is that, although it is reasonably simple to compute the function, finding an inverse for a arbitrary value of the function is computationally infeasible [Eva74a]. F is a publicly known one-way function that defines the relationship between a get port G and a put port P as follows:

$$P = F(G)$$

Thus, when one knows the get port, the associated put port can be straightforwardly computed, but knowledge of a put port alone cannot be used to find the associated get port.

A server does *get_request*s with the get port as one of the arguments and the client does *do_operation*s with the service's put port in the capability argument. In order to get replies back equally securely, clients also use a pair of ports. Internal to the implementation of *do_operation* the client asks for replies addressed to the client's get

port and the server addresses the reply with the client's put port. The client's get port may be viewed as the client process' UID – it is kept in the process' environment and normally inherited on process creation.

As long as one assumes that Amoeba processes use the *get_request*, *put_reply* and *do_operation* interfaces exclusively, this mechanism is secure. Unfortunately, it is very difficult to enforce the use of this interface and prevent use of another. If just one machine on an Ethernet cheats, the security of the whole system is compromised.

The friendly-environment implementation of the Amoeba protection mechanism assumes that the exclusive use of the Amoeba communication interface can indeed be enforced. The implementation is in the Amoeba kernel and also in the Amoeba-communication package of our UNIX kernels (in a UNIX device driver). Under the assumption that the Amoeba kernel is tamper proof, that the super-users on UNIX can be trusted, and that there are no other untrustworthy machines on the network, the friendly-environment implementation is secure. In the Amoeba systems in use at CWI and VU, friendly-environment protection is deemed sufficient – we keep few secrets anyway.

A secure version of this protection mechanism using cryptographic techniques has been designed but not yet implemented.

Although, at the system level, objects are identified by their capabilities, at the level where most people program and do their work, objects are named using a human-sensible hierarchical naming scheme. The mapping is carried out by the *directory server*. It maintains a mapping of ASCII path names onto capabilities. For replicated objects it can map the name onto a set of capabilities, one for each replica. The directory server has mechanisms for doing atomic operations on arbitrary collections of name-to-capability mappings. Thus, as long as the objects themselves are used as immutable objects, the directory server can be used as a simple transaction-management system.

Hierarchical directory structures are ideal for implementing partially shared name spaces. Objects that are shared between the members of a project team can be stored in a directory that only team members have access to. By implementing directories as ordinary objects with a capability that is needed to use them, members of a group can be given access by giving them the capability of the directory, while others can be withheld access by not giving them the capability. A capability of a directory is thus a capability for lots of other capabilities.

It does not make sense in this naming hierarchy of capabilities to have a common root: Through the root, every user would be able to access exactly the same set of capabilities and this is obviously not desirable. Instead, every *principal* – that is, every entity that can maintain a set of private objects, a human user, a service, or something else – has a *home directory* which serves as the root of that principal's naming universe. When an individual logs in, a *login server*, which is assumed to be secure and trusted, starts a command interpreter and provides it with the capability of his home directory. From the home directory, all the capabilities that a user needs must be reachable.

Amoeba processes can have multiple threads of control. A process, essentially, consists of a segmented virtual address space and one or more threads. Processes can be remotely created, destroyed, checkpointed, migrated and debugged.

On a uniprocessor, threads run in quasi-parallel; on a shared-memory multiprocessor, as many threads can run simultaneously as there are processors. Processes can not be split up over more than one machine.

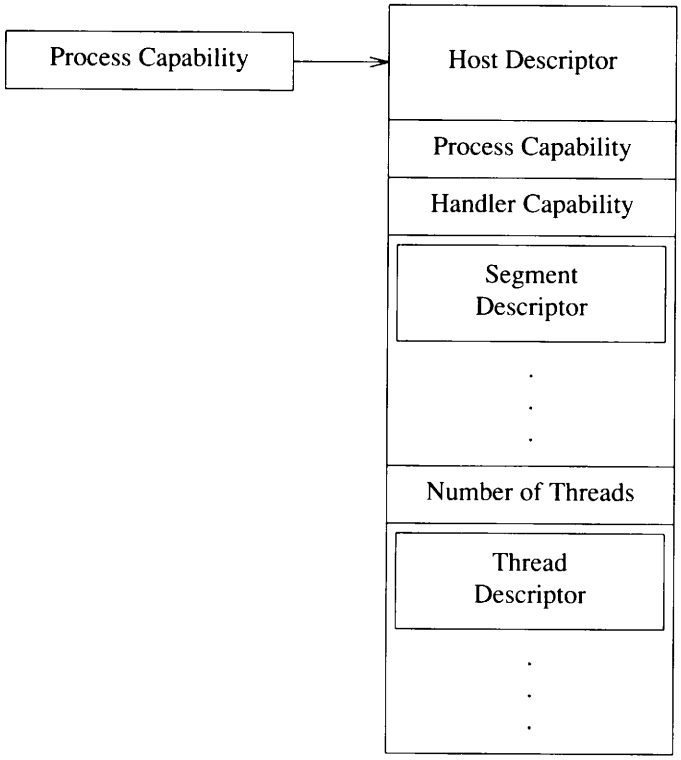


Figure 2: *Layout of a process descriptor*

Processes have explicit control over their address space. They can add new segments to their address space by *mapping them in* and remove segments by *mapping them out*. Besides virtual address and length, a *capability* can be specified in a map operation. This capability must belong to a file-like object which is read by the kernel to initialize the new segment. This allows processes to do mapped-file I/O.

A process is created by sending a *process descriptor* in an *execute process* request to a kernel. A process descriptor consists of four parts as shown in Figure 2. The host descriptor describes on what machine the process may run, e.g., its instruction set, extended instruction sets (when required), memory needs, etc., but also it can specify a class of machines, a group of machines or a particular machine. A kernel that does not match the host descriptor will refuse to execute the process.

Then come the capabilities, one is the capability of the process which every client that manipulates the process needs. The other is the capability of a *handler*, a service that deals with process exit, exceptions, signals and other anomalies of the process

The memory map has an entry for each segment in the address space of the process to be. An entry gives virtual address, segment length, how the segment should be mapped (read only, read/write, execute only, etc.), and the capability of a file or segment from which the new segment should be initialized.

The thread map describes the initial state of each of the threads in the new process, processor status word, program counter, stack pointer, stack base, register values, and system call state. This rather elaborate notion of thread state allows the use of process descriptors not only for the representation of executable files, but also for processes being migrated, being debugged or being checkpointed.

In most operating systems, system call state is a very large and complicated to represent outside an operating system kernel. In Amoeba, fortunately, there are very few system calls that can block in the kernel. The most complicated ones are those for communication: *do_operation* and *get_request*.

Processes can be in two states, *running*, or *stunned*. In the stunned state, a process exists, but does not execute instructions. A process being debugged is in the stunned state, for example. The low-level communication protocols in the operating system kernel respond with "this-process-is-stunned" messages to attempts to communicate with the process. The sending kernel will keep trying to communicate until the process becomes running again or until it is killed. Thus, communication with a process being interactively debugged continues to work.

A running process can be stunned by a stun request directed to it from the outside world (this requires the stunner to have the capability of the process which is taken as evidence it is the owner), or by an uncaught exception. When the process becomes stunned, the kernel sends its state in a process descriptor to a *handler* whose identity is a capability which is part of the process' state. After examining the process descriptor, and possibly modifying it or the stunned process' memory, the handler can either reply with a *resume* or *kill* command.

Debugging processes is done with this mechanism. The debugger takes the role of the handler. Migration is also done through stunning. First, the candidate process is stunned; then, the handler gives the process descriptor to the new host. The new host fetches memory contents from the old host in a series of file read requests, starts the process and returns the capability of the new process to the handler. Finally, the handler returns a *kill* reply to the old host. Processes communicating with a process being migrated will receive "process-is-stunned" replies to their attempts until the process on the old host is killed. Then they will get a "process-not-here" reaction. After locating the process again, communication will resume with the process on the new host.

The mechanism allows command interpreters to cache process descriptors of the programs they start and it allows kernels to cache code segments of the processes they run. Combined, these caching techniques make process start-up times very short.

Amoeba is a new operating system with a system interface that is quite different to that of the popular operating systems of today. Amoeba was developed by a group of people who all used UNIX as their operating system vehicle, so it will come as no surprise that the absence of UNIX tools on Amoeba would be a major obstacle to using Amoeba for daily work.

The Amoeba support for UNIX is called *Ajax*, not after the ancient Greek hero of that name, but because the name starts with an "A" and ends with an "X" and because the training grounds of the Ajax soccer club are visible from the CWI windows. It was designed and implemented by Guido van Rossum who used the approach that 90% of the effort goes into getting the last 10% of the UNIX utilities running on Amoeba and nobody needs those UNIX utilities anyway.

A library was developed that implemented the most common UNIX system calls using data structures in the library and calls on the Bullet file server, the Soap directory server and the Amoeba process management facilities. The system calls implemented initially were those for file I/O (*open*, *close*, *dup*, *read*, *write*, *lseek*) and a few of the *ioctl* calls for ttys. These were very easy to implement under Amoeba

(about two week's work) and were enough to get a surprising number of UNIX utilities to run.

Currently, about 100 utilities have been made to run on Amoeba without any changes to the source code. The X-window system has been ported to Amoeba and supports the use of both TCP/IP and Amoeba RPC there so that an X client on Amoeba can still converse with an X server on Amoeba and vice versa.

We have found that the availability of the UNIX utilities have made the transition to Amoeba much easier. Slowly, however, many of the UNIX utilities will be replaced by utilities that are better adapted to the Amoeba distributed environment.

3. Things we like

Amoeba was designed with a very different philosophy from UNIX. Amoeba does not, at the operating system level, have files, pipes, *fork* or *exec*. Instead, Amoeba has three fundamental notions: *address spaces*, *threads*, and *communication between threads*. These have proven to be extremely powerful abstractions for building distributed systems and, as we shall see later, they did not get in the way of providing "conventional" operating system functionality.

Capabilities provide a uniform way of addressing objects (and, indirectly, services) and have given us a system-wide protection mechanism which has proven to be very effective. Capability-based protection is quite different from the conventional access-control-list protection that we are so familiar with in, for instance, UNIX. It takes some time to get used to capabilities, but now they are as natural a way to architect system security as any.

Capability-based addressing and our interface-definition language AIL make object interaction very uniform and very natural. The directory service, in addition, provides a mapping of hierarchical ASCII names to objects' capabilities so that every object, independent of its type, can be named in the same name space. UNIX names files in one name space (hierarchical ASCII strings), processes in another (PIDs) and users in a third (UIDs) and fourth (*mullender@cs.utwente.nl*), while some objects (e.g., pipes) are not named at all. In Amoeba, all of these are named using capabilities at one level and directory paths in another.

AIL's multiple-inheritance mechanism allows the sharing of interfaces across a range of object types. For example, files, terminal I/O, and our equivalence of pipes all share an interface for reading and writing so that to applications that write sometimes to a file and sometimes to a terminal, you only need to pass the output object's capability and not its type.

The RPC transport mechanism has proven to be among the best performing in any system. RPC mechanisms on UNIX certainly perform nowhere as well. Obviously, we are pretty happy with this. The success of our high-speed RPC can be attributed to a number of factors. First, we have made sure that most large RPCs can go directly out of existing in-memory buffers to the correct target in-memory location. This is illustrated nicely by the file server: It caches files in memory and can satisfy read requests by sending replies directly out of the file cache, saving copying. The file server also does not have to reserve space for message headers, since header and buffer can be sent separately.

RPC transport is made fast also by the use of a specialized light-weight transport protocol, tailored for use over local networks where packet losses are rare. A typical wide-area network RPC consists of three RPCs: a local one between client and gateway, one between gateways over the WAN and another local one between gateway and server. The two gateways can then use a different wide-area network transport protocol (such as an X.25 or TCP/IP-based protocol).

But the most important cause of high-speed is the way in which the transport protocols have been integrated with the operating system kernel. This needs to be done very carefully. A redesign of the UNIX kernel can give similar speedups as suggested by the Sprite kernel [Ous88a].

Amoeba's directory server is a mapping service of ASCII path names to capabilities. Mutable objects can be constructed out of immutable ones by changing the directory mapping from one immutable object into the next. This is done, for instance, to reflect change in the file system which, itself, provides immutable-file service. The directory server has mechanisms to replace sets of mappings atomically and it has support for managing replicated objects. Using the directory service, it is very simple to carry out transaction and create some fault tolerance, even though the underlying services have by themselves no support for this. So far, however, these mechanisms have not been used much.

The Amoeba process management mechanisms have proven to be very valuable. The process descriptor was originally invented so that process migration might be possible. As it turns out now, process migration is hardly used, but the mechanisms themselves are still extremely useful.

A process descriptor is a portable description of a multithreaded process in an arbitrary state. It is thus used in debugging, checkpointing, and *forking*. Our current debugger, derived from the GNU debugger, uses process descriptors to communicate between the debugger and the debugged process. The Ajax server implements the fork system call by checkpointing the UNIX process at the *fork* call and creating a copy on another host. *Fork* under Ajax is thus by default remote.

4. Things we do not like

Even though we have had a free hand in designing Amoeba to be anything we wanted, some design decisions have turned out to be not to our liking. There are some things that we know now how to do better, and others where we know what's wrong, but we don't really know what to do about it.

The capabilities that were praised so highly in the previous section are also the source of some fundamental problems. First, a capability, through the port contained in it, tells us the name of the service managing it. If the service is replicated, any server must be able to give access to the object. This is not realistic in very large systems, where a service can have hundreds of servers, but each object is only held by a few of them. In a capability, the information of what servers exactly store a particular object cannot be represented, especially, because, although the service itself remains, its servers go through generations of incarnations and this cannot be communicated to the capabilities which are stored by the individual users.

As usual, this problem could be solved by the proverbial "extra level of indirection" as provided, for instance, by the DEC Global Name Server

[Lam86a] which is now part of OSF's Distributed Computing Environment. Here, a directory entry can be made to store not only a capability for the object, but also the names of the servers storing the object. By looking up the names of one or more of these servers, one would find a directory entry with location information, authentication information, etc. Since the server directory entry is maintained by the service, it can be kept up to date.

Amoeba also provides a directory service, but does not provide it as a fundamental system service – in other words, Amoeba must be usable without one. I am inclined to believe now that this may have been an error and in my next system I intend to introduce this extra level of indirection.

The problem just sketched is an example of a "problem of scale", a problem that appears when the system grows beyond a certain size. Avoiding problems of scale is one of the hardest problems in distributed systems research. It will thus come as no surprise that Amoeba has more of them, also related to capabilities.

Capabilities contain ports for addressing services. These ports are flat 48-bit names and do not reveal where a server for that port might be located. To find this out, Amoeba uses broadcast. The result of a broadcast query for a server on a particular port is cached so that it does not have to be done very often. It is clear that this technique of locating servers does not scale very well. In wide-area networks, therefore, Amoeba uses a different strategy. Services available over wide-area networks *publish* their port: They give their port to their local gateway server which writes it to stable storage and broadcasts it to all other gateway servers in the network. Since this happens only when a new service for the wide-area network community is created, these broadcasts are infrequent. Also, the number of local networks and thus the number of gateways is one or two orders of magnitude smaller than the number of potential clients or the total number of services. Still, this algorithm does not scale very well to systems of world-wide scale. It also presents a security problem that is currently unsolved.

This problem of scale forms another reason to choose a hierarchical directory service as a better-scaling alternative.

A problem of a different kind is presented by the way in which processes are allocated to processors. Since all process creation is essentially remote, Amoeba can take relatively little advantage of locality of reference. It is, for example, rather useless to cache files on client machines, because they are seldom used on the same machine again: An edit-compile-link-run sequence leads to editing the source file in one location, compiling it in another, linking the resulting object files in yet another location, and running the executable in a different on again. Caching is useless here, especially, because at the next iteration the compiler, linker and executable will run on different machines again.

The current Bullet file server in Amoeba does no client caching for this reason and it should come as no surprise that the Sprite file system, which does, outperforms the Bullet file system in all but contrived cases. This is a problem that can, in principle, be solved. One can implement client caching relatively straightforwardly, and one can make the process allocation algorithms clever enough to use knowledge of what is cached where when allocating processes to processors.

5. Conclusions

We have learned very much from the design and subsequent use of Amoeba and we believe that others have too. Amoeba has been one of the pioneering distributed systems and, as such, has been very successful. As it gets exercised by day-to-day use, Amoeba will continue to improve and overcome some of the problems discussed above.

Amoeba is now, or will soon be available as a distributed systems platform for research and education in distributed systems. Compared to today's UNIX it may still appear to be rather unfunctional, but, unlike UNIX, it is a true transparently distributed system and, unlike UNIX, it has not yet had a history of nearly twenty years of improvement (although, of course, not all changes to UNIX have been improvements).

The Amoeba project will continue at the Vrije Universiteit, albeit without the author.

References

[Bir84a] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* **2**(1), pp. 39–59 (Feb 1984).

[Eva74a] A. Evans, W. Kantrowitz, and E. Weiss, "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Communications of the ACM* **17**(8), pp. 437–442 (Aug 1974).

[Lam86a] B. Lampson, "Designing a Global Name Service," *Proceedings of the 5th ACM Conference on Principles of Distributed Computing*, Calgary, Canada, pp. 1–10, 1985 Invited Talk (Aug 1986).

[Mul90a] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and J. M. van Staveren, "Amoeba – A Distributed Operating System for the 1990s," *IEEE Computer* **23**(5) (May 1990).

[Ous88a] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *IEEE Computer* **21**(2), pp. 23–35 (Feb 1988).

[Ros89a] G. van Rossum, "AIL – A Class-Oriented Stub Generator for Amoeba," *Workshop on Progress in Distributed Operating Systems and Distributed Systems Management*, Berlin **433**, Springer Verlag Lecture Notes in Computer Science (1989).

[Tan90a] A. S. Tanenbaum, R. van Renesse, J. M. van Staveren, G. J. Sharp, S. J. Mullender, A. J. Jansen, and G. van Rossum, "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM* (Dec 1990).

A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility

Allan Bricker Michel Gien Marc Guillemont
Jim Lipkis Douglas Orr Marc Rozier

Chorus systèmes, France
{allan | mg | mgu | lipkis | doug | mr}@chorus.fr

Abstract

An important trend in operating system development is the restructuring of the traditional monolithic operating system kernel into independent servers running on top of a minimal nucleus or "microkernel". This approach arises out of the need for modularity and flexibility in managing the ever-growing complexity caused by the introduction of new functions and new architectures. In particular, it provides a solid architectural basis for distribution, fault tolerance, and security. Microkernel-based operating systems have been a focus of research for a number of years, and are now beginning to play a role in commercial UNIX systems.

The ultimate feasibility of this attractive approach is not yet widely recognised, however. A primary concern is efficiency: can a microkernel-based modular operating system provide performance comparable to that of a monolithic kernel when running on comparable architectures? The elegance and flexibility of the client-server model may exact a cost in message-handling and context-switching overhead. If this penalty is too great, commercial acceptance will be limited. Another pragmatic concern is compatibility: in an industry relying increasingly on portability and standardisation, compatible interfaces are needed not only at the level of application programs, but also for device drivers, streams modules, and other components. In many cases, binary as well as source compatibility is required. These concerns affect the structure and organisation of the operating system.

The Chorus team has spent the past six years studying and experimenting with UNIX "kernelisation" as an aspect of its work in modular distributed and real-time systems. In this paper we examine aspects of the current CHORUS system in terms of its evolution from the previous version. Our focus is on pragmatic issues such as performance and compatibility, as well as considerations of modularity and software engineering.

This paper is a revision of an article that appeared in the Proceedings of the Winter 1991 Usenix Conference, Dallas, TX.

1. Microkernel Architectures

A recent trend in operating system development consists of structuring the operating system as a modular set of system servers which sit on top of a minimal microkernel, rather than using the traditional monolithic structure. This new approach promises to help meet system and platform builders' needs for a sophisticated operating system development environment that can cope with growing complexity, new architectures, and changing market conditions. In this operating system architecture, the microkernel provides system servers with generic services, such as processor scheduling and memory management, independent of a specific operating system. The microkernel also provides a simple Inter-Process Communication (IPC) facility that allows system servers to call each other and exchange data independent of where they are executed, in a multiprocessor, multicomputer, or network configuration.

This combination of primitive services forms a standard base which in turn supports the implementation of functions that are specific to a particular operating system or environment. These system-specific functions can then be configured, as appropriate, into system servers managing the other physical and logical resources of a computer system, such as files, devices and high-level communication services. We refer to such a set of system servers as a *subsystem*. Real-time systems tend to be built along similar lines, with a very simple generic executive supporting application-specific real-time tasks.

1.1. UNIX and Microkernels

UNIX introduced the concept of a standard, hardware-independent operating system, whose portability allowed platform builders to reduce their time to market by obviating the need to develop proprietary operating systems for each new platform.

However, as more function and flexibility is continually demanded, it is unavoidable that today's versions become increasingly more complex. For example, UNIX is being extended with facilities for real-time applications and on-line transaction processing. Even more fundamental is the move toward distributed systems. It is desirable in today's computing environments that new hardware and software resources, such as specialised servers and applications, be integrated into a single system, distributed over a network. The range of communication media commonly encountered includes shared memory, buses, high-speed networks, local-area networks, and wide-area networks. This trend to integrate new hardware and software components will become fundamental as collective computing environments emerge.

To support the addition of function to UNIX and its migration to distributed environments, it is desirable to map UNIX onto a microkernel architecture, where machine dependencies may be isolated from unrelated abstractions and facilities for distribution may be incorporated at a very low level.

The attempt to reorganise UNIX to work within a microkernel framework poses problems, however, if the resultant system is to behave exactly as a traditional UNIX implementation. A primary concern is efficiency: a microkernel-based modular operating system must provide performance comparable to that of a monolithic kernel. The elegance and flexibility of the client-server model may exact a cost in

message-handling and context-switching overhead. If this penalty is too great, commercial acceptance will be limited. Another pragmatic concern is compatibility: in an industry relying increasingly upon portability and standardisation, compatible interfaces are needed not only at the level of application programs, but also for device drivers, streams modules, and other components. In many cases binary as well as source compatibility is required. These concerns affect the structure and organisation of the operating system.

There is work in progress on a number of fronts to emulate UNIX on top of a microkernel architecture, including the Mach [Gol90a], V [Che90a], and Amoeba [Tan90a] projects. Plan 9 from Bell Labs [Pik91a] is a distributed UNIX-like system based on the "minimalist" approach. CHORUS versions V2 and V3 represent the work we have done to solve the problems of compatibility and efficiency.

1.2. The CHORUS Microkernel Technology

The Chorus team has spent the past six years studying and experimenting with UNIX "kernelisation" as an aspect of its work in modular, distributed and real-time systems. The first implementation of a UNIX-compatible microkernel-based system was developed during 1984 through 1986 as a research project at INRIA. Among the goals of this project were to explore the feasibility of shifting as much function as possible out of the kernel and to demonstrate that UNIX could be implemented as a set of modules that did not share memory. In late 1986, an effort to create a new version, based on an entirely rewritten CHORUS nucleus, was launched at Chorus systèmes. The current version maintains many of the goals of its predecessor and adds some new ones, including real-time support and – not incidentally – commercial viability. A UNIX subsystem compatible with System V Release 3.2 is currently available, with System V Release 4.0 and 4.4BSD systems under development. The System V Release 3.2 implementation performs comparably with well-established monolithic-kernel systems on the same hardware, and better in some respects. As a testament to its commercial viability, the system has been adopted for use in commercial products ranging from X terminals and telecommunication systems to mainframe UNIX machines.

In this paper we examine aspects of the current CHORUS system in terms of its evolution from the previous version. Our focus is on pragmatic issues such as performance and compatibility, as well as considerations of modularity and software engineering.

In section 2, we review the previous CHORUS version. Section 3 evaluates the previous version and discusses how the lessons learned from its implementation led to the main design decisions for the current version. The subsequent sections focus on specific aspects of the current design.

2. CHORUS V2 Overview

The CHORUS project, while at INRIA, began researching distributed operating systems with CHORUS V0 and V1. These versions proved the viability of a modular, message-based distributed operating system, examined its potential performance, and explored its impact on distributed applications programming.

Based on this experience, CHORUS V2 [Arm86a, Roz87a] was developed. It represented the first intrusion of UNIX into the peaceful

CHORUS landscape. The goals of this third implementation of CHORUS were:

- To add UNIX emulation to the distributed system technology of CHORUS V1;
- To explore the outer limits of "kernelisation"; demonstrate the feasibility of a UNIX implementation with a minimal kernel and semi-autonomous servers;
- To explore the distribution of UNIX services;
- And to integrate support for a distributed environment into the UNIX interface.

Since its birth, the CHORUS architecture has always consisted of a modular set of servers running on top of a microkernel (the nucleus) which included all of the necessary support for distribution.

The basic execution entities supported by the V2 nucleus were mono-threaded *actors* running in user mode and isolated in protected address spaces. Execution of actors consisted of a sequence of "processing-steps" which simulated atomic transactions: ports represented operations to be performed; messages would trigger their invocation and provide arguments. The execution of remote operations were synchronised at explicit "commit" points. An ever-present concern in the design of CHORUS was that fault-tolerance and distribution are tightly coupled; hardware redundancy both increases the probability of faults and gives a better chance to recover from these faults.

Communication in CHORUS V2 was, as in many current systems, based upon the exchange of messages through *ports*. Ports were attached to actors, and had the ability to migrate from one actor to another. Furthermore, ports could be gathered into *port groups*, which allowed message broadcasting as well as functional addressing. For example, a message could be directed to all members of a port group or to a single member port which resided on a specified site. The port group mechanism provided a flexible set of client-server mapping semantics including dynamic reconfiguration of servers.

Ports, port groups, and actors were given global unique names, constructed in a distributed fashion by each nucleus for use only by the nucleus and system servers. Private, context-dependent names were exported to user actors. These *port descriptors* were inherited in the same fashion as UNIX file descriptors.

2.1. UNIX

On top of this architecture, a full UNIX System V was built.

In V2, the whole of UNIX was split into three servers: a Process Manager, dedicated to process management, a File Manager for block device and file system management, and a Device Manager for character device management. In addition, the nucleus was complemented with two servers, one which managed ports and port groups, and another which managed remote communications (see Figure 1). UNIX network facilities (sockets) were not implemented at this time.

A UNIX process was implemented as a CHORUS actor. All interactions of the process with its environment, i.e. all system calls, were performed as exchanges of messages between the process and system servers. Signals were also implemented as messages.

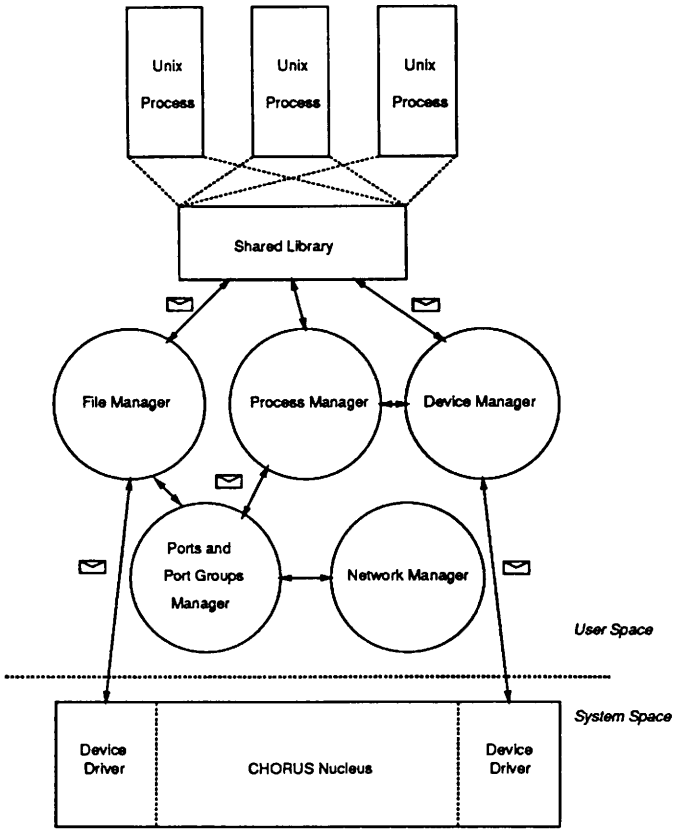


Figure 1: *CHORUS-V2 Architecture*

This "modularisation" impacted UNIX in the following ways:

- UNIX data structures were split between the nucleus and several servers. Splitting the data structures, rather than replicating them, was done to avoid consistency problems. Messages between these servers contained the information managed by one server and required by another in order to provide its service. Careful thought was given to how UNIX data structures were split between servers to minimise communication costs.
- Most UNIX objects, files in particular, were designated by network-wide capabilities which could be exchanged freely between subsystem servers and sites. The context of a process contained a set of capabilities representing the objects accessed by the process.

As many of the UNIX system calls as possible were implemented by a process-level library. The process context was stored in process-specific library data at a fixed, read-only location within the process address space. The library invoked the servers, when necessary, using an RPC facility. For example, the Process Manager was invoked to handle a *fork*(2) system call and the File Manager for a *read*(2) system call on a file.

This library offered only source-level compatibility with UNIX, but was acceptable because binary compatibility was not a project goal. The library resided at a predefined user virtual address in a write-protected area. Library data holding the process context information was not completely secure from malicious or unintentional modification by the user. Thus, errant programs could experience new, unexpected error

behaviour. In addition, programs that depended upon the standard UNIX address space layout could cease to function because of the additional address space contents.

2.2. Extended UNIX Services

CHORUS V2 extended UNIX services in two ways:

- By allowing their distribution while retaining their original interface (e.g. remote process creation and remote file access).
- By providing access to new services without breaking existing UNIX semantics (e.g. CHORUS IPC).

2.2.1. Distribution of UNIX Services

Access to files and processes extended naturally to the remote case due to the modularity of CHORUS's UNIX and its inherent protocols. Files and processes, whether local or remote, were manipulated using CHORUS IPC through the use of location-transparent capabilities.

In addition, CHORUS V2 extended UNIX file semantics with *port nodes*. A port node was an entry in the file system which had a CHORUS port associated with it. When a port node was encountered during pathname analysis, a message containing the remainder of the path to be analysed was sent to the associated port. Port nodes were used to automatically interconnect file trees.

For processes, new protocols between Process Managers were developed in order to distribute *fork* and *exec* operations. Remote *fork* and *exec* were facilitated because:

- The management of a process context was not distributed; each process context was managed entirely by only one system server (the Process Manager),
- A process context contained only global references to resources (capabilities).

Therefore, creating a remote process could be done almost entirely by transferring the process context from one Process Manager to another.

Since signals were implemented as messages, their distribution was trivial due to the location transparency of CHORUS IPC.

2.2.2. Introduction of New Services

CHORUS IPC was introduced at user-level. Its UNIX interface was designed in the standard UNIX style:

- Ports and port groups were known, from within processes, by local identifiers. Access to a port was controlled in a fashion analogous to the access to a file.
- Ports and port groups were protected in a similar fashion to files (with *uids* and *gids*).
- Port and port group access rights were inherited on *fork* and *exec* exactly as are file descriptors.

3. Analysis of CHORUS V2

Experience developing and using CHORUS V2 gave us valuable insight into the basic operating system services that a microkernel must provide to implement a rich operating system environment such as UNIX.

CHORUS V2 was our third reimplementation of the CHORUS nucleus, but represented our first attempt at integrating an existing, complex operating system interface with microkernel technology. This research exercise was not without faults. However, it demonstrated that we did a number of things correctly. The CHORUS V2 basic IPC abstractions – location transparency, untyped messages, asynchronous and RPC protocols, ports, and port groups – have proven to be very well suited to the implementation of distributed operating systems and applications. These abstractions have been entirely retained for CHORUS V3; only their interface has been enriched to make their use more efficient.

The basic modular architecture of the UNIX subsystem has also been retained in the implementation of CHORUS V3 UNIX subsystems. Some new servers, such as a BSD Socket Manager, have been added to provide new function that was not included in CHORUS V2.

Version 3 of the CHORUS nucleus has been completely redesigned and reimplemented around a new set of project goals. These goals were put in place as a direct result of our experience implementing our first distributed UNIX system.

In the following subsections we briefly state our new goals and then explain how these new goals affected the design of CHORUS V3.

3.1. CHORUS V3 Goals

The design of CHORUS V3 system [Arm89a, Arm90a, Her88a, Roz88a] has been strongly influenced by a new major goal: to design a microkernel technology suitable for the implementation of commercial operating systems. CHORUS V2 was a UNIX-compatible distributed operating system. The CHORUS V3 microkernel is able to support operating system standards while meeting the new needs of commercial systems builders.

These new goals determined new guidelines for the design of the CHORUS V3 technology:

- **Portability:** the CHORUS V3 microkernel must be highly portable to many machine architectures. In particular, this guideline motivated the design of an architecture-independent memory management system [Abr89a], taking the place of the hardware-specific CHORUS V2 memory management.
- **Generality:** the CHORUS V3 microkernel must provide a set of functions that are sufficiently generic to allow the implementation of many different sets of operating system semantics; some UNIX-related features had to be removed from the CHORUS V2 nucleus. The nucleus must maintain its simplicity and efficiency for users or subsystems which do not require high level services.
- **Compatibility:** UNIX source compatibility in CHORUS V2 had to be extended to binary compatibility in V3, both for user applications and device drivers. In particular, the CHORUS V3 nucleus had to provide tools to allow subsystems to build binary compatible interfaces.
- **Real-time:** process control and telecommunication systems comprise important targets for distributed systems. In this area, the responsiveness of the system is of prime importance. The CHORUS V3 nucleus is, first and foremost, a distributed real-time executive. The real-time features may be used by any subsystem, allowing for example, a UNIX subsystem to be naturally extended to be suitable for real-time applications needs.

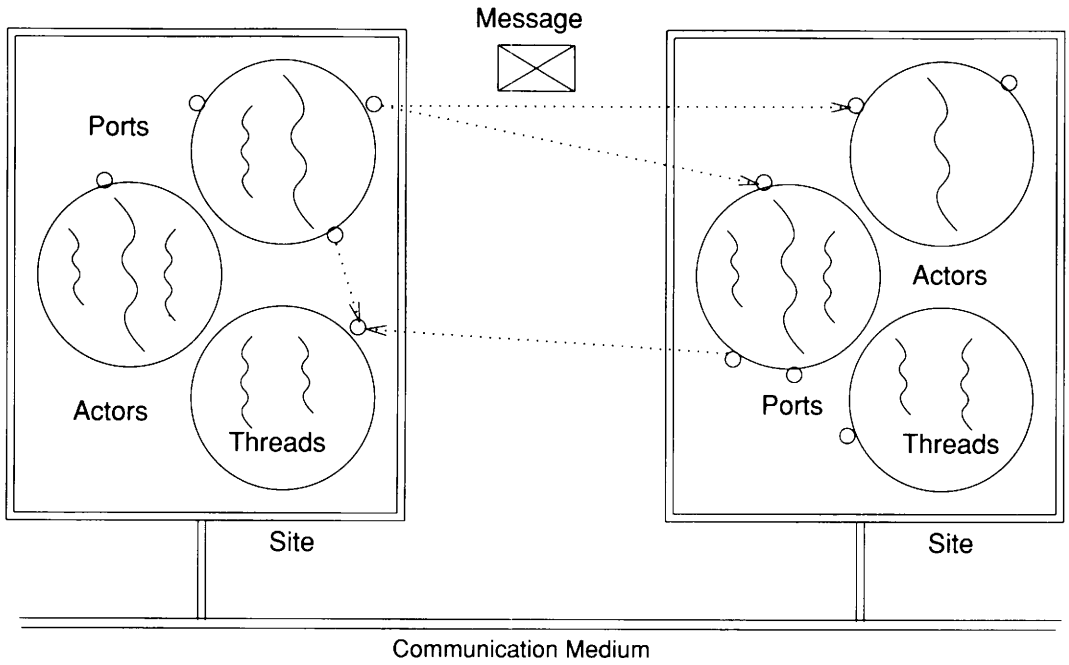


Figure 2: *CHORUS V3 Nucleus Abstractions*

- **Performance:** for commercial viability, good performance is essential in an operating system. While offering the base for building modular, well-structured operating systems, the nucleus interface must allow these operating systems to reach at least the same performance as conventional, monolithic, implementations.

These new goals forced us to reconsider CHORUS V2 design choices. In most cases, the architectural elements were retained in CHORUS V3; only their interface evolved. Whenever possible, the V3 interface reflects our desire to leave it to the subsystem designer to negotiate the tradeoffs between simplicity and efficiency, on the one hand, and more sophisticated function, on the other.

3.2. CHORUS Processing Model

Problems arose with the CHORUS V2 processing model when UNIX signals were first implemented. To treat asynchronous signals in V2 mono-threaded actors, it was necessary to introduce the concept of priorities within messages to expedite the invocation of a signaling operation. Even so, the priorities went into effect only at fixed synchronisation points, making it impossible to perfectly emulate UNIX signal behaviour. Further work has shown that signals are one of the major stumbling blocks for building fault tolerant UNIX systems.

> **Lesson:** *We found the processing-step model of computation to be a poor fit with the asynchronous signal model of exception handling. In order to provide full UNIX emulation, a more general computational model was necessary for CHORUS V3.*

The solution to this problem gave rise to the V3 multi-threaded processing model. A CHORUS V3 actor is merely a resource container, offering, in particular, an address space in which multiple threads may execute. Threads are scheduled as independent entities, allowing real parallelism on a multiprocessor architecture. In addition, multiple

threads allow the simplification of the control structure of server-based applications. New nucleus services, such as thread execution control and synchronisation have been introduced.

3.3. CHORUS Inter-Process Communication

As a consequence of the change to the basic processing model, the inter-process communication model also evolved. In the V2 processing-step model, IPC and execution were tightly bound, yielding a mechanism that resembled atomic transactions.

This tight binding of communication to execution did not necessarily make sense in a multi-threaded CHORUS V3 system. Thus, the atomic transactions of V2 have been replaced, in V3, by the remote procedure call (RPC) paradigm and has since evolved into a very efficient light-weight RPC protocol.

One aspect of the IPC mechanism that has not changed in CHORUS V3 is that messages remain untyped. The CHORUS IPC mechanism is simple and efficient when communicating among homogeneous sites. When communicating between heterogeneous sites, higher-level protocols are used, as needed. A guideline in the design of CHORUS V2, retained in V3, was to allow the construction of simple and efficient applications without forcing them to pay a penalty for sophisticated mechanisms which were required only by specific classes of programs.

3.4. CHORUS Ports

A number of enhancements concerning CHORUS ports have been made to provide more generality and efficiency in the most common cases.

3.4.1. Port Naming

Recall that in V2 context-dependent port names were exported to the user-level while global port names were used by the nucleus and system servers. The user-level context-dependent port names of V2 were intended to provide security and ease of use. It was difficult, however, for applications to exchange port names, since it required intervention by the nucleus and posed bootstrapping problems. As a result, context-dependent names were inconvenient for distributed applications, such as name servers. In addition, many applications had no need of the added security the context-dependent names provided.

> **Lesson:** *CHORUS V3 makes global names of ports and port groups (Unique Identifiers) visible to the user, discarding the UNIX-like CHORUS V2 contextual naming scheme. Contextual identifiers turned out not to be an effective paradigm.*

The first consequence of using Unique Identifiers is simplicity: port and port group names may be freely exchanged by nucleus users, avoiding the need for the nucleus to maintain complex actor context. The second consequence is a lower level of protection: the CHORUS V3 philosophy is to provide subsystems with the means for implementing their own level and style of protection rather than enforcing protection directly in the microkernel. For example, if the security of V2 context-dependent names is desired, a subsystem can easily and efficiently export a protected name-space server. V3 Unique Identifiers have proven to be key to providing distributed UNIX services in an efficient manner.

3.4.2. Port Implementation

A goal of the V2 project was to determine what were the minimal set of functions that a microkernel should have in order to support a robust base of computing. To that end, the management of ports and port groups was put into a server external to the nucleus. Providing the ability to replace a portion of the IPC did not prove to be useful, however, since IPC was a fundamental and critical element of all nucleus operations. Maintaining it in a separate server rendered it more expensive to use.

> **Lesson:** *We found that actors, ports, and port groups are basic nucleus abstractions. Splitting their management did not provide significant benefit, but did impact system performance. Actor, port, and port group management has been moved back into the nucleus for V3.*

3.4.3. UNIX Port Extensions

When extending the UNIX interface to give access to CHORUS IPC, we maintained normal UNIX-style semantics. Employing the same form as the UNIX file descriptor for port descriptors was intended to provide uniformity of model. The semantics of ports were sufficiently different from the semantics of files to negate this advantage. In operations such as *fork*, for example, it did not make sense to share port descriptors in the same fashion as file descriptors. Attempting to force ports into the UNIX model resulted in confusion.

> **Lesson:** *A user-level IPC interface was important, but giving it UNIX semantics was cumbersome and unnecessary. This lesson is an example of a larger principle; the nucleus abstractions should be primitive and generally applicable – they should not be coerced into the framework of a specific operating system.*

V3 avoids this issue by, as previously mentioned, exporting global names. Since the V3 nucleus no longer manages the sharing of global port and port group names, it is up to the individual subsystem servers to do so. In particular, if counting the number of references to a given port is important to a subsystem, it is the subsystem itself that must maintain the reference count. On the other hand, a subsystem that has no need for reference counting is not penalised by the nucleus.

Using V2 port nodes to interconnect file systems was a simple, but extremely powerful, extension to UNIX. Since all access to files was via CHORUS messages, port nodes provided network transparent access to regular files as well as to device nodes. They also, however, introduced a new file type into the file system. This caused many system utilities, such as *ls* and *find*, to not function properly. Thus, all such utilities had to be modified to take the new file type into account.

Port nodes have been maintained in CHORUS V3 (however, they are now called *"symbolic ports"*). *In future CHORUS UNIX systems, the file type "symbolic port" may be eliminated by inserting the port into the file system hierarchy using the mount system call. These "port mount points" would carry the same semantics as a normal mounted file system.*

3.5. Virtual Memory

The virtual memory subsystem has undergone significant change. The machine dependent virtual memory system of CHORUS V2 has been replaced, in V3, by highly portable VM system. The VM abstractions presented by the V3 nucleus include "segments" and "regions." Segments encapsulate data within a CHORUS system and typically represent some form of backing store, such as a swap area on a disk. A region is a contiguous range of virtual addresses within an actor that map a portion of a segment into its address space. Requests to read or to modify data within a region are converted by the virtual memory system into read or modify requests within the segment. "External Mappers" interact with the virtual memory system using a nucleus-to-Mapper protocol to manage data represented by segments. Mappers also provide the needed synchronisation to implement distributed shared memory. For more details on the CHORUS V3 virtual memory system, see [Abr89a].

3.6. Actor Context

CHORUS V2 was built around a "pure" message-passing model, in which strict protection was incorporated at the lowest level; all servers were implemented in protected user address spaces. This distinct separation enforced a clean, modular design of a subsystem. However, it also led to several problems:

- A UNIX subsystem based on CHORUS V2 required the use of user-level system call stubs and altered the memory layout of a process and, therefore, could never provide 100% binary compatibility;
- All device drivers were required to reside within the nucleus;
- Context switching expense was prohibitively high.

The most fundamental enhancement made between CHORUS V2 and V3 was the introduction of the *Supervisor Actor*. Supervisor actors share the supervisor address space and whose threads execute in a privileged machine state. Although they reside within the supervisor address space, supervisor actors are truly separate entities; they are compiled, link edited, and loaded independently of the nucleus and of each other.

The introduction of supervisor actors creates several opportunities for system enhancement in the areas of compatibility and performance. Section 4 discusses the ramifications of supervisor actors in-depth.

3.7. UNIX Subsystem

As a consequence of these nucleus evolutions, the UNIX subsystem implementation has also evolved. In particular, full UNIX binary compatibility has been achieved. Internally, the UNIX subsystem makes use of new nucleus services, such as multi-threading and supervisor actors. The CHORUS V2 user-level UNIX system-call library has been moved inside the Process Manager and is now invoked directly by system-call traps.

Experience with the decomposition of UNIX System V for V2 showed, not surprisingly, that performing this modularisation is difficult. Care must be taken to decompose the data structures and function along meaningful boundaries. Performing this decomposition is an iterative process. The system is first decomposed along broad functional lines.

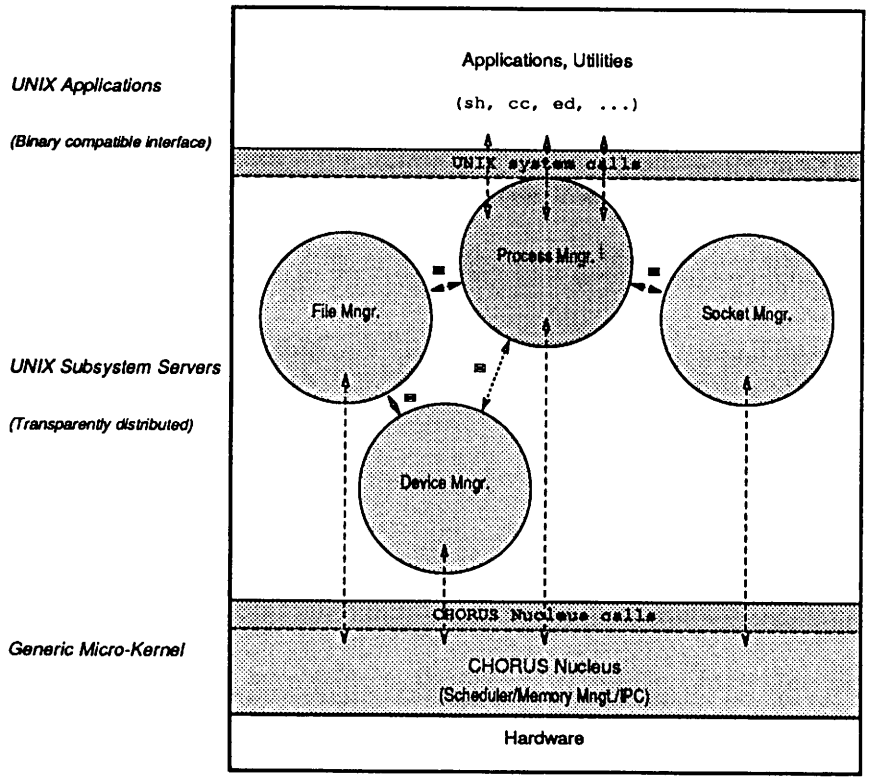


Figure 3: *CHORUS/MiX-V3 Architecture*

The data structures are then split accordingly, possibly impacting the functional decomposition.

4. Evolution in Nucleus Support for Subsystems: Supervisor Actors

Supervisor actors, as mentioned above, share the supervisor address space and whose threads execute in a privileged machine state, usually implying, among other things, the ability to execute privileged instructions. Otherwise, supervisor actors are fundamentally very similar to regular user actors. They may create multiple ports and threads, and their threads access the same nucleus interface. Any user program can be run as a supervisor actor, and any supervisor actor which does not make use of privileged instructions or *connected handlers* (see below) can be run as a user actor. In both cases a recompilation of the program is not needed (a relink is required, however). Although they share the supervisor address space, supervisor actors are paged just as user actors and may be dynamically loaded and deleted.

Supervisor actors alone are granted direct access to the hardware event facilities. Using a standard nucleus interface, any supervisor actor may dynamically establish a handler for any particular hardware interrupt, system call trap, or program exception. A connected handler executes as an ordinary subroutine, called directly from the corresponding low-level handler in the nucleus. Several arguments are passed, including the interrupt/trap/exception number and the processor context of the

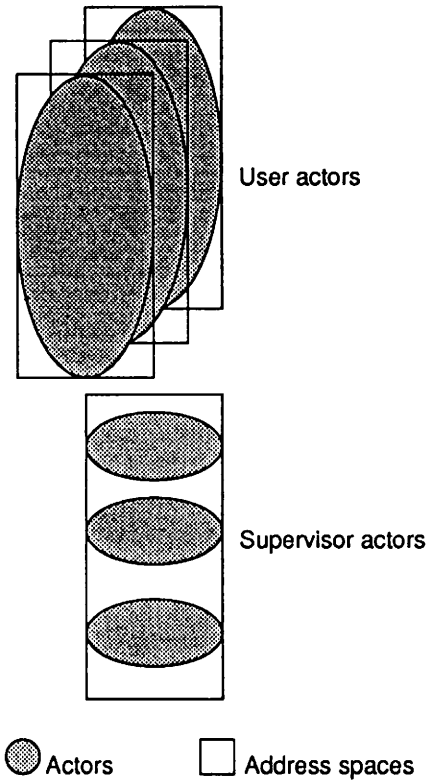


Figure 4: *Supervisor Actors*

executing thread. The handler routine may take various actions, such as processing an event and/or awakening a regular thread in the actor. The handler routine then returns to the nucleus.

4.1. External Device Drivers

It is important to note that no subsystem in CHORUS V3 is ever *required* to use connected handlers or supervisor actors. For example, a subsystem designer may choose to export a programming interface based entirely upon messages rather than upon traps. The CHORUS nucleus can handle program exceptions either by sending an RPC message to a designated exception port or by calling a connected exception handler. Only actors that process device interrupts are required to be implemented as supervisor actors. Even so, device drivers may be split into two parts, if desired; a "stub" supervisor actor to translate interrupts into messages and a user-mode actor that processes these interrupt messages. Connected handlers, however, provide significant advantages in both performance and binary compatibility:

- The nucleus need not be modified each time that a new device type is to be supported on a given machine;
- Interrupt processing time is greatly reduced, allowing real-time applications to be implemented outside of the nucleus.

Connected interrupt handlers allow device drivers to exist entirely outside of the nucleus, and to be dynamically loaded and deleted, with no loss in interrupt response or overall performance. For example, to demonstrate the power and flexibility of the CHORUS V3 nucleus, we

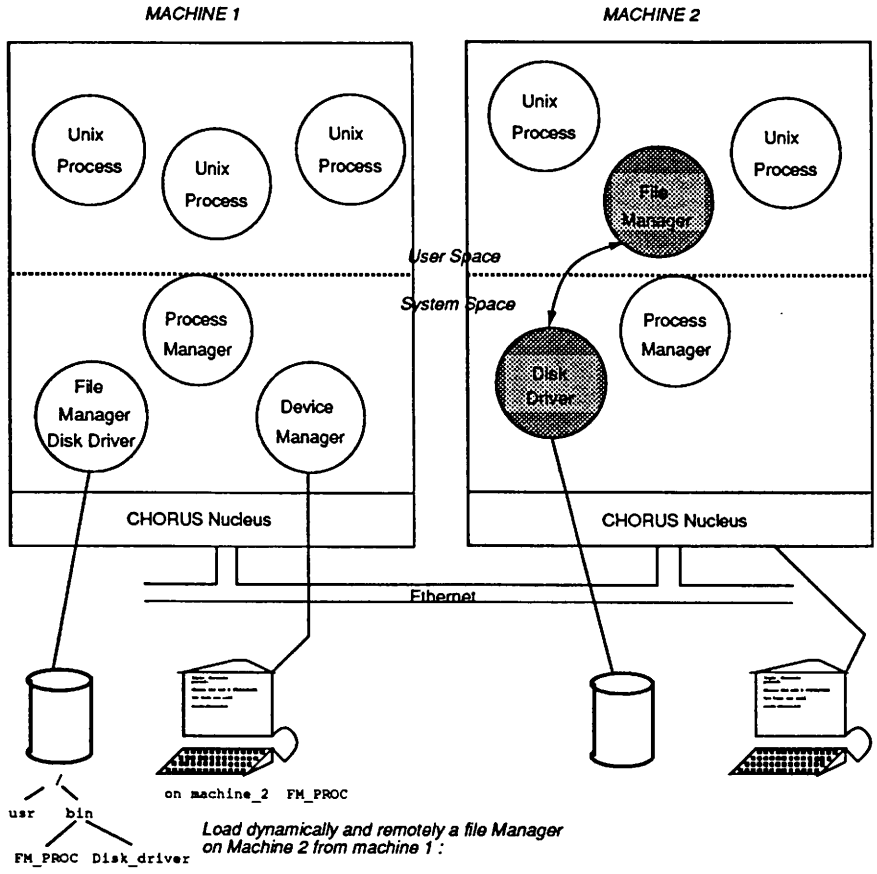


Figure 5: *CHORUS/MiX File Manager as a Unix Process in User Space*

have constructed a user-mode File Manager that communicates using CHORUS IPC with a supervisor actor which manages a physical disk. Both the supervisor actor and the user-mode File Manager can be dynamically loaded from a remote site. Additionally, the user-mode File Manager can be debugged using standard debuggers.

Interrupt handlers may be stacked, since multiple device types often share a single interrupt level. In this case the sequence of handlers is executed in priority order until one of them returns a code indicating that no further handlers should be called. Connected interrupt handlers have been designed to allow subsystems to incorporate proprietary, object-only device drivers that conform to one of the relevant binary standards that are emerging in this area. Without this mechanism, object compatibility would require incorporating entire device drivers within the nucleus.

4.2. Compatibility

System call trap handlers are essential for both performance and, as it has been pointed out in [Tan90a], binary compatibility. Any subsystem may dynamically connect either a general trap-handling routine or a table of specific system call handlers, the latter providing an optimised path for UNIX-style interfaces. An alternative mechanism, the system-wide user-level shared library used in CHORUS V2, would seem to provide equivalent system call performance. However, we found that it is

difficult to protect subsystem data that share the address space of the user program, especially if processes are multi-threaded. As we have seen, malicious or innocent but erroneous programs can change the behaviour of system calls. If functions must be moved from the shared library into separate servers for protection, increased IPC traffic results. Finally, the presence of the library code and data in the user context can interfere with binary programs that use a large portion of the address space or manage the address space in some particular fashion. Traps to supervisor actors, by contrast, provide a low-overhead, self-authenticating transfer to a protected server, while maintaining full transparency for the user program.

> **Lesson:** *Use of shared libraries produces compatibility and error-detection problems. For 100% UNIX binary compatibility, it is necessary to maintain the standard UNIX trap interface and address space layout.*

4.3. Performance Benefits

Performance benefits of supervisor actors come in several areas. Memory and processor context switches are minimised through use of connected handlers rather than messages, and in general through address-space sharing of actors of a common subsystem which happen to be running on a single site. Trap expense can be avoided for nucleus system calls executed by supervisor actors. Finally, supervisor actors allow a new level of RPC efficiency. The "lightweight RPC" mechanism of [Ber90a] optimises pure RPC for the case where client and server reside on the same site. We further optimise for the case where no protection barrier need be crossed between client and server. This "featherweight" RPC is substantially lower in overhead, while still mediated by the nucleus and still using an interface similar to that of pure RPC.

> **Lesson:** *Implementing part of an operating system in user-level servers, while elegant, imposes prohibitive message passing and context switching overheads not present in a monolithic implementation of the system. To allow microkernel technology to compete in the marketplace, it was necessary to provide a solution to these problems. Supervisor actors provide the advantages of a modular system while minimally sacrificing performance.*

4.4. Construction of Subsystems

Subsystems may be constructed using combinations of supervisor or user actors. Any server may itself belong to a subsystem, such as UNIX, as long as it does not produce any infinite recursions, and may be either local or remote. Servers that need to issue privileged instructions or that are responsible for handling traps or interrupts must be supervisor actors.

4.5. Protection Issues

Computer systems often give rise to tradeoffs between security and performance, and we must consider the nature of the sacrifice being made when multiple servers and the microkernel share the supervisor address space. Protection barriers are weakened, but only among mutually-trusted system servers. The ramifications of the weakening of protection barriers can be minimised by systematically adhering to the

following design rule: individual servers must never pass data through shared memory.

Allowing a server to explicitly access other servers' data would completely break system modularity. This rule being enforced, the only genuine sacrifice for using supervisor actors is a degree of bug isolation among the components of a running system. This is somewhat mitigated by the fact that subsystem servers may be debugged in user mode. In fact, this forms our day-to-day development activity: servers are developed and debugged in user mode. When validated, they are loaded as supervisor actors for better performance, if desired. However, the overall CHORUS philosophy is to allow the subsystem designer or even a system manager to choose between protection and performance on a case-by-case basis, and to alter those choices easily.

5. Evolution in CHORUS IPC

CHORUS V3 IPC is based on the accumulated experience gained since CHORUS V0. Here again, the main characteristics of the IPC facilities are their simplicity and performance.

5.1. Naming

The first aspect which has evolved since V2 is naming: for many reasons, distributed applications need to transfer names among their individual components. This is most efficiently achieved with a single space of global names that are usable in any context, from nucleus to application level. The main difficulty with this style of naming is protection.

In CHORUS V3, ports and port groups are named using Unique Identifiers which are visible at every level. Basic protection for these names is threefold:

- All messages are stamped by the nucleus with the sending port's Unique Identifier as well as its *Protection Identifier*. Protection Identifiers allow the source of a message to be reliably identified as they may be modified only by trusted actors. Using these facilities provided by the nucleus, subsystems have the choice to implement their own more stringent user authentication mechanisms if needed.
- Global names are randomly generated in a large, sparse name space; knowing a valid global name does not help much in finding other valid names.
- Objects within CHORUS may be named using capabilities which consist of a <name, key> tuple. Capabilities are constructed using whatever techniques are deemed appropriate by the server that provides them, and may incorporate protection schemes.

Port groups, as implemented by the nucleus, have keys related to the group name by means of a non-invertible function. Knowledge of the group name conveys the right to send messages to the group, but knowledge of the key is required to insert or delete members from the group.

Higher degrees of port and/or message security can be implemented by individual subsystems, as required. Subsystems may act as intermediaries in message communications to provide protection, or may choose

to completely exclude CHORUS IPC from the set of abstractions they export to user tasks.

5.2. Message Structure

A second area of evolution in the CHORUS V3 IPC is message structure.

The memory management units of most modern machines allow moving data from the address space of one actor to the address space of another actor by remapping. This facility is exploited in CHORUS V3 IPC, which allows transmission of message bodies between actors within a single site by means of address remapping. In situations where data is to be copied and not moved between address spaces, CHORUS V3 has copy-on-write facilities that allow the data to be efficiently transferred only as needed. The typical communication that makes use of this facility involves the exchange of a large amount of data (e.g. I/O operations).

It is often the case that messages contain a large data area, accompanied by some auxiliary information such as a header or some parameters, such as a path-name, a size, or the result of an I/O operation. Frequently, the auxiliary information is physically disjoint from the primary data. In CHORUS V2, assembling these two discontiguous fragments into a single message required that extra copying be done by the user.

CHORUS V3 splits message data into two parts:

- A message body, which has a variable size and may be copied or moved; it typically contains the raw data;
- The message annex, which has a fixed size and is always copied; it typically contains the associated parameters or headers. This division also allows one software layer to provide data, while another provides header or parameter information. For example, the V3 implementation of the *write* system call receives the address of a data buffer from the caller and appends a header describing the data area and sends both to the device responsible for performing the operation.

5.3. Processing vs. Communication

A third issue is the relationship between the processing model and communication model. The CHORUS V2 execution model was event or communication-driven. In CHORUS V3, the processing model has been inverted – actors are multi-threaded and the basic mechanism for inter-process synchronisation is RPC. Thus, the CHORUS V3 model is much closer to the traditional procedural model of computation. Multi-threading allows the multiplexing of servers, simplifying their control structure while potentially increasing concurrency and parallelism. RPC is well understood and straightforward to program.

In addition, for applications that require basic, low-level communication, asynchronous IPC is provided. This IPC has very simple semantics – it provides unidirectional communication incorporating location transparency, with no error detection or flow control. Higher-level protocol layers provided by the user or subsystem can be built on top of this minimal nucleus function.

6. Conclusion

With CHORUS V2, we experimented with a first-generation microkernel-based UNIX system. UNIX emulation was built as an application of a pure message-based microkernel. Our microkernel approach proved its applicability to building UNIX operating systems for distributed architecture in a research environment.

The challenge in designing CHORUS V3 was to make this technology suitable for commercial systems requirements; to provide performance comparable to similar monolithic systems and to provide full compatibility with these systems. Our second-generation microkernel design was driven by these requirements and we were forced to reconsider the role of the microkernel. Instead of strictly enforcing a single, rigid, system architecture, the microkernel is now comprised of a set of basic, flexible, and versatile tools. Our experience with CHORUS V2 taught us that some functions, such as IPC management, belong within the microkernel. Device drivers and support for heterogeneity, on the other hand, are best handled by separate servers and protocols. Supervisor actors are crucial to both performance and binary compatibility with existing systems. A global name space is necessary to simplify the interactions between system servers and the nucleus. Using CHORUS V3, subsystem designers have the freedom to define their operating system architecture and to select the most appropriate tools. Decisions, such as the choice between high security and high performance, are not be enforced a priori by the microkernel.

The CHORUS V3 microkernel has met its requirements: the CHORUS/MiX microkernel-based UNIX system provides the level of performance of real-time executives, is compatible with UNIX at the binary level, and is truly modular and fully distributed. It has been adopted by a number of manufacturers for real-time and distributed commercial UNIX systems.

Further work will concentrate on exploiting this technology to provide advanced operating system features, such as a distributed UNIX with a single system image and fault tolerance.

7. Authors' Biographies

Allan Bricker <allan@chorus.fr> is a Senior Engineer at Chorus systèmes. In 1985 he received his Master of Science Degree in Computer Science from the University of Wisconsin-Madison. From 1984 to 1989 he was the senior researcher in charge of the design and development of the multi-threaded communications kernel for the Gamma parallel database machine at the University of Wisconsin.

Michel Gien <mg@chorus.fr> is a co-founder, general manager, and director of R&D at Chorus systèmes. He joined the Cyclades computer network team at INRIA in 1971 after graduating from Ecole Centrale des Arts et Manufactures de Paris. He then led a project that introduced UNIX in France and helped to understand how it could be redesigned along the CHORUS distributed systems concepts. Michel is currently serving as the chair of EurOpen (formerly EUUG) after having served as its vice chair since 1985.

Marc Guillemont <mgu@chorus.fr> is a co-founder and the director of engineering at Chorus systèmes. He joined INRIA in 1977 to work on the Cyclades project and was also member of the initial CHORUS

research project team in 1980 before he became head of the team. He managed the final research phases of CHORUS before developing the commercial version. Marc Guillemont graduated from Ecole Polytechnique in 1971 and earned a PhD in Computer Science from Grenoble University.

Jim Lipkis <lipkis@chorus.fr> is a Senior Engineer at Chorus systèmes. At New York University, he was responsible for design and development of operating system and programming language software for highly parallel shared-memory multi-processors, including the NYU Ultracomputer.

Douglas Orr <doug@chorus.fr> is a Senior Engineer at Chorus systèmes. He graduated from the University of Michigan and has worked for Apollo Computer and Carnegie Mellon University. His interests include operating systems, computer networks, and existentialism.

Marc Rozier <mr@chorus.fr> is the head of the CHORUS distributed microkernel development team within Chorus systèmes. He graduated from ENSIMAG before earning a PhD in Computer Science from the University of Grenoble. He joined INRIA in 1982 as a researcher in the CHORUS distributed operating system project. He worked on both the design and implementation of two versions of CHORUS. In 1987, he became one of the founders of Chorus systèmes.

References

[Abr89a] V. Abrossimov, M. Rozier, and M. Shapiro, "Generic Virtual Memory Management for Operating System Kernels," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ (USA) (December 1989).

[Arm86a] François Armand, Michel Gien, Marc Guillemont, and Pierre Léonard, "Towards a Distributed UNIX System – The CHORUS Approach," in *Proceedings of the EUUG Autumn'86 Conference*, Manchester, England (Autumn 1986).

[Arm89a] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier, "Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues"," in *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL (USA) (October 1989).

[Arm90a] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier, "Multi-threaded Processes in CHORUS/MiX," pp. 1-13 in *Proceedings of the EUUG Spring'90 Conference*, Munich, Germany (April 1990).

[Ber90a] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systems* **8**(1), pp. 37-55 (February 1990).

[Che90a] David R. Cheriton, Gregory R. Whitehead, and Edward W. Sznyter, "Binary Emulation of UNIX using the V Kernel," pp. 73-86 in *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA (USA) (June 1990).

[Gol90a] Davic Golub, Randall Dean, Alessandro Forin, and Richard Rashid, "UNIX as an Application Program," pp.

87-96 in *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA (USA) (June 1990).

[Her88a] Frédéric Herrmann, François Armand, Marc Rozier, Vadim Abrossimov, Ivan Boule, Michel Gien, Marc Guillemont, Pierre Léonard, Sylvain Langlois, and Will Neuhauser, "CHORUS, a New Technology for Building UNIX Systems," pp. 1-18 in *Proceedings of the EUUG Autumn'88 Conference*, Cascais, Portugal (October 1988).

[Pik91a] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, "Designing Plan 9," *Dr. Dobb's Journal* **16**(1), pp. 49-60 (January 1991).

[Roz87a] Marc Rozier and José Legatheaux-Martins, "The CHORUS Distributed Operating System: Some Design Issues," pp. 261-286 in *Distributed Operating Systems, Theory and Practice*, Springer-Verlag, Berlin, BRD (1987).

[Roz88a] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser, "CHORUS Distributed Operating Systems," *Computing Systems Journal* **1**(4), pp. 305-370, The Usenix Association (December 1988).

[Tan90a] Andrew Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum, "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM* **33**(12), pp. 46-63 (December 1990).

The OSF/1 Operating System

Open Software Foundation

Abstract

The OSF/1 operating system fulfills the requirements of open systems for both compatibility and innovation. It is compatible with the most widely used systems and supports all relevant industry standards. A wide array of innovative features, such as fully kernel supported symmetric multiprocessing, enhanced security, and dynamic system configuration, complete the picture.

The design of OSF/1 represents a return to the roots of the UNIX operating system. This system is based on Mach technology and in order to increase maintainability, extensibility and flexibility, is evolving into a micro-kernel architecture, where services are moved into user space. This modular approach provides a open strategy for future developments in operating system technology, particularly well adapted to distributed systems.

1. Introduction

Operating systems for open computing environments require a balance between innovative features and compatibility with current operating system implementations and standards.

The UNIX operating system, as it was originally developed in the late 1960s and early 1970s, provided a simple set of basic operating system abstractions which could be implemented on a wide range of hardware architectures. However, over the last 20 years, changing needs and technologies have fueled a dramatic expansion of UNIX based operating systems. They have become progressively larger, more complex, more difficult to understand, and more difficult to extend.

The design of OSF/1 represents a return to the roots of the UNIX operating system. Its architecture is based on a modular kernel, and its strategy is to move many kernel services into user space to increase system maintainability, extensibility, and flexibility. This strategy provides users of the OSF/1 operating system with a clear growth path toward future developments in operating system technology, particularly distributed systems.

On the other hand, it provides significant innovations to support advanced applications as well as the compatibility features required to run existing applications. Full compliance with standards and industry specifications, including POSIX and XPG3, and compatibility with UNIX System V and Berkeley programming interfaces, ensures application portability.

OSF/1 integrates in-house developments with proven technologies from a number of sources to meet the needs of commercial, scientific, and government users worldwide. These sources include Carnegie Mellon University, Encore Computer Corporation, IBM, Mentat, SecureWare, and the University of California.

The result is a complete, innovative open systems platform featuring:

- A redesigned operating system core that reduces the costs associated with system maintenance and modification and provides a robust basis for functional evolution
- Symmetric multiprocessing capability, for achieving maximum performance from multiprocessor hardware
- Commercial processing capability such as logical volume management, disk mirroring, and dynamic system configuration for increased system availability and flexibility
- Enhanced security functionality to protect sensitive data and installations
- Compatibility features and adherence to industry standards and specifications to protect investments in software.

The OSF/1 system presents a clear choice to applications developers. They can invest in current operating systems technology, that will need a redesign to meet future computing requirements, or invest in new OSF/1 systems, which will meet those requirements by design.

OSF/1 makes this choice easy. Through compatibility with existing operating system implementations, OSF/1 brings the best of the past forward. Its innovations provide clear advantages for today, and its design embodies emerging computing concepts for protection of investments well into the future.

2. The OSF/1 Kernel

An advanced kernel architecture based on Mach technology forms the foundation for OSF/1. Mach was developed at Carnegie Mellon University and refined over the past six years by commercial suppliers, universities and industrial research centers. Today, thousands of Mach-based systems have been shipped by several commercial suppliers.

The OSF/1 kernel architecture provides:

- A highly efficient kernel built upon five fundamental programming abstractions (task, thread, port, message, and memory object) providing a simpler, more manageable, portable system
- Fine grained kernel parallelism to support parallelized file systems and networking, while retaining the ability to use non-parallelized subsystems
- POSIX compliant threads, allowing multiple instruction streams to run concurrently in a single address space
- A highly portable and efficient virtual memory system, providing optimized performance for today's large applications and easier porting to different platforms.

Features and Benefits

Reduced complexity – The Mach component of the OSF/1 kernel is compact and modular. Mach provides the OSF/1 system with the basic kernel services of scheduling, memory management, and interprocess communication. Subsystems are layered on top of it in a modular way to provide other system services, such as managing the file system. The result is an easier to understand, flexible, more manageable, and more portable system.

Inherent support for multiprocessing – Multiprocessor computer systems enable users to match computing power to application needs by adding processors to their system, rather than purchasing additional systems. Because applications spend a significant portion of their time using kernel services, a kernel specifically designed for multiprocessing is important for overall system throughput. The Mach kernel at the core of OSF/1 was designed from the outset for multiprocessing. Non-multiprocessing applications can run unmodified on OSF/1, giving end users great flexibility in their choice of hardware.

Support for external memory managers – The OSF/1 virtual memory system enables developers to create external memory managers. An external memory manager is a user level task which cooperates with the kernel to manage the way data is moved between main memory and external storage. This is especially useful for applications that must maintain fine control of memory management, such as transaction processing systems.

User space "kernel" services – Over time, traditional UNIX systems have suffered from the accretion of new features into the operating system kernel. What started as a relatively small and compact operating system has grown progressively larger. Today's UNIX systems are large, complex, and difficult to maintain and extend. The OSF/1 system, and its Mach foundation, represents a return to the original concept of a small and compact operating system. The movement of relevant kernel services into user address space is part of OSF's long range architectural strategy for OSF/1. This will make the system easier to maintain and extend and give software developers more control over the use of its facilities.

OSF/1 subsystem parallelization

Key layered subsystems have been fully parallelized for concurrent operation on multiprocessor hardware. Subsystem parallelization further increases the performance on multiprocessor hardware of applications using these subsystems. System performance, and overall job throughput, will be higher with the parallelized OSF/1 operating system than with other, unparallelized multiprocessing implementations.

OSF/1 File Systems

The OSF/1 file system provides a path to several file system types through its Virtual File System (VFS) interface. OSF's implementation of the VFS interface is derived from the 4.4BSD (Berkeley Software Distribution) operating system. The VFS interface enables multiple types of file systems to be used transparently.

For compatibility, OSF/1 applications can reach through the VFS interface to the UNIX file system (UFS), which is compatible with the 4.3BSD Tahoe release; a System V file system; and an NFS compatible distributed file system.

The VFS, UFS, 4.3BSD, and NFS-compatible file systems have been fully parallelized for maximum performance on multiprocessor hardware.

3. Program Development Features

The OSF/1 system features significant innovations to support advanced applications as well as the compatibility features required to run existing applications. Compliance with standards and industry specifications, including POSIX and XPG3, and compatibility with UNIX System V and Berkeley programming interfaces ensure application portability.

In addition to the standard UNIX development tools and interfaces, OSF/1 includes other, powerful programming features. These features include an extensible program loader; shared libraries; support for relocatable code; dynamic loading of relocatable program modules; dynamic system configuration; external pagers for memory management; memory mapped files; and fully kernel supported threads.

These features enable software developers to take advantage of the latest developments in hardware, especially multiprocessor systems, and to provide solutions to user problems in new ways. Other features enable developers to create internationalized applications, and to incorporate OSF/1 security functionality to create trusted applications.

OSF/1 Program Loader

The OSF/1 program loader resides in user space, maintaining the simplicity of the operating system's modular kernel while supporting enhanced program loading functions. A major advantage of its design over traditional systems is the ability for user level applications to use loader services without modification of the kernel.

Features and Benefits

Efficient use of memory resources – The OSF/1 program loader supports position independent shared libraries for most efficient use of system memory. With position independent shared libraries, each application no longer needs to contain its own, private copy of the called routines in the library. The loader can dynamically place the shared library at a location in virtual memory where it can be used by any application requiring its routines.

Flexibility in the design and modification of applications – Because applications do not require relinking after modification to a shared library, all applications automatically benefit from any enhancements to the code of the shared library. In addition, user level programs can use the OSF/1 loader to call in program modules and unload them as needed. This facility is particularly useful for larger applications facing memory constraints.

Extensibility – The OSF/1 program loader was designed to handle multiple object module formats. Its clean separation of format independent and format dependent parts, as well as the separation of machine independent and machine dependent code, makes extending the loader for new formats a straightforward process.

Threads

Threads allow developers to write applications which have essentially cooperating routines, or cooperating threads, all sharing access to the same data in memory. The implementation of threads in OSF/1 also makes it easy for applications to take advantage of the power of multiprocessor hardware as well as uniprocessor machines. The threads programming interface in OSF/1 complies with the POSIX 1003.4a draft standard.

Features and Benefits

Enhanced programming model – The use of threads, instead of traditional UNIX facilities, enables easier and more natural solutions for many types of problems – especially problems requiring the management of asynchronous events. Developers can write applications so that each thread has a small, self-contained job. An example is an appointment manager in which all threads share the same in-memory database. Each thread, operating independently of other threads, could process one request for service, and respond at a higher rate than a traditional UNIX application using multiple processes.

Increased application performance – Multiple threads in an application automatically take advantage of additional processors in a multiprocessor system. Even on a single- processor system, threads can increase performance by providing an easy way to overlap computation and input/output.

Ease of development – Developers can write and run multiprocessor applications on a uniprocessor machine. The same application can be recompiled and run on a multiprocessor machine and automatically take advantage of the increased power.

Memory Mapping

Memory mapping gives developers a simple, efficient method for obtaining data from secondary storage. No library code or system calls are involved in file access, and the system retrieves only the particular page of data that was called for by an application, avoiding unnecessary input/output operations. The result is simplified coding of applications and increased throughput.

4. Internationalization

The internationalization and localization capabilities of the OSF/1 system enable developers to meet the linguistic and cultural needs of many countries without altering applications for each country. These capabilities will enable most users to process data and interact with the computer in their own language.

Features and Benefits

Eight-bit clean commands – Enables the system to process data in European and other alphabetic languages.

Collating sequences for European alphabets – Enables the correct sorting of data in non-English languages.

Character classification functions – Provides a means to classify (for example, upper or lower case) non-English characters.

Message catalogs – Enables the display of error messages in various languages.

International date, time, and monetary formatting, and numeric conventions – Enables the further tailoring of applications to match international and local requirements.

OSF/1 internationalization features conform with the XPG3 and POSIX specifications.

5. Networking

OSF/1 networking facilities provide compatibility with current applications and the ability to take advantage of the power of multiprocessor systems.

Features and Benefits

Compatibility – Current applications using the System V Interface Definition (SVID) Issue 2-based STREAMS, or 4.3 or 4.4BSD Sockets, and adhering to the POSIX and XPG3 specifications, will run on OSF/1 without change. The OSF/1 system also is backward compatible with applications written using the System V Transport Layer Interface (TLI).

Performance – Network related components within OSF/1 have been fully parallelized for maximum performance on multiprocessor hardware. New and existing network applications running on OSF/1 can take advantage of the system's inherent support for multiprocessing and the parallelization of the STREAMS and Sockets frameworks to achieve high performance levels.

Network independence – OSF/1 provides the X/Open Transport Interface (XTI). XTI is an emerging industry wide application programming interface for network applications. Using XTI, developers can write network applications that are independent of the underlying transport mechanism. The XTI implementation in OSF/1 provides a path to either the Berkeley or STREAMS frameworks for compatibility with protocol families using either framework.

6. System Administration

OSF/1 provides 4.3BSD system administration and System V accounting, providing familiarity for users of many currently available systems. To these features OSF has added others that enable the addition or removal of subsystems to the running kernel, and that provide the logical volume management and disk mirroring required in many commercial processing environments.

Dynamic System Configuration

The dynamic configurability of the OSF/1 system enables system administrators to add or remove subsystems, such as physical and pseudo device drivers, network protocols, file systems, and STREAMS modules and drivers, to or from the running kernel. Benefits of dynamic system configurability include

- Simplified management of system configuration.
- Increased system availability. Dynamic system configuration means a greatly reduced need to shut down the system and rebuild and reboot it to change the subsystem configuration.
- Increased flexibility. Changes to the running kernel do not affect the OSF/1 modularized subsystems, and changes to subsystems (a device driver, for example) do not require rebuilding the kernel.
- Minimum memory usage by an OSF/1 system because unconfigured subsystem modules need not remain in physical memory.
- A simple and consistent manner to install and configure kernel subsystems.

Logical Volume Manager

The OSF/1 logical volume manager solves a problem posed by traditional UNIX implementations – limitation of file size to the capacity of a single external storage device. The logical volume manager within OSF/1 overcomes this limitation and provides features to safeguard data integrity.

Features and Benefits

Files can span multiple storage devices – The logical volume manager enables the creation of file systems and files that span multiple external storage devices. This is essential for applications, such as database management systems, that create large files which can exceed the capacity of a single external storage device.

Disk mirroring – Disk mirroring increases system reliability and availability by transparently creating mirror images of files on different physical devices. Reliability is increased because the OSF/1 system, in the event of damage to files on a particular volume, can switch to the mirror image files on another volume. The system can be configured to create up to three mirror images. Performance of read operations may be increased because the system dynamically tracks and obtains images from the location that can be reached within the minimum access time.

No changes to current applications are required to use the logical volume manager facilities.

7. Security

OSF/1 security provides the means to control access to the system and files, and to track and report security related events throughout the system. With OSF/1, vendors can deliver systems that can be rated at either the C2 or B1 level (as defined by the U.S. National Computer Security Center Trusted Computer Security Evaluation Criteria). The OSF/1 system also includes several higher level B2 and B3 features.

The OSF/1 security features include discretionary access control, mandatory access control, least privilege, auditing, password management, and labeling. Application developers can use these operating system security features to create trusted applications. Trusted applications are explicitly designed to uphold the security policies of the system.

8. Future Directions

Users of the OSF/1 operating system can be assured that the system will keep pace with near term technological developments, as well as provide a growth path toward future developments. The long range OSF/1 architectural strategy is to move many kernel services out of the kernel and into user space to increase system maintainability, extensibility, and flexibility. This microkernel architecture will provide a foundation for the development of distributed computing systems.

The content and direction of OSF/1 revisions are influenced by input from OSF members who represent a broad cross section of the worldwide information processing industry. Additional input comes from the OSF Research Institute and the worldwide research community. This process assures users of the OSF/1 system that they will be able to incorporate the latest advances in technology whenever such technology can best serve their businesses.

9. Compatibility

While the innovative features of the OSF/1 operating system are often the most talked about, its compatibility features are of equal importance to its users.

Hardware vendors require compatibility with their previous operating system implementation to protect their customers' installed base of applications.

Independent software vendors require compatibility with current operating system implementations and standards to provide a smooth migration path for the applications they market.

End users require compatibility with current operating system implementations to protect their investments in software.

OSF/1 provides compatibility features to satisfy all of these constituencies.

For hardware vendors, the OSF/1 system provides an extensible user space program loader. The loader can be modified easily to support the object file formats of applications running on the vendor's current operating system. This feature allows system vendors to provide backward binary compatibility for existing applications.

For independent software vendors and end users, OSF/1 can execute

- System V applications written to the System V Interface Definition Issue 2 (base and kernel extensions) that do not conflict with XPG3 and POSIX specifications
- 4.3BSD applications
- Applications written solely to the POSIX 1003.1 and XPG3 specifications.

10. The OSF Open Systems Software Environment

The OSF/1 operating system is the foundation for a collection of open systems technologies that enables users to mix and match software and hardware from several suppliers in a virtually seamless environment. By breaking down the barriers between diverse systems, this software

portfolio gives users the freedom to choose the systems and technologies that best meet their business needs.

OSF's vendor neutral software environment consists of several offerings, all based on relevant industry standards.

Layered on top of the OSF/1 foundation are

- The OSF Distributed Computing Environment (DCE), an integrated set of technologies that lets users access diverse network resources from the desktop
- The Motif graphical user interface, an offering that gives applications a common appearance and behavior on all classes of systems, from desktop devices to mainframes.

DCE and Motif are operating system independent and can be layered on other operating systems as well as OSF/1.

The OSF open computing environment is a rich one that will continue to evolve to meet industry needs. Working with its membership, OSF will consider both proven and emerging technologies for inclusion in the environment.

Through the Request for Technology process (RFT), for example, OSF is evaluating technologies for a distributed management environment (DME) and an architecture neutral distribution format (ANDF) for software. The DME will make open systems management more efficient and cost effective. The ANDF technology will enable software suppliers to distribute applications in a single format for use on a wide range of diverse hardware platforms.

11. Conclusion

The OSF/1 operating system melds the well accepted, time proven features of UNIX System V and Berkeley systems with advanced Mach kernel technology. It provides a portable, extensible, commercial quality operating system base for computing today, and a clear migration path to future distributed operating environments.

Hardware vendors can bring innovative and compatible open systems to market faster using OSF/1 as an operating system base. Independent software vendors will have a broad market for their applications. Commercial users will benefit from access to the large number of applications written for open, standards based systems, the innovative solutions made possible by the advanced features of the OSF/1 system, and from the availability of OSF/1-based hardware from many vendors.

Plan 9, A Distributed System

Dave Presotto Rob Pike
Ken Thompson Howard Trickey

Bell Labs, New Jersey, USA
{presotto | rob | ken}@research.att.com

Abstract

Plan 9 is a computing environment physically distributed across many machines. The distribution itself is transparent to most programs giving both users and administrators wide latitude in configuring the topology of the environment. Two properties make this possible: a per process group name space and uniform access to all resources by representing them as files.

1. Introduction

Plan 9 is a general-purpose, multi-user, portable distributed system implemented on a variety of computers and networks. Because commands, libraries, and system calls are similar to those of the UNIX operating system, it is possible to port many UNIX programs to Plan 9 with little or no changes. A casual user would find little difference between the two systems.

What distinguishes Plan 9 is its organization. The goals of this organization were to reduce administration and to promote resource sharing. Our programming style was minimalism. We believe that a small number of well-chosen abstractions can, with much less code, provide most of the function of a larger system. This is the approach that made the UNIX operating system so much smaller than its contemporaries such as Multics. In building Plan 9, we generalized proven ideas from the UNIX operating system rather than add new untried concepts.

Plan 9 is divided along lines of service function. Diskless CPU servers concentrate computing power into large multiprocessors; file servers provide repositories for storage; and terminals give each user of the system a dedicated computer with bitmap screen and mouse on which to run a window system. The sharing of computing and file storage services provides a sense of community for a group of programmers, amortizes costs, and centralizes and hence simplifies management and administration.

Since both CPU servers and terminals use the same kernel, users may choose whether to run programs locally on their terminals or remotely on CPU servers. Plan 9 provides this flexibility without constraining the choice. Therefore, both users and administrators can configure their environment to be as distributed or centralized as they wish. At work,

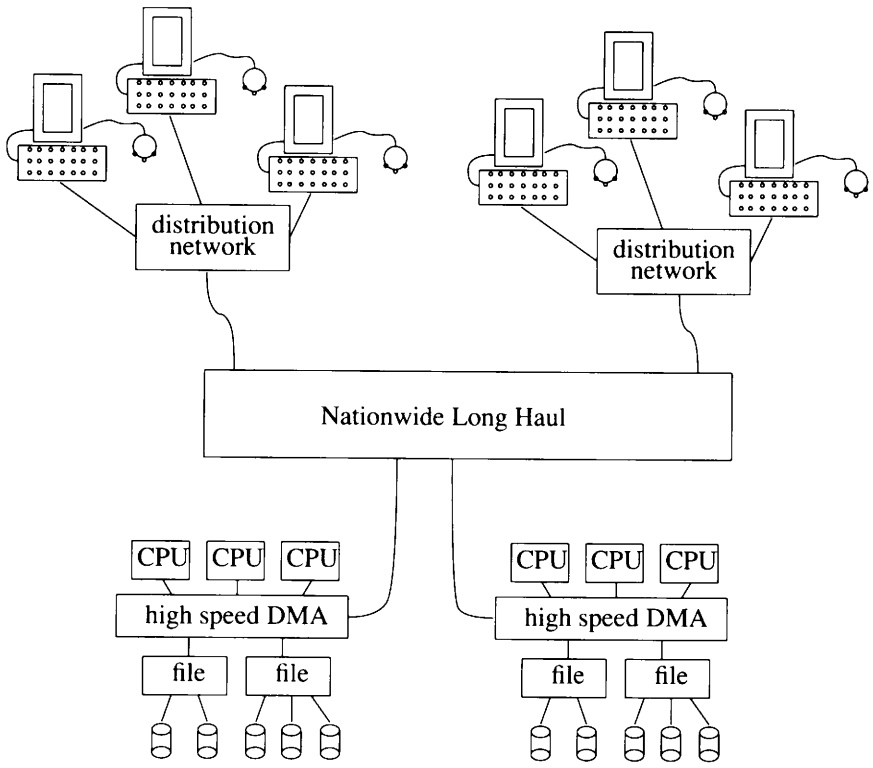


Figure 1: *Plan 9 Topology*

users tend to use their terminals more like workstations running all interactive programs locally and reserving the CPU servers for data or compute intensive jobs such as compiling and computing chess end games. At home, connected via a dedicated 9600 baud line to work, users choose what they run locally and remotely to reduce communication cost. Some applications, such as the editor [Pik87a], are split into multiple programs to make this choice even more flexible.

Figure 1 in any Plan 9 paper shows how we have configured our environment. Multiprocessor CPU and file servers are clustered in a few computer rooms and connected via 7 megabyte/sec point-to-point links [Pre88a]. This permits the CPU servers to be used as high performance compute engines without becoming starved for data. Terminals are connected to the servers via lower speed, lower cost distribution networks such as the 10 megabit Ethernet [Met80a] and 2 megabit Incon [Kala, Resa]. By emphasizing the shared service clusters we can quickly and cheaply incorporate new technologies as they arise. At the same time, users wishing more autonomy can incorporate as much computing power as they wish in their own offices without losing the advantage of transparently sharing other resources.

The rest of this paper describes the features of Plan 9 that make possible such a flexible topology. For more information on hardware and use of the system, see our previous paper [Pik90a]. For details of the file server, see [Quia].

2. Minimalism

All resources that a process can access, aside from program memory, reside in one name space and are accessed uniformly. Simply stated, all resources are implemented to look like file systems and, henceforth, we shall call them file systems. A file system is a strict tree with no links. File systems can be the traditional type representing persistent storage on a disk as implemented by the shared file servers. They can also represent physical devices such as terminals or complex abstractions such as processes. The file systems can be implemented by kernel resident drivers, by user level processes, or by remote servers.

A file system representing a physical device normally contains one or two files. For example, an RS232 line is represented as a directory containing a `data` and a `ctl` file. The `data` file is the stream of bytes transmitted/received on the line. The `ctl` file is a control channel used to change device parameters such as baud rate.†

Some file systems represent software concepts. Environment variables (as in UNIX) are implemented as files in a kernel resident file system. Even processes themselves are represented as directories with separate files representing different aspects of the process such as memory, text file, and control. Many things that require a system call in other operating systems are represented by I/O operations on files in Plan 9; reading the id of a process, the user id associated with a process, the time, etc.

A kernel data structure, called a *channel*, is used as a pointer to a file. A user level file descriptor is just a handle for a kernel channel. All I/O system calls eventually translate into nine primitive operations on channels. They are:

attach – point a channel to the root of a file system. The file system is told which user is attaching.

clone – make a copy of a channel. The new channel points to the same file as the old one.

walk – do a one level directory lookup on the channel and point it to the new file (or directory).

stat – get the attributes of the file pointed to.

wstat – change the attributes of the file pointed to.

open – check permissions prior to I/O on the channel.

read – read from the opened file.

write – write to the opened file.

close – close the opened file.

Each kernel resident file system is implemented by a *device driver* containing a procedure for each primitive operation. The device drivers are accessed indirectly via a kernel array, `devtab`, which contains 9 pointers per driver, one to each primitive procedure. Each channel contains an offset into `devtab` indicating the driver to be used in accessing the file it points to.

Accessing file systems not resident in the kernel is via a special device driver, the *mount driver*. All channels pointing to this driver contain a pointer to a communication channel. The mount driver turns operations on such channels into request messages written to the communication channel. The mount driver is written as a multiplexor allowing multiple outstanding messages. Because the messages on the commun-

† We neither need nor have an `ioctl` system call.

ication channel are transmitted using `read`'s and `write`'s, any type of channel can be used: a pipe to a process, a network connection, even an RS232 line. The `mount` system call, described below, is used to create a new mount device channel and supply a communication channel for it.

All Plan 9 components are connected using this file system protocol. The code used to encapsulate the primitives into request and reply messages is 580 lines long. The mount driver is 899 lines long. Compared to the equivalent NFS code implementing vnodes and XDR this is tiny.

Of the 18000 lines of code that make up Plan 9, about 5000 lines perform memory management, process management, hardware interface, and system calls. The rest are for the 17 different file systems implementing devices, networks, process control, etc. Since most of the file systems are completely self contained, the complexity of the kernel code is even lower than its 18000 lines would imply. A working, albeit not very useful, kernel can be configured containing only the file systems implementing pipes, a local root, and a console. This totals 5899 lines of commented C code (counted using `wc *.[ch]`). As a comparison, Mach's micro-kernel without device drivers has 25530 lines of C code (calculated, we're told, by counting semi-colons). By the same metric our minimal kernel is only 4622 lines long, less than 1/5 the size. In fact, our kernel with every file system included is still less than half the size of their micro-kernel.

One might note the similarities between `devtab` and parts of the UNIX operating system; the block device switch, character device switch, file system switch and vnodes. One advantage of Plan 9 is that we have recognized that these are all essentially the same mechanism and have implemented them as such.

3. Virtual Name Space

When a user boots a terminal or connects to a cpu server, a new process group is created for her processes. This process group starts with an initial name space that provides at minimum a root (`/`), some binaries for the processor the process is running on (`/bin/*`), and some local devices (`/dev/*`). The processes in the group can then either add to or rearrange their name space using two systems calls, `mount` and `bind`. The mount call is used to attach a new (not kernel resident) file system to a point in the name space. Its syntax is

```
mount(int fd, char *old, int flags, ...)
```

where *fd* is a file descriptor for a communication stream such as a pipe or a network connection and *old* is the name of an existing file in the current name space where the file system will be attached. The attachment creates a new mount device channel whose communication channel is that referred to by *fd*. Subsequent accesses to *old* and any files below it in the hierarchy become request messages written to the communication stream.

The bind call is used to attach a kernel resident file system to the name space and also to rearrange pieces of the name space. Its syntax is

```
bind(char *new, char *old, int flags)
```

where *new* is a name in the current name space[†] and *old* is the same as in mount.

How the attachment works depends on the *flags* specified in the call. One possibility is that the old file is replaced by the new one. How-

ever, when both files are directories, Plan 9 allows another possibility. The result can be the union of the two directories. The effect is that of putting one directory behind the other. In the case of name conflicts for files contained in the directories, the one in front wins. *Flags* specifies whether the new directory replaces, goes in front of, or goes behind the old one. This concept is essentially the same as the search paths used in the UNIX libraries and the various shells. In fact, Plan 9 has no search paths and uses these *union directories* in their place. When a command is executed, Plan 9 uses the directory `/bin` the same way UNIX uses an execution path.

The ability to specify the complete name space for a process that contains all resources the process can access forms the basis for a true virtual machine. Any aspect of a process' world can be rearranged. Remote objects can be substituted for local ones. Processes can implement part or all of the name space of other processes. This capability is the basis for a number of important services, three of which we present here.

3.1. The Cpu Command

We consider the shared CPU servers as accelerators for our terminals, someplace where commands can run while maintaining the same environment. It is important that as little as possible change when running on the CPU server. The virtual name space provides us with a means to make the CPU servers actually feel this way to our users. A command, `cpu`, calls a CPU server across a network. A daemon process on the server answers the call, creates a new process group for the caller, sets up a name space, and starts a shell process in the new process group. The name space set up is an analogue of the name space of the calling process on the terminal. In particular, local resources on the terminal, such as the screen and the mouse, become visible to the server processes at the same place in the name space as on the terminal. The standard input, standard output, standard error, and current directory of the cpu command become those of the remote shell. The directories mounted on `/bin` are changed to be those that contain executables for the CPU server's processor type (the terminal may be a 68020 while a CPU server could be a MIPS). In general, a user typing the cpu command just notices that things such as compilations speed up while graphics operations slow down.

After the initial handshake to pass information describing the caller's environment, the cpu command becomes a file server answering file system requests from the network connection. The server daemon mounts the network connection to the terminal in a standard place, `/mnt/term`, and then binds the resources it decides to keep into the same places in the new name space. For example, it binds

`/mnt/term/dev/mouse` onto `/dev/mouse`,

`/mnt/term/dev/bitblt` onto `/dev/bitblt`, etc.

Subsequent accesses to those files are converted by the mount driver in the CPU server into file system messages sent to the terminal.

† Local kernel resources are referred to by a syntactic escape (hack) in the name space. Any name starting with a "#" refers to a local resource. The first character following the "#" specifies the type of resource and the remaining characters are a parameter specifying the instance of the resource. Thus, to bind the local console to a standard place in the name space, one would use `bind("#c", "/dev", FRONT)`.

3.2. The Window System

The user interface is made up of three files:

`/dev/bitblt` – writes represent bitblt operations to the screen

`/dev/mouse` – reads return mouse events, i.e., button clicks and movement

`/dev/cons` – reads return keyboard input, writes put characters to the screen.

Between them, these devices represent all I/O to the user. The window system, 8.5 [Pik91a], offers processes a multiplexed view to these devices. When a window is opened, the window system starts a new process group for a command (usually a shell) that will run in that window. In that process group's name space, the window system mounts a pipe to itself in front of `/dev`. Subsequent references by the new process group to any of these devices are sent as file system messages to the window server. 8.5 interprets those requests as accesses of the window instead of the whole screen. Similarly, 8.5 multiplexes the mouse and the keyboard so that mouse and keyboard input is available to processes only when their window is selected.

The result is that any program written to use the kernel resident user interface will also work inside a window. Because this is also true of the window system itself, new versions of the window system can be run and debugged in windows of the current window system.

3.3. Network Gateways

One, sometimes insurmountable, problem is accessing a network to which a system is not physically attached. For example, a system may be connected to our Datakit [Fra80a] network but not to the DoD Internet. Many gateways exist that try to solve this problem by performing protocol to protocol translation. Unfortunately, few transport protocols have completely equivalent concepts. In order to perform the best translation, it is be necessary to know the semantics requested by the program. For example, TP4 has message delimiters but TCP does not. A protocol translator going from TCP to TP4 would not know which bytes correspond to a single write by the sender.

In Plan 9, every network interface is a file system. A gateway is a file server that serves its own network interfaces to other machines. A process that wants to get at a remote network connects to the gateway and mounts the gateway's interface to the remote network into its name space. Whenever the process accesses the interface, the mount driver will send the request to the gateway. Thus, the gateway sees exactly what the process does.

4. File Caching

In building our environment, we've been reluctant to add local disk file systems to any of our terminals or CPU servers. There are essentially two reasons for this choice. The first is administration. Anyone with a local disk must administer it. Any disk that has unique long term state requires both knowledge and time to administer. In fact, the Bell Labs computer center at Murray Hill is doing a lucrative business maintaining other peoples' disked Sun workstations because the owners have neither the time nor the experience necessary to do it themselves.

The second reason is sharing. Although most workstations can export access to their local file systems, when left up to individual users, this rarely happens. Terminals become personified and users become tied to a particular room to do their work.

Plan 9 survives without local disk file systems thanks partially to hardware and partially to caching. The CPU servers do so because their links to the file servers transfer at a substantial percentage of memory speed. The file servers maintain large main memory caches for their disk file systems. These servers are configured with 128 megabytes or more of main memory to ensure that there is plenty of room for cache. Getting a file from a file server is generally faster than it would be to get it from a local disk.

Office terminals are connected to the file servers by shared 1 or 10 megabit/sec links. Home terminals use 9600 or 19200 baud links. In both cases, the link is much slower than access to a local disk would be. To avoid the obvious performance hit, we use caching. To keep the caches coherent, we use file identifiers supplied by the file server. The identifiers are unique 64 bit quantities. 32 bits identify the file, the other 32 bits identify the version of the file. The version number is incremented each time the file is modified. Each time a file is opened the file server returns the identifier with the reply. Therefore, it is possible to guarantee coherency at each opening of a file.

Office terminals only cache pages of executable files. Whenever a program terminates, its unmodified text and data pages are not immediately freed. Instead they are retained until the space is required by other programs. When a program is rerun its executable file is reopened and the current version number returned. If the version number has not changed and pages remain from the last run, they are reused. If the version number has changed, any remaining pages of the stale version are discarded. Since most data intensive work is done on the CPU servers, this simple cache saves most of the traffic between office terminals and the file servers. Other caching could be helpful but would require much more complexity.

This cache might also have sufficed for home terminals if it were persistent, but it is not. Therefore, we have added disks to our home terminals to be used as write through caches of the file server files. As a write through cache, it contains no state that isn't duplicated on the file servers. Therefore, it needs little maintenance compared to a local file system. If the code discovers a disk problem, it reformats the disk discarding the current contents. If the user should suspect that the cache is contaminated, she can request that it be reformatted at the next boot. The system slows down until subsequent use refills the cache but no information is lost. The user need not consciously update the disk because the cache uses file identifiers to maintain coherency with the file servers. Each time a file is opened, the cache discards any stale data it might have for that file. The user doesn't have to copy what she needs to the disk because it is done as a consequence of her using the data.

The disk based cache is implemented by a process that resides between the kernel and the file server connection. For every read request, the process satisfies as much as it can with data cached on the disk. It gets the rest from the file server. Any new data that passes through it is saved on the disk. When the cache fills up the least recently used file is discarded. The amount of data cached for any one file is limited to 1.75 megabytes to prevent one file from displacing all others.

Because the disk based cache only checks for coherency when a file is opened, it provides slightly different semantics than that seen on office terminals which do not cache data files. This looser coherency constraint forces programs that communicate via files to ensure an open between each transaction. Thus far we have not had to change any programs because of it.

5. Conclusion

We have presented a distributed system that is simple in structure and flexible in its use. Both the flexibility and simplicity are the result of two properties, a per process group name space and a single resource interface. Coupled with some minimal caching we provide a simple system that is as usable at home as at work.

6. Acknowledgements

Many people helped build the system. We would like especially to thank Bart Locanthi, who built our terminal, the Gnot, and encouraged us to program it; Tom Duff, who wrote the command interpreter `rc`; Tom Killian, who built and programmed the Gnot's SCSI interface; Ted Kowalski, who cheerfully endured early versions of the software; and Dennis Ritchie, who frequently provided us with much-needed wisdom.

References

[Fra80a] A. G. Fraser, "Datakit–A Modular Network for Synchronous and Asynchronous Traffic," in *Proc. Int. Conf. on Commun.*, Boston, MA (June 1980).

[Kala] C. R. Kalmanek, "INCON: Network Maintenance and Privacy," Internal Memorandum 220106-0450, AT&T Bell Laboratories.

[Met80a] R. Metcalfe, D. Boggs, C. Crane, E. Taft, J. Shoch, and J. Hupp, "The Ethernet Local Network: Three Reports," CSL-80-2, XEROX Palo Alto Research Centers (February, 1980).

[Pik87a] Rob Pike, "The Text Editor sam," *Software – Practice and Experience* **17**(11), pp. 813-845 (November 1987).

[Pik90a] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," in *UKUUG Proceedings of the Summer 1990 Conference*, London, England (July, 1990).

[Pik91a] R. Pike, "8.5, The Plan 9 Window System," *1991 USENIX Summer Conference Proceedings* (1991).

[Pre88a] D. Presotto, "Plan 9 from Bell Labs – The Network," in *EUUG Proceedings of the Spring 1988 Conference*, London, England (April, 1988).

[Quia] S. Quinlan, "A Cached WORM File System," *Software – Practice and Experience*, p. To appear.

[Resa] R. C. Restrick, "INCON Wire Interface Integrated Circuit Design," Internal Memorandum 52413-860314-01TM, AT&T Bell Laboratories.

UNIX International's Enterprise Computing Architecture for UNIX System V

Andrew Schuelke

UNIX International – Europe

andy@uieu.ui.org

Abstract

With UNIX System V Release 4 now firmly established in the marketplace, UNIX International recently announced its new extended charter which expands UI's focus to encompass functionality beyond that of the base operating system. With this expanded focus UNIX International will address the requirements of the industry in three environments critical to the success of open systems in the coming decades: Distributed Computing, Corporate Hub Computing, and Desktop Computing. These areas are tied together in the Enterprise Computing Architecture (ECA).

In this talk I will present the architectural framework of the distributing computing components of the ECA. These components are the Basic Services (the core operating system) at the lowest level, moving up through the Network Communication Services (e.g. STREAMS), the System Services (e.g. object management services), the Application Services (e.g. transaction services), and at the highest level the Application Tools. Spanning these hierarchical services are the Security and Interoperability Services. In discussing Interoperability Services I will cover interoperability with IBM SAA, PCs, and existing distributed computing tools.

Evaluation of Distributed Operating Systems in Open Networks

Holger Herzog
Markus Kolland Juergen Schmitz

Siemens AG, Germany
makol%venedig%ztivax@unido

Abstract

This report presents an evaluation method for distributed operating systems, which is based on necessary operating system functionality for specific application areas. Application areas in this context are distributed, real time, multiprocessor, security and UNIX applications. For this purpose it is necessary to investigate the functionality offered by a given system to support features like Object Orientation, Concurrency, Multiprocessor or Real Time capabilities. The evaluation method is applied to Mach and Chorus and gives hints to estimate the suitability of these systems for different application areas.

1. Introduction

Distributed systems are regarded as the computing base of the nineties. They offer significant advantages like transparent view of the system, high availability, fault tolerance and concurrency. Future operating systems should cope with aspects of distribution as well as real time capabilities and compatibility with standards (preservation of existing SW). In addition they should provide support for heterogeneity and structured system design. Kernelized systems like Mach and Chorus are possible candidates for the implementation of a system fulfilling these requirements.

A sound basis for estimating the suitability of an operating system for building an application environment with the requested characteristics is therefore necessary. The methodology for evaluating given operating systems with respect to specific requirements we developed is such a basis.

We structured the evaluation of operating systems into three phases. In the first phase, we identify the architecture, components and mechanisms offered by the system and it's terminology. The second phase (structural evaluation) is the focus of this paper. Here we investigate the given system with respect to the functionality necessary for specific application areas (distributed environments, real time or UNIX applications, multiprocessor architectures and applications with special secu-

rity requirements). The mechanisms offered for these different purposes can be quite different in their implementation and efficiency. Given the limited space of this paper, it is focusing on issues like object orientation, support of transparency aspects, support of concurrency and consistency, multiprocessor and real time support. In the final section of the paper the viability of this part of our methodology is shown by applying it to the Mach and Chorus systems. The third phase, measuring the performance of a system with respect to specific application areas, will be investigated in the future.

The paper is generally structured with respect to these topics. Beside the operating systems themselves we also take into account developments in the environment of a given system which provide support for some required functionality. This should provide a better judgement of the future perspectives of a given distributed operating system.

2. Object Orientation

Object Orientation is a method of structuring large systems. Objects in the programming language sense have the following features:

- Objects are capsules which include data structures and operations similar to an abstract data type (information hiding).
- Operations are invoked by messages.
- Each Object is member of a certain class (called it's type).
- Classes are hierarchically ordered and can therefore share data and operations (inheritance).

In the context of operating systems we usually only regard the first two features. Future operating systems should provide this object oriented view in their internal construction as well as in their provision of mechanisms to support object oriented applications.

Distributed systems make it possible to implement replicated objects, which can be executed on physical distinct machines. They are the conceptual basis for implementing migration, load balancing and fault tolerant services. From an application point of view the implementation details of these features are transparent. Conceptually they are implemented by object groups (functional object groups, object groups with a coordinator, object groups with a voter) [Esp89a].

Within functional object groups, different objects are tied together into a group. Each object implements an access point to the service offered by the group. These access points can be invoked in parallel.

In object groups with coordinator one of the replicated objects takes the role of a coordinator which carries out the requested operations. The other replicas are getting informed about the status of the coordinator at fixed time intervals [Bir89a].

In object groups with a voting mechanism all objects are carrying out the same request. To achieve fault tolerance a voter is deciding on the result of the operation after it has examined every result of the replicated objects. In a simple case the decision can be based on a majority vote.

Beside the support of the application's object oriented view the internal structure of the system with respect to object orientation should be examined.

Figure 1 summarizes the different object structures in the object oriented approach.

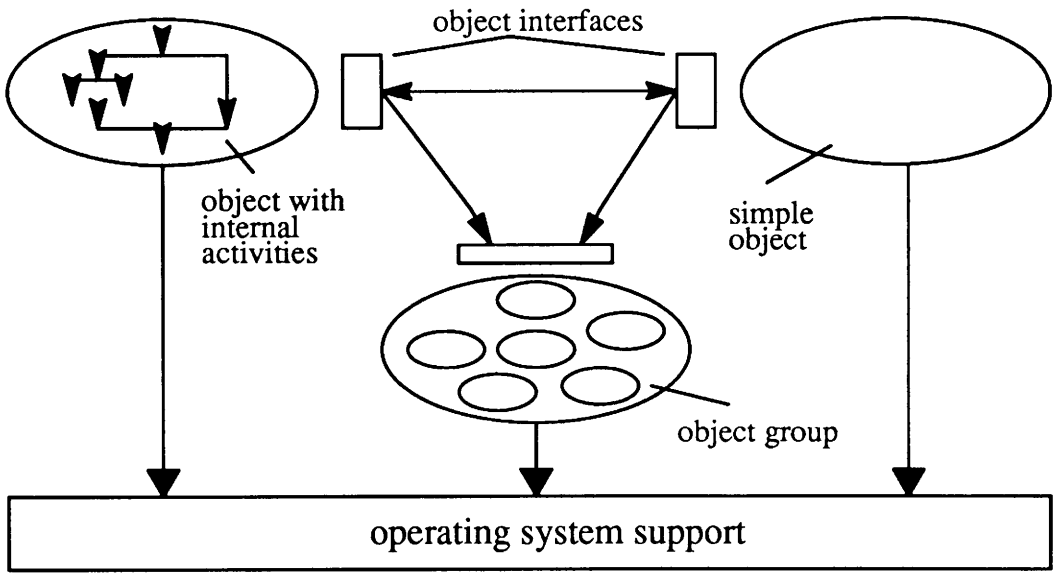


Figure 1: *Different kinds of object*

3. Transparency

One of the major requirements of office and business applications is the transparent view onto the architecture of the underlying system. The advanced features distributed systems can offer (fault tolerance, concurrency, etc.) can be stressed in terms of the functionality hidden from the user. This is described by the different kinds of transparency listed below. Transparency in this case is a synonym for the implementation of system functionality while abstracting from the implementation details. The following kinds of transparency can be identified [Esp89a, Her90a]:

- *access transparency* – hiding the explicit addressing of objects in the system
- *location transparency* – hiding the topology of the system
- *scaling transparency* – system mechanisms are stable due to growth of the system
- *replication transparency* – hiding multiple instances of objects (states)
- *failure transparency* – hiding the occurrence of failures in the system

Other transparency aspects like performance, scaling or migration transparency can also be identified but are of minor importance here.

Distributed operating systems offer various mechanisms to hide an underlying distributed system from an object (application) running on top of it. The question however is, whether there are kernel mechanisms which directly implement the given transparency or whether there are mechanisms which can be used by some software layer to implement the transparency under consideration. Additional investigations should be made whether transparency aspects can not be supported at all or only with major performance drawbacks?

4. Concurrency and Consistency

Activities in a computer system are normally denoted by "process" or "task". A process is the smallest entity of an activity from an operating system point of view. The term "task" is normally used instead of "process" in real time systems. In the field of database systems "threads" have been introduced as the smallest unit of activity in a system. Communication protocols divide between "lightweight" and "heavyweight" processes.

The trend in operating system technology towards the division of the former process concept into the environment of an activity and the activity itself (Task/Thread, heavyweight/lightweight process) corresponds to that. This is visible from the system's and user's point of view. There is a heavy demand that multiple activities should be able to run in parallel in the context of the same environment. The environment denotes all resources associated with the activities, the activity denotes a virtual processor (registers, etc.).

The capability of having parallel activities running in an operating system brings forth the problem of data consistency. Atomic transactions, known from the database world, have been proved as a helpful model in the field of distributed systems. Nested Transactions are helpful for structuring activities in a system in a modular way and to support the parallel execution of multiple activities while preserving consistency. Faults can be handled in any hierarchy level which is especially useful for developing distributed applications [Wei89a].

In the field of programming languages efforts have been undertaken in developing language constructs to specify concurrent actions and their synchronization in terms of the consistency of accessed data.

From this point of view an operating system should be capable to efficiently manage parallel activities and provide mechanisms for their specification, synchronization and obedience of consistency constraints.

5. Multiprocessor Support

Because of the increased need for processing power computations are getting more and more decoupled and run on parallel hardware. Multiprocessor architectures and special programming language features have been developed for this purpose. An operating system has to support this functionality in two ways:

- Applications are normally written in a language like C or PASCAL, which do not inherently support parallel programming. Creation and management of parallel activities have to be done by a runtime system (library) which is based on certain operating system functionality (Synchronization, etc.).
- Multiprocessor hardware has special requirements with respect to scheduling and dispatching of activities. Efficient use of the high performance and parallel resources is a major requirement.

To offer fine grain parallelism of operations, there must be a simple smallest entity of activity managed by an operating system. Thus context switches can be performed with minor performance penalties. These leads to the concept of lightweight processes (section 4). To offer a generic approach for different kinds of architectures, operating

system have to deal with different kinds of processor structures (topology, connection) and different kinds of memory architectures (uniform, non-uniform, remote). The latter is especially important to give parallel applications ta transparent view of the multiprocessor architecture of the underlying hardware. Uniprocessor machines must be supported without performance penalties New developments in the operating system's domain try to fulfill these requirements by providing the task/thread concept (section 4), implementing an architecture independent memory management and offering extended functionality for the support of parallel applications.

6. Realtime Capabilities

A computing system operating in real time mode serves requests due to external events in step with the occurrence of these requests. This means, that the result of any computation has to be available at some fixed point in (real-)time, and that there must be a capability to manage multiple tasks in the system. Because these single tasks can't be computed independent of each other some kind of cooperation between the tasks is necessary (communication, synchronization). Fulfilling these requirements, an operating system must provide some minimal functionality, which include the following features [IEE89a]:

- Because of the nature of real time applications the system must be able to manage multiple processes (tasks) each running multiple parallel threads in it's context. With respect to timing constraints, efficient means for communication (IPC, shared memory) and synchronization(semaphores) must be available.
- The system must respond to external events in a predictable time span. This put's constraints on the time for a context switch and the reaction to an external event. Mechanisms to support these aspects are fixed priorities for processes and interrupts and a preemptive scheduling algorithm. Timers must be available to take action in case a timing constraint has not been satisfied.
- To speed up context switches the memory management should be capable of locking whole memory areas to save them from being swapped/paged out.
- To cope with the real time requirements a system must provide sophisticated mechanisms to serve asynchronous events. The mechanisms which are offered by general purpose operating systems have several unacceptable disadvantages (signals can be lost, no data can be transmitted with signals, random sequence of signal handling).
- The I/O Subsystem must offer asynchronous (to enhance throughput) and synchronous (to ensure integrity) I/O.

7. Structural Evaluation of Mach

7.1. Mach and Object Orientation

The Mach port concept implies an object oriented view of the system. A port is a guarded simplex communication channel between tasks, for which the end points (tasks) can have send, receive or ownership right or a combination of these [Bar90a].

From a user's point of view a port represents an object. Data of this object is encapsulated in a task with receive right on this port. Normally another task can only access data of this object by sending a message to it's associated port. This concept fulfills the principles of data encapsulation and information hiding. Mach however offers the possibility to share memory regions between tasks, which is restricted under certain rules. This means a violation of the object oriented approach.

The invocation of operations in an object has it's counterpart in the mechanism for inter task communication. A task (object) invokes an operation in another task (object) by sending a message to the appropriate port. The message causes the invocation of the operation within the task.

From this point of view the Mach system itself and the services running on top of it are structured in an object oriented way. Services are requested by sending a message to the port representing the server, which causes the invocation of an operation in the server. This concept directly reflects the object oriented approach.

Dynamic Binding can be also achieved in the Mach system. The Environment Server does the mapping between portnames and portnumbers. The portnumber represents the task (i.e. object) which actually invokes the operation. The portname represents the object, to which the message is sent from the users point of view. This dynamically configurable mapping directly reflects the concept of dynamic binding in the object oriented approach.

However major object oriented features are missing in Mach:

- **classes**
  There is no analogy to the class concept, i.e. there is no notion of the type of a task (object) or its's instantiation from a type description.
- **inheritance**
  Mach completely lacks the principle of inheritance respectively delegation of operations or data between objects, which is characteristic for the object oriented approach. Although Mach tasks can inherit memory regions to their children, this does not correspond to inheritance in the object oriented sense.

The Mach system offers two ways to support the user in writing object oriented applications, MACHOBJECTS and MATCHMAKER/MIG.

MACHOBJECTS

This is a set of C macros which offer syntactic constructs to define classes, objects and methods (like in C++). MACHOBJECTS offers different strategies of inheritance (single inheritance, delegation). Interesting in this context ist the construct remote objects. Thi s allows delegation to objects in other address spaces. It's implementation is based upon the mechanisms of inter task communication, where a local operation invokes an operation on an object in a different address space by communicating with it's associated port (sending a request). Detailed information on MACHOBJECTS can be found in [Jul90a].

MATCHMAKER / MIG

MATCHMAKER is a language to specify interfaces between objects (tasks in Mach terminology). MIG is a subset of MATCHMAKER. The user can write a specification in C like syntax in which he describes the

operations of the interface and their invocation conventions. MATCHMAKER generates a client side module, a server side module and an include file with global parameters and definitions from this description. This allows a user to abstract from the level of explicit communication between two objects and to view the request of a service like the invocation of an operation in the object oriented view. Detailed Information on MATCHMAKER can be found in [Jon86a].

7.2. Support of Transparency Aspects in Mach

Access and Location Transparency

An object in Mach can only be accessed by sending a message to it's associated port. The user knows this port by it's name (string), which is mapped to a portnumber by the Environment Server. This portnumber is used to locate the port. Ports which are located on a remote site are handled by the Network Server. He transparently maps the portnumber onto an access to the remote site (via the underlying network). From the users point of view the access to an object in the system is independent from the location of the object and it's features. This can be used e.g. for the implementation of a file system, where a file server hides the physical distribution of the data in the system. The Mach notion of memory mapped Files in [Tev87a] has to be mentioned in this context. The memory management implemented in the Mach kernel allows the mapping of memory objects into the virtual address space of a task. So file data can be accessed in the same way as memory data. Paging is done by a pager (user task) associated with each memory object. Data can therefore be transparently accessed by a virtual address, independent of it's location (memory, local storage, remote storage). This gives a completely transparent access to all memory and storage resources in the system.

Scaling Transparency

A Mach system, which is enlarged due to additional services, resources or growth of applications must not been altered structurally nor has the construction of applications to be changed. All mechanisms and components are stable due to any incremental growth of the system. Additional services can be added simply by registering their names and port numbers in the Environment Server and Network Server respectively. Additional devices can be added by connecting them physically and starting the respective server (see above), making the new services available to the user community. Additional nodes can be integrated in an existing system by connecting them and registering all services running on that node in the Environment/Network Servers (simplified). An interesting case of scaling is the enhancement of processor capacity. There are no changes necessary on the application's side to use this capacity. The assignment of processor's to a task in Mach is done at the user level. Mach offers certain kernel services to implement this in a controlled and consistent manner, where the actual number of processors at a node can be used. The major functionality available is the creation and mangement of processor sets which can be assigned to tasks. Threads in this task are then scheduled on the processors of the processor set [Bla89a]. Due to the virtual memory management implemented in Mach, the address space of an application can be enlarged independent of it's size. The virtual address space of a task is a set of memory regions which are represented by address map entries. These specify access rights, the virtual address range and inheritance/sharing

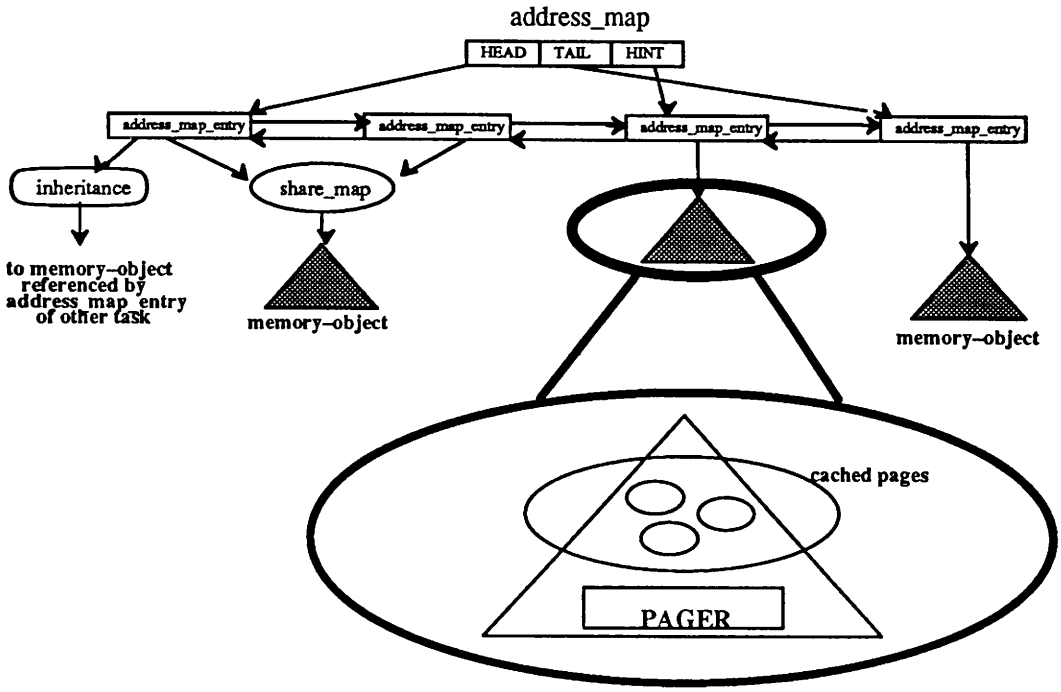


Figure 2: *Virtual Memory Management in Mach*

information. A pointer to a memory object gives the reference to the storage area which is mapped into the region and which contains a user level pager and caching information on currently mapped pages. This scheme is illustrated in Figure 2. There are kernel services to allocate a memory region (create a new entry in the linked list) and to establish a reference to a memory object independent of the application's size.

Replication Transparency

There is no direct support of this feature in Mach, but there are mechanisms which an implementation can be based upon. Object groups can be built by registering the names of multiple servers (each providing the same service) in the under different names. Their associated ports can be grouped together by the Mach port group concept, which allows a client to send a request to a port, not knowing which of the task in the portgroup is carrying out his request (replication is transparent to the client).

Failure Transparency

Similar to Replication Transparency Mach only offers mechanisms to base an implementation of failure transparency on, but no direct support. The concept of the for example offers the capability to assign a backup port to a primary port which means that in case the primary has been crashed, the task with receive right on the backup port automatically receives this right for the crashed primary. So a backup task can transparently take over the job of the primary task. Subsystems which support the development of fault tolerant (transparent) applications have been developed or ported on Mach basis, like the transaction system CAMELOT [Wei89a] or the ISIS Toolkit [Bir89a]. There are however deficiencies, especially in the Mach communication mechanisms, which make it quite difficult to implement a generic fault tolerant system on top of it [Bab89a].

7.3. Support for Concurrency and Consistency in Mach

Task/Thread

A "task" in Mach denotes the environment multiple activities are executed in. A task has associated with it all the resources it's activities can access (memory space, ports, processors, etc.). The resources of the task are protected against access from other tasks, memory regions can be inherited by ancestor tasks ore shared between tasks (see Figure 2). A "thread" is the smallest entity of activity in a system and denotes a virtual processor (registers, stack, etc.). Multiple threads can run in parallel in the context of a task and access the task's resources. This can be used to write where a server is not blocked by a request but creates a parallel thread for each request. In UNIX terminology a Mach task with threads is equivalent to a process. Synchronization can be achieved by using the event mechanisms of the Mach kernel, communication can be done by the Mach IPC or by using shared memory regions.

C Threads

The C Threads package is a runtime library which supports the development of parallel applications in C. The application itself is running as a task, the parallel activities are mapped onto threads in the system. The following mechanisms are available [Coo87a]:

- Fork/Join of parallel threads
- Synchronization with events by means of condition variables and operations on them
- Mutex variables and operations for mutual exclusion in critical regions
- Scheduling of threads using the processor allocation interface (section 7.2)

Camelot

From a user's point of view CAMELOT is a language extension through libraries. The application programmer thus has access to the concept of nested transactions in his native language environment. Statements (assignments, procedure calls, RPC, etc.) can be put into transaction declarations, which can be nested and are executed in parallel. CAMELOT allows locking of certain program regions (mutual exclusion). Locks are inherited given the usual semantics of nested transactions. Data areas can be specified as recoverable which means that CAMELOT insures the consistency of this data area with respect to aborted transactions.

7.4. Multiprocessor Support in Mach

One of the main objectives in developing Mach was the efficient support of multiprocessor architectures. Dividing the process abstraction into execution environment (task) and activity (thread) (section 7.3) allows rapid context switches between threads, because the context is task specific. This leads to efficient exploitation of multiprocessor architectures and special purpose processors. Memory management is one of the main problems in exploiting a multiprocessor architecture. The virtual memory management implementation in Mach is strictly divided in hardware dependent and independent part. Where the first

part maps a virtual address in to a tuple (logical page#, offset), the latter maps this tuple onto the physical address. This mapping can deal with non-uniform memory accesses (NUMA) but only on the local node (no NORMA layer). The processor resources of a node, given the necessary privileges, can be managed at user level by invoking the processor allocation interface. Thus characteristics of the application with respect to parallelism can be exploited by assigning processors to tasks (section 5).

7.5. Real Time Capabilities of Mach

The Mach kernel originally has not been build with real time capabilities in mind. Nevertheless some of the features stated in section 6. are supported. There is currently a lot of work going on at CMU to develop a real time Mach kernel [Tok90a]. Mach fulfills the multitasking requirement. The thread concept caters for the possibility of parallel activities within processes and minimal performance penalty in switching between activities. Copy-on-write in the memory management scheme is one of the means to speed up context switches between tasks. The demand for deterministic and efficient IPC and shared memory for very fast communication is satisfied as well. Binary semaphores are supported by the C Thread mutex variables. The Mach kernel can not react to external events with a fixed guaranteed response time due to it's scheduling policy and lack of priority concept. Memory locking as well is not yet possible. I/O in Mach is done by servers which act as device drivers. They are invoked by sending (receiving from) them data via the Mach IPC. The characteristics of the Mach IPC fulfill the requirements for I/O in real time systems stated in section 6. The currently missing features concerning real time capabilities, mainly faster context switch between tasks, fast communication, a preemptive kernel and hard real time threads/tasks with priorities will be integrated into the Mach kernel in the future.

8. Structural Evaluation of Chorus

If not stated otherwise all statements made in the following paragraphs concern the Chorus kernel, not the UNIX subsystem on top of it.

8.1. Chorus and Object Orientation

Chorus offers actors, activities, memory regions, memory segments, ports, portgroups and semaphores as basic system abstractions. According to the given definition they do not represent objects in the strict object oriented sense, because either they do not implement data encapsulation (e.g. data in the kernel is not only accessible via the given operations) or operations can not only be invoked by messages. The presence of classes or the possibility of inheritance between objects isn't implemented at all in the Chorus system.

Nevertheless the offered mechanisms can support an object oriented users point of view. Data can be encapsulated in actors and be made accessible only through the invocation of operations in this actor. This "object" can then be used from another actor by sending a message to the port associated with the "object" actor. The principle of dynamic binding, where operation invocations are dynamically bound to the operation code, is achieved by the possibility of port migration. Port

migration is a concept, in which ports can be associated with different actors during runtime.

The Chorus kernel is implemented in the object oriented language C++. All in all it can be stated, that the implementation of the Chorus system itself was done by object oriented means. However the offered system abstractions hardly fit into the object oriented paradigm. Besides the lack of classes and inheritance, one of the main reasons for that is the possibility to reference data and operations not only by sending a message to a port but also by their internal addresses.

To fully support the object oriented paradigm Chorus offers the Chorus Object Oriented Layer COOL [Cho90a], which is currently under development. It extends the set of Chorus system abstractions and provides additional mechanisms for a generic support of object oriented systems. These abstractions allow the creation, deletion, migration of objects as well as the remote access to objects and the storing and retrieval of them. Specific type models, complex object semantics or high level transaction mechanisms are not offered at this layer but they can be implemented on COOL basis. COOL is implemented as a layer on top of the kernel to provide an interface for a wide range of object oriented languages (e.g. Eiffel), applications (e.g. Emerald) and systems (e.g. Comandos) [Esp87a].

The additional abstractions provided by COOL are contexts, objects, threads, inter-object communication and persistency.

- *context* represents the execution environment of objects. A context is defined as a virtual address space into which one or more objects are mapped and corresponds to a Chorus actor.
- *object* represents the usual object abstraction consisting of a code and data segment, which are mapped into a context. COOL manages objects but defines no fixed semantics on them.
- *threads* are the smallest unit of activity in the sense of "lightweight processes". COOL threads are mapped onto Chorus threads. Multiple threads can run in parallel in a context.
- *inter-object*communication is based upon the Chorus IPC. Objects have ports associated with them on which threads can send and receive messages. Special messages for copying or migrating objects are available.
- *persistency* is a feature, where the lifetime of objects is independent of the applications which reference them. Persistency is available for objects and whole contexts. Objects are being stored on a stable storage medium where contexts are saved by checkpointing and can be later restored from them.

A further feature of COOL is the capability to tie objects together into groups. This allows communication with groups of objects instead of single objects. It's implementation is based upon the broadcast/multicast mechanism in the Chorus IPC.

8.2. Support of Transparency Aspects in Chorus

Access and Location Transparency

The access to services in the Chorus system is done by sending a message to the associated port/portgroup of the actor (section 8.1) which implements the service. The clients need not be aware of the server specific invocation requirements, it just supplies the parameters for the invocation in the message. Access to memory is handled like in cus-

tomary virtual memory management schemes. The mapping between virtual address and physical address is carried out transparently with the help of dedicated servers, which do the paging. The association of a memory region with a storage area to transparently access data in memory and storage (section 7.2) is not implemented in Chorus. Like stated above, services are accessed via their associated ports. A port is identified by a system unique number with some access information (capability). These unique identifiers (UI) are used for all system abstractions. The location of the object associated with an UI is figured out by the "Unique Identifier Location Service" (UILS). For a local object, the UILS can resolve the UI by searching it's local tables, otherwise this is done by the It first searches the local cache of ports and portgroups. In case of failure it requests the information from the location, where the port was created (coded in the UI). If this fails too, a message is broadcasted to find the actual location of the object.

Scaling Transparency

Adding a service in the Chorus system can be done by starting the respective actor and making available an interface to this service. This is usually done by making available the UI of a port, associated with the actor, which can be used for sending requests. This scheme does neither require a change of the application nor does it require to rebuild the system or change certain mechanisms. Adding new devices requires some administration effort and a new configuration of the kernel. New nodes in the system can be made available by adding entries in the respective tables and booting the system.

Replication Transparency

Chorus does not support replication transparency directly by any system abstraction, but given the Chorus IPC an implementation of transparently replicated objects is possible. Ports can be grouped together to port groups and messages can be send to these port groups using different modes:

- **Broadcast** – message is sent to all ports of the group
- **Functional** – message is sent to one port of the group
- **Associative** – message is sent to one port of the group, which complies to specific attributes

The different communication modes and their use are illustrated in Figure 3. Replicated object groups can be implemented quite straightforward and the communication mechanisms make it possible for the application to send requests and receive results without having to know the structure of the object group.

Failure Transparency

Failure transparency can be implemented building an object group with coordinator/voter for providing a service instead of one single object. Any application can use this service without having to know the internal construction of the group (see replication transparency). For the IPC, Chorus uses standard protocols (TCP/IP, ISO OSI) and thus provides their specific fault transparency features. The implementation of the RPC allows to detect communication failures like crashed sites and lost requests. This can be used to capture these problems transparently to the application using a service. Interrupts, exceptions and traps can be dynamically associated with handler routines in the private task

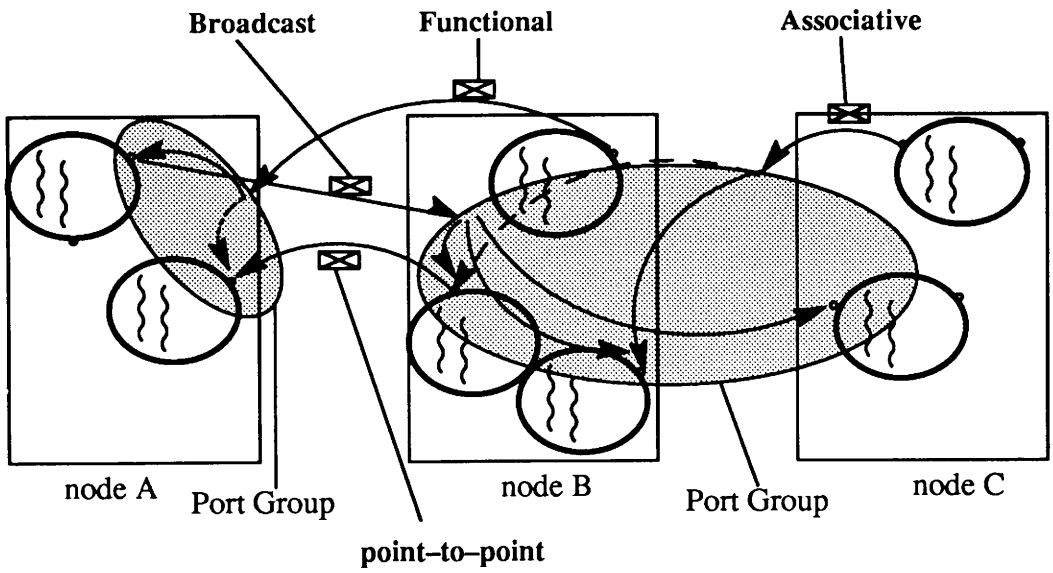


Figure 3: *Communication modes in Chorus*

address space. This mechanisms supports the capturing of external events and failures transparently to the application.

8.3. Support for Concurrency and Consistency in Chorus

Actor/Activity

The actor/activity concept implemented in Chorus directly corresponds to the task/thread concept in Mach. The statements made for Mach in this context can be applied to Chorus respectively [Arm90a] (section 7.3). In the Chorus system there is a distinction between supervisor actors, system actors and user actors to specify different privileges. The address space represented by an actor is also divided into system and user space for this purpose. Scheduling of the activities is preemptive, which means that the activity with the highest priority is actually running on a processor, where the priority is computed on the basis of the actors and the activities assigned priorities. Scheduling is done by the kernel.

Synchronization and Communication

To insure consistency in case of parallel activities Chorus offers semaphores and mutex objects (binary semaphores), which are based upon the shared memory concept. Actors can communicate via the Chorus IPC or shared memory. Activities, which share the same address space can implement their own communication.

Language Support

The COOL layer on top of Chorus offers mechanisms to specify parallelism and synchronize parallel activities. Efforts are currently undertaken to implement the object oriented distributed system COMANDOS (on top of the COOL interface. Beside the management of distributed object oriented applications the COMANDOS language offers constructs to specify, synchronize and communicate between parallel activities

8.4. Multiprocessor Support in Chorus

Like stated in section 8.3 the process concept known from customary operating systems is split up in Chorus into actor and activity. The minimal context associated with an activity allows switching between activities with minimal overhead which means efficient exploitation of parallel hardware. To support different memory architectures, Chorus divides it's memory management into hardware dependent and hardware independent part. However there is currently only support for uniform memory architectures.

8.5. Real Time Capabilities of Chorus

From the beginning Chorus was developed with real time requirements in mind. Actors provide the multitasking capability of a Chorus based real time system, activities offer the required parallelism within tasks. To guarantee response times to external events, Chorus employs different mechanisms. Preemptive scheduling and the priority scheme in connection with rapid context switches, fast communication, copy-on-write and memory locking are proper means for that. Actors are divided into real time actors and time sliced actors. The former group has fixed priorities which are higher than the variable (UNIX like) priorities of the latter ones. In addition Chorus offers timers to control the obedience of response time constraints. To synchronize tasks Chorus offers efficient mechanisms at the system level (semaphores, signals, etc.). Communication can be done via the Chorus IPC or by employing shared memory in case of special performance requirements. The specification of timeouts for invoking system calls helps avoiding deadlocks. The handling of asynchronous events (interrupts) by offering the opportunity to associate any interrupt with a handler routine in user space and it's implementation fulfill the requirements for real time support. I/O in the Chorus system is done by external servers which are invoked by sending a request to their associated port. The option to choose synchronous or asynchronous communication corresponds to the demands of a real time system (see section 6)

9. Conclusion

In this paper we have presented an evaluation method for distributed operating system. Focusing on the structural evaluation we took into account the importance of various application areas which have to be supported by the underlying system in the future. We have given a brief introduction into the most significant application areas investigating the basic necessary functionality of the underlying system to support the latter. The first phase, identifying the architectural concepts and the last phase, measuring the performance are not presented in this context but have been already finished.

The application of this methodology onto the systems Mach and Chorus show the suitability of these systems as a basis for the respective application areas. It's not enough to compare architectures or mechanisms of different systems, this comparison has to be put into the context of the required functionality. This is what the presented methodology does. More work has to be spend on performance measurements, because the same is true for this evaluation phase: it's no use comparing mechanisms in isolation, it's necessary to evaluate them in

the scope of their application. We will do that in the scope of a successor project.

In our opinion, applying these three phases to the evaluation of operating systems in general should provide a sufficient base to estimate the suitability of a system for the specific application areas which will be coming into focus in the next years.

References

[Arm90a] F. Armand, "Multi-threaded processes in Chorus," *EUUG Conference Proceedings*, Munich, Germany (1990).

[Bab89a] O. Babaoglu, *Fault-Tolerant Computing Based on MACH*, University of Bologna (1989).

[Bar90a] E. Baron and others, *MACH Kernel Interface Manual*, Carnegie Mellon University (1990).

[Bir89a] K. Birman and T. Joseph, *Exploiting replication in distributed systems in [Mul89a]*, 1989.

[Bla89a] D. Black, *A MACH Processor Allocation Interface*, Carnegie Mellon University (1989).

[Cho90a] Chorus, *COOL2 – Chorus Object Support Platform: Initial Specification*, February 1990.

[Coo87a] E. Cooper and R. Draves, *C-Threads*, Carnegie Mellon University (1987).

[Esp87a] Esprit, "COMANDOS: "Object Oriented Architecture"," *Project 834, Deliverable D2T2.1* (September 1987).

[Esp89a] Esprit, "An Engineer's Introduction to the Architecture," *ISA Project* (November 1989).

[Her90a] H. Herzog and B. Stork, "Cooperation Support for UNIX-based Industrial Applications," *EUUG Conference Proceedings*, Munich, Germany (1990).

[IEE89a] IEEE, *Realtime Extensions for Portable Operating Systems P1003.4, Draft 8*, August 1989.

[Jon86a] M. B. Jones and R. F. Rashid, "MACH and Matchmaker: Kernel and Language Support for ObjectOriented Distributed Systems," *Proceedings 1st ACM Conference on ObjectOriented Programming, Systems, Languages and Applications (OOPSLA)* (September 1986).

[Jul90a] D. P. Julin and R. F. Rashid, *MachObjects*, Carnegie Mellon University (1990).

[Mul89a] S. Mullender, "Distributed Systems," *Frontier Series*, , ACM Press (1989).

[Tev87a] A. Tevanian, *Architecture Independant Virtual Memory Management for parallel and distributed Environments*, Carnegie Mellon University (1987).

[Tok90a] H. Tokuda, *Real-Time MACH Progress Report*, Carnegie Mellon University (1990).

[Wei89a] W. Weihl, *Using Transactions in Distributed Systems in [Mul89a]*, 1989.

A Distributed Computing Environment Framework: An OSF Perspective

Brad Curtis Johnson

Open Software Foundation
bradcj@osf.org

Abstract

This paper articulates an architectural framework, and the fundamental mechanisms that are required to support that framework, for a distributed computing environment. The emphasis of this framework is on open distributed systems. That is, it serves as a model that supports many of the requirements of a distributed system: such as the need for interoperability, the need to support the client/server distributed application model, and the need to account for the characteristics and challenges that are unique to a distributed environment.

This paper will describe a number of commonalities and differences between the stand-alone environment and the distributed environment. In doing so, it will raise a number of distributed system issues that must be resolved in order to satisfy the needs of an open distributed system.

Finally, this paper will also articulate the need and process for building an open distributed system so that it behaves like a system rather than a set of disparate components.

1. Introduction

The most significant characteristic of computing in the 90's will be the evolution of distributed computing environments. This sentiment is echoed by people involved in development, research [Mul89a] and management. This gradual change is predicated on the ability of the industry (that is, both research and development organizations) to develop at least one architecture that can satisfy several key needs: increasing demand for interoperability among systems, support for the client/server distributed application development model, and the ability to account for the characteristics and challenges that are inherent in a distributed computing environment.

Such an architecture would be considered appropriate if it were applicable to a wide variety of physical topologies (that is, a variety of hardware platforms), to a wide variety of management topologies (such as centralized or decentralized), and to a wide variety of software topologies (such as operating systems and development languages). The

architecture would be considered appropriate if it were able to adapt to changing requirements, such as the ones just mentioned, over time. And finally, the architecture would be considered appropriate if its components behaved predictably and coherently, that is, if they behaved like a system rather than a set of disparate pieces. Now let us take a closer look at some of the needs that must be accounted for in this architecture.

1.1. Interoperability

Interoperability is usually a concern only in an environment that contains dissimilar fundamental hardware or software services (that is, a heterogeneous environment). This, in a nutshell, is the difference between a distributed operating (homogeneous) system and a distributed networking (heterogeneous) system. A distributed operating system is not preoccupied with interoperability because homogeneity is a fundamental characteristic of its environment. Each system that is cooperating in the distributed operating environment runs the same set of fundamental services. Although there is occasionally a need for homogeneity in some environments, this is not typical.

The need for a distributed networking system, on the other hand, is typically based on the need for some degree of generalization. For example, in many organizations computing resources are acquired by satisfying the needs of the local department. This typically is done with little or no attention to the prospect of integrating these resources with other departments. This results in a diverse set of hardware and software services. And even in the case where this integration is anticipated, it is reasonable to assume that the specific requirements of each department will lead to the accumulation of dissimilar hardware and software services. However, specialization too can drive the need for interoperability. That is, the demand for a special device or service (for example, a high-capacity disk drive, a high-performance graphics workstation, or a multiprocessor operating system) is what drives a variety of organizations to solve that need. For example, the demand for a low-priced multiuser workstation has led to many different implementations. Inherently, these implementations are not interoperable unless additional services are introduced.

Put another way, the need for interoperability is great because there are only a few situations that demand a homogeneous rather than a heterogeneous environment. A distributed computing environment architecture needs to support the requirement of interoperability because it is assumed that most environments have varying degrees of dissimilar fundamental hardware or software services.

1.2. Client/Server Model

One of the most widely used models in the development of distributed applications is the client/server model (see Figure 1). This model is based on the premise an application, referred to as the client, makes a request of a particular service, referred to as the server. Although this is the canonical definition of the client/server model, it needs to be extended.

The new definition is based on the premise that the client receives the benefit of the service. Therefore, the model is defined by the application that receives a service rather than the application that initiated it. Each service has a consumer and a supplier as well as an invoker and a responder. The current definition can be thought of as a pull-down

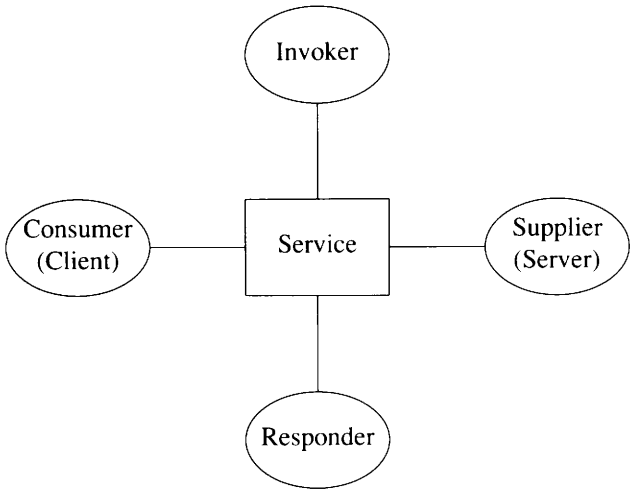


Figure 1: *Extended Client/Server Model*

approach to services. The invoker is associated with the consumer (the client) and the responder is associated with the supplier (the server). This would typically be the case for a distributed file server, a remote procedure call, or a distributed name service request. However, for other applications that fit into the client/server model, the invoker is associated with the supplier and the responder is associated with the consumer. This is the case for many distributed system management applications [Joh89a]. For example, imagine an application which performs backups of systems on the network. This task, in many cases, would be initiated centrally by a system administrator and be executed remotely on all of the managed systems. In this case, the client is the consumer of the service, although the request was initiated at the supplier.

1.3. Defining a Distributed Environment

Logically, people understand that the difference between a distributed computing environment and an environment which is not distributed (either a stand-alone system or a collection of stand-alone systems) is that related code can be executing on more than one system in the network. If the code is not related, then two systems executing completely unrelated code on two different systems would not constitute a distributed environment. Additionally, it is not a requirement that related code execute on more than one system, but it needs to be possible. That is, an extensible architecture is defined by assuming that the components of a distributed environment will be distributed. The degenerate case occurs when all of the components are on the same system. The reverse is true of services developed for a stand-alone system, which assumes that all of the components are local.

Ideally, this functionality should have a desired amount of transparency. This is somewhat different from the need for total transparency in that the users of the system should have the ability to require (manage) non-transparency if it is appropriate. For example, in a distributed environment which supports load balancing (the ability to maximize the usage of idle CPUs, or conversely, the ability to avoid overwhelming any particular CPU), it is desirable to have the ability to specify which systems are and which are not participating in the load-balancing algorithm. Without this capability, the load-balancing

software would transparently (indiscriminately) choose from any of the systems on the network.

Architecturally, the approach to defining a distributed computing environment is to consider the characteristics which distinguish a distributed environment from a stand-alone environment and then engineer a solution which is responsive to those characteristics. This solution must take into account any challenges that are associated with a distributed environment.

1.3.1. Characteristics

Following is a list which articulates some of the characteristics which help distinguish a distributed environment from a stand-alone environment.

- Physical separation

  Through the use of both local area network (LAN) and wide area network (WAN) hardware capabilities, the computing environment can be physically dispersed and potentially extended over a large geographical area.

- Scalability

  The number of software components such as a file server, a name server, or a time server and the number of hardware processing entities such as a multiuser time sharing system, a printer, or a disk server can range from small to quite large. Realistically, a large distributed environment might have tens of thousands of components and processing entities.

- Administrative autonomy

  Especially as the distributed environment grows, as defined by both physical separation and scale, there is likely to be an increasing need to support a variety of administrative policies. Each administrative domain needs to be able to dictate administrative policies (for example, based on national language, organizational requirements, or geographical restrictions) independent of other domains.

- Heterogeneity

  Again, especially as the distributed environment grows, there is likely to be an increasing variety of both hardware and software capabilities. This variety can be realized in two ways: one, the preservation of the current installation base, and two, the anticipation of solutions which are delivered by different organizations. The need to support or coexist with the current installation base (for example, hardware and operating systems) is not just a marketing requirement but an acknowledgement of how new technologies are typically integrated: by evolution. In addition, it is only reasonable to expect that there will be more than one supplier of either hardware or software for any particular service. All of these issues help to drive the distributed environment to be increasingly heterogeneous.

1.3.2. Challenges

Following is a list which articulates some of the challenges which help distinguish a distributed environment from a stand-alone environment.

- Naming

  As in a stand-alone system, the ability to retrieve information associated with a name (object) is a primary function of the dis-

tributed system. The name could be associated with, for example, a file, a device, or an application. In a distributed environment, however, this information needs to be available from anyplace in the network. Additionally, a name should reveal the same information regardless of where the name is referenced.

- Security

  Again, similar to a stand-alone environment, the need to control access to resources is a primary function of the system. However, a distributed environment introduces new vulnerabilities in managing access to a resource. For example, information typically flows across a physical medium (like an Ethernet) which cannot be assumed to be trustworthy. That is, a user can maliciously or unintentionally read or change data as it propagates around the network. Additionally, in many cases a resource (such as a file, a printer, or a disk) is not physically close to the people who would like to manage it; therefore, security can not be assumed because of physical proximity.

- Manageability

  As related to the characteristic of administrative autonomy, portions, or domains, of the network will be managed in different ways. The distributed computing environment architecture needs to support a variety of management policies without imposing undue restrictions or constraints on these management implementations. Additionally, in order to work effectively in a distributed environment, the management functions must have the same qualities as the fundamental distributed system technologies. For example, they must execute in a heterogeneous environment and scale to large networks.

- Indeterminacy

  Although the ability to have true parallelism is a trademark of an effective distributed environment, it is also the root of one of the most significant challenges, that of indeterminacy. Indeterminacy can be manifested by inconsistency of multiple copies of data, by asynchrony (for example, related functions running in parallel on separate systems), by latency (that is, functions that may succeed, temporarily fail, or completely fail at varying and unpredictable rates), and by event-ordering (that is, the ability to accurately determine the order of events as they occur on separate systems).

2. Motivation for Distributed Computing

One explanation for the necessity of distributed computing can be expressed by answering two questions. Why do we need to think in terms of being distributed? And what are the benefits of being distributed?

2.1. Why do we need to think in terms of being distributed?

The main reason we need to think in terms of being distributed is because we exist in a distributed world that is becoming increasingly distributed [Lam81a]. That is, by definition a network is a collection of connected systems (such as, multiuser workstations, personal computers, and mainframes). Except for the case in which there is one and only one storage mechanism, data is stored on many different systems

in many different ways, and the users of the systems either want to or are required to share this information.

Two factors increase the likelihood of distribution.

- The user community prefers to support open systems (that is, standard interfaces and protocols) as a means of increasing the opportunity to share information and services.
- The overall decrease in costs of hardware components and the increase in price-performance ratios allows more people to become investors in low-cost personal computers and mid-range computing systems.

It should be noted that people have been thinking in terms of being distributed for quite some time. Until recently, however, the state of technology, or lack thereof, prevented distributed environments from becoming more prevalent.

2.2. What are the benefits of being distributed?

In the previous section a distributed computing environment was defined in terms of a set of characteristics and challenges. If these characteristics and challenges can be addressed in a distributed systems architecture, potential advantages can be realized in the resulting distributed computing environment. They include

- Resource sharing

  Resource providers reach a wider consumer audience in a distributed computing environment than in a stand-alone environment, and conversely, resource consumers can choose from a wider selection of providers.

- Performance

  Similar to a multiprocessor system, a distributed environment can execute a number of operations simultaneously. The key benefit to performing tasks in parallel is increased throughput – through the advantage of decreased latency in processing separable functions.

- Availability

  By taking advantage of the parallel nature of a distributed environment, the availability of services can be increased because there is potentially no single-point of failure. That is, one instance of a service can continue to run in the event that another instance of the same service cannot due to, for example, heavy loading, temporary network failure, or complete failure of the system it is running on.

3. Developing the Distributed Computing Architectural Framework

The question of how to define such an architectural framework remains. One way is to resolve two general issues, first, to define the paradigms that are essential to support the development environment, and second, to define a process which will produce instantiations of these paradigms. Let us take a look at each of these issues.

3.1. Necessary Distributed Computing Paradigms

Although many differences have been articulated between the stand-alone and distributed environment, there is some set of issues common to both. That is, in all programming environments, some set of paradigms must exist to support the development of applications. Historically, this has been done in the stand-alone environment by many different organizations in many different ways. However, at the root of these development environments is the support for a set of fundamental programming paradigms. Following is a description of how these paradigms have been actualized in stand-alone systems and an indication of how they would be applied to a distributed system.

- Execution space

  On almost all operating systems, the execution space is defined as the process. For multiprocess operating systems, the local execution space is a process and the global space is the collection of all processes. The focal point of administration for the system is determined by the current state of a process or set of processes.

  In the distributed environment, the local execution space can be thought of as a host (system) and the global space as the collection of all systems. Components of a distributed application usually are instantiated as a process, or set of processes, on some particular system. Therefore, the focal point of administration in the distributed environment is determined by the current state of a system or set of systems.

- Physical data exchange and interexecution space communication

  In the stand-alone environment, the standard means for execution spaces to exchange data is through main memory. This can be achieved in many ways. For example, a single process might use a local procedure call which would take advantage of a common address space, and multiple processes might use an interprocess communication (IPC) mechanism which would take advantage of some memory-management technique such as shared memory.

  In the distributed environment, the standard data-exchange mechanism is the network (for example, the Ethernet). Given that the execution is defined across multiple systems, the data-exchange mechanism needs to support inter-procedure communication paths that will probably occur over the network. Therefore, it logically should behave similarly to the local procedure call except that the sending and receiving functions may be on different systems.

- Primary stable data storage

  A key mechanism for any type of non-transient system is the ability to store information that persists longer than the duration of a process. This is commonly referred to as stable storage. The type of device used to satisfy this need (whether it be a floppy, a tape, a hard disk, or a write once optical disk) is unimportant. In the stand-alone environment, the stable storage mechanism is typically a local file system.

  In the distributed environment, a similar type of technology is needed for the same reasons. In this case, a file system which allows a consistent view of files from anywhere on the network is needed. For the distributed environment, the concepts of the local file system need to be extended to support the same operations, and provide similar benefits and guarantees, across many systems in the network.

- Concurrent programming

  The idea of a component in the computing system working on more than one thing at a time is very appealing. The hope is that such a scheme would allow more tasks to be completed. This idea is referred to as concurrent programming and it has been accomplished in a number of ways. Multitasking uniprocessor systems achieve concurrency by swapping out processes that are waiting for another event to complete (or because the process has used up some (pre)determined time-slice) and swapping in another process that can execute. Multiprocessor systems achieve concurrent programming by actually running code on more than one processor at the same time. Another alternative is time-slicing activities within a given process. By allowing a single process to have multiple execution code segments, the segment ready to execute can continue. Therefore, other segments which are potentially waiting for the completion of some event (such as input or output) can be suspended.

  A network is inherently a concurrent programming environment because autonomous computing systems can be independently running a process on their own dedicated processor(s). Although this concept is applicable to a stand-alone environment, concurrent programming becomes a necessary component in the distributed system. Typically, the server is required to service more than one client request at a time. Without this mechanism, all requests to the server would be handled sequentially and would not satisfy certain requirements of the distributed environment – such as the need to scale to support large numbers of systems.

- Event ordering

  In the stand-alone environment there are at least two types of event ordering. One type occurs when a process is pending on a system or library call to complete. For example, a process has made some synchronous request and waits in a pending state until that request has been satisfied; the process could be waiting for some I/O to complete, for a semaphore to clear, or for some other locked resource to become available. Another type occurs when a process needs to make an ordering decision based on the time-stamp of some well known event. It typically does so by checking a time-stamp associated with a file or an object in a file (such as a database). In the stand-alone environment, a decision based on a time-stamp is trustworthy because there is only one provider of the time within the environment – the local operating system.

  In a distributed environment each system has its own representation of time; therefore any decision based on the time-stamp associated with events (such as file modification) from different systems is suspect. Hence, a mechanism is needed to ensure that applications can continue to make decisions based on the order of events within the distributed environment.

- Stable data storage for execution space

  A process normally makes reference to data in two storage areas, temporary and permanent. Temporary data can be thought of as anything stored in memory (physical, virtual, or shared) and permanent data as anything stored in stable storage. In any event, things stored in stable storage are data constructs that need to

1 This does not rule out the possibility of storing temporary data in stable storage.

persist longer than the duration of the process;[1] such as, administrative, transaction, configuration, or log information.

In a distributed environment, there is also a need to access data that persists longer than the duration of any particular process or system. Such data also needs to be consistently available to any process on any system in the network.

- Secure access to objects

  In the stand-alone execution space, there are usually two types of objects that need an access control (security) mechanism: process objects and file objects. The local operating system is usually responsible for process objects (such as shared memory or IPC channels) and the local file system is responsible for file objects.[2] The goal is to control access to a particular resource.

  In the distributed environment, there is a similar need to control access to objects. However, in this environment the objects may be dispersed or even migrate among many systems. Additionally, there are new security issues that do not exist in the local environment. For example, the location of a resource may not be known until it is referenced or a resource will be accessible by users who are not defined in the local system.

These paradigms are available for most application development environments. Their implementation can take many forms, but their existence is a necessity. The distributed computing environment is just another example of an application development environment which needs to provide solutions for these paradigms. However, there are characteristics and challenges specific to the distributed environment.

3.2. A Process For Defining a Distributed Computing Environment

Most people would agree that the prospect of developing a distributed computing environment architecture is a formidable task. The effort required to meet such a diverse set of needs, requirements, and challenges makes it difficult to understand the entire problem. However, the issue of diversity also leads to an appropriate method for addressing these issues. That is, given that the problem area is diverse, it makes sense to formulate a potential solution by reviewing and incorporating a diverse set of perspectives.

This is exactly what the Open Software Foundation (OSF) did: it used an open process to design the architectural framework as part of its distributed computing environment request for technology effort. The evaluation team reviewed many distributed system architectures[3] and solicited the opinions of leading developers, researchers,[4] and users in an effort to push forward the state of open distributed systems. The key aspect of this process was to use these opinions in influencing the

2 It should be noted that in some operating systems, another object that requires access control, and is not explicitly either a process or a file object, is a device – such as a disk drive, a floppy drive, or a network controller. In this case, the device is usually accessed through a special file but is handled by the operating system.

3 For example, A Distributed Systems Architecture for the 1990's [Lam89a], Network Computing Architecture [Apo89a], Advanced Networked Systems Architecture [APM89a], Open Distributed Processing [ISO90a] and The Digital Distributed System Security Architecture [Gas89a].

4 Dr. Andrew Birrell at the Digital Equipment Corporation Systems Research Center, Professor David Cheriton at the Stanford University Computer Science Department, Dr. Paul Mockapetris at the University of Southern California Information Sciences Institute, Dr. Sape Mullender at the Centrum voor Wiskunde en Informatica, Dr. Roger Needham at the University of Cambridge Computer Laboratory, Dr. Michael D. Schroeder at the Digital Equipment Corporation Systems Research Center, and Dr. Peter J. Weinberger at AT&T Bell Laboratories.

design. In this respect, the design is in fact a conglomeration of a diverse set of viewpoints and requirements.

4. Putting It All Together

There are three requirements for an appropriate distributed computing environment architecture. First, there must be mechanisms to support the needs of the fundamental application development paradigms. Second, these fundamental technologies must account for the characteristics and challenges which are unique to a distributed environment. And third, these technologies must be incorporated in such a way that they have the qualities of a well formed system.

4.1. Fundamental Technologies

As stated above, a set of fundamental programming paradigms must exist to support the development of applications, in this case distributed applications. Following are the necessary technologies which can support these paradigms in the distributed environment and a brief description of their capabilities.

- Physical data exchange and interexecution space communication

  Remote Procedure Call

  The remote procedure call (RPC) provides programmers a familiar programming model by extending the local procedure call to a distributed environment. RPC maintains the useful aspects of the local programming model while handling purely distributed issues such as operation semantics (for example, at-most-once), server selection (binding), and communication or server failures. RPC also provides a convenient and consistent mechanism for specifying the interactions between components of a distributed system.[5]

  In addition to basic RPC features, such as supporting a variety of data types and transport independence, the remote procedure call needs to support extended features such as

  - Context handles, allowing a server to reclaim resources when either the communications or client fails
  - Mutable pointers, allowing pointer-handling capabilities that mimic local pointer semantics
  - Multiple language bindings, allowing applications to be implemented in different programming languages
  - Orderly quit, allowing the application to cancel outstanding requests
  - National language support, allowing data types from multibyte character sets to be supported

- Primary stable data storage

  Distributed File System

  Distributed file system technology provides the ability to access and store data at remote locations. It extends the local file system model to a distributed environment.

  To do this, it needs to

5 It should be noted that there is some debate about the difference between procedure- and message-oriented systems. I believe, as Lauer and Needham [Lau78a] stated, that these categories are duals of each other. My extension to their theory is that it applies not just to local operating systems but to distributed systems as well.

- Provide transparent access to both local and remote files
- Support local functionality, such as file locking and sharing, when accessing remote files
- Allow users to address files with the same pathname from anywhere in the system, regardless of the computer that is being used
- Provide high availability to all accessible data resources
- Serve a very large number of concurrent requests with good performance
- Work in a wide area network configuration
- Provide secure access to local and remote files and directories
- Ensure that the logical view of a file is independent of its physical location.

- Concurrent programming

  Threads

  Threads provide a useful model for structuring applications to allow them to exploit parallelism. To a large degree, this is accomplished by taking advantage of multiple processors in a multiprocessor system. In a single processor system this is accomplished by allowing the application to overlap operations. That is, when one thread is blocked (for example, because it is waiting for an I/O call to complete) another thread in the same process can be executed.

  This technique is useful in client applications and is most important in servers, which must handle requests from multiple clients concurrently. For example, in a typical client/server scenario, a server program can be processing the data from a read request while it is waiting (that is, blocking on the next read request) for another chunk of data to be received. Threads use re-entrant programming techniques that allow simpler designs than other parallelism alternatives such as multiprocess implementations using shared memory, or explicit asynchronous operations.

  To do this, threads must support features such as:

  - Mutexes: a way to synchronize threads access to shared data
  - Condition variables: used to develop race-free multithreaded applications (that is, coordinate threads within an application)
  - Alerts: used to provide graceful and reliable thread termination

- Event ordering

  Distributed Time Service

  The purpose of a time service is to synchronize the clock of a local computer with Universal Coordinated Time (UTC), as well as clocks of other computers on the network. Distributed services that compare dates generated at different computers require some mechanism for determining the appropriate order of events as they occur across different systems within the network. Distributed file systems and authentication are two examples of such services.

  The network consists of local system clocks that are divided into clients and servers. Clients take the time from servers, whereas

servers synchronize with each other. Because any single server can fail or be inaccurate, a requestor, either client or server, needs to ask the time from a number of servers. Additionally, the time service needs to handle error situations like faulty servers and a temporary breakdown of the network as well as avoiding instabilities that arise from loops (that is, the same two servers asking time from each other) and topology changes (for example, the addition of new servers).

- Stable data storage for execution space

  Directory or Name Service

  The purpose of a naming or directory service in a distributed computing environment is to map user-oriented names to computer-oriented entries in a special-purpose distributed database that describes the objects of interest. Objects contain information for such things as organizations, persons, groups, organizational roles, computers, printers, files, processes, and application services and their interfaces. The clients of the directory service span a wide range from other services comprising the distributed environment, such as remote procedure call and management programs, to applications, such as print spoolers and mail services. The basic added-value of a directory service is its ability to provide location- and routing-independence. It allows objects to be addressed by human readable names, regardless of the locations of the directory client and of the named object, or of the communications path between the two.

  The directory service needs to support features such as:

  - White pages: the ability to perform straightforward name-to-entry lookups
  - Yellow pages: the ability to perform lookups based on object attributes
  - Link or alias service: the ability to perform name-to-different-name mappings
  - Group service: the ability to map a single name into a set of names

- Secure access to objects

  Security Services

  Security consists of authentication of entities (or principals) in an open system, authorization of principals for using resources, and guarantees of integrity and privacy of messages sent over the network. Authentication in an open network context is the verification of a given principal's identity (such as a user or service). In a network system, authenticated communication is necessary because messages are subject to forgery and therefore cannot necessarily be trusted. Authorization is concerned with granting privileges with respect to resources, such as access to files.

  Because messages on an open network can be read and forged, a way of determining whether a message from a given principal arrived intact and unaltered is needed. This assurance provides message integrity based on the identity of the invoking principal. While guarantees of message integrity are an absolute minimum requirement, some applications require that the confidentiality of the data be guaranteed as well. This is done by encrypting the message contents, which provides message privacy. It should be noted that although authentication and authorization are essential

in a distributed computing environment, the use of security mechanisms is expensive (in terms of overhead applied to each request); therefore different levels of security may be appropriate for different applications.

4.2. Fundamental Technology Design Mechanisms

As pointed out earlier, one of the keys to understanding the makeup of open distributed systems is to understand the characteristics, challenges, and benefits that distinguish a distributed environment from a stand-alone environment. In addition a number of fundamental paradigms are necessary to support the development of applications in both a stand-alone and a distributed environment. The task, then, is to design services that support these paradigms in a way that accounts for the issues that are unique to the distributed environment.

Following is a list of design goals that should be considered in the development of the fundamental technologies, and for that matter, in the development of distributed applications in general. Listed with each goal are some of the common mechanisms that should be used to achieve that goal.

- Autonomy

  The principle behind autonomy within a distributed environment is that each site has its own set of execution and management policies. Therefore, it is difficult, if not impossible, to predict which policies will be acceptable for a given site. In fact, each site may have a number of administrative domains that have varying degrees of policy conformity. Moreover, those policies typically change over time.

  Therefore, with respect to autonomy, the technology developer must strive to meet the following design objectives:

  - Minimize the number of predetermined execution and management parameters
  - Maximize the number of system parameters that can be reconfigured

- Availability

  A resource accessible via only one mechanism would become unavailable if the mechanism failed. Therefore, the key to increasing the availability of any resource in a system is to increase the probability that no single point of failure would prohibit access to that resource.

  Hence, with respect to availability, the technology developer must consider the following design objective: there must be alternate mechanisms to access valuable resources. This typically is done by using either replication or duplication techniques. The key distinction between these mechanisms is that replication algorithms attempt to coordinate copies of the information automatically, and duplication algorithms do not. For example, a directory service attempts to ensure that all references to any particular piece of data reveal the same result independent of where that data is referenced.

- Hardware and operating system independence

  Hardware and operating system independence is important in the distributed system because applications need to be ported to, execute in, and interoperate in a heterogeneous environment. The key to providing both hardware and operating system indepen-

dence is to not rely on any features, side effects, or traits that are specific to any particular system.

This would include

- Services that only exist in certain environments. For example, the availability of a broadcast mechanism is endemic to a local area network. Therefore, reliance on its availability might prohibit a service from working in a wide area network.

- Local services that parallel distributed fundamental services. Often, services within the local operating system parallel services in the distributed system. An example is the use of an interprocess communication mechanism versus a remote procedure call. In general, a service should be developed by assuming that all resources are remote. Otherwise, reliance on a local mechanism might impede the ability for this service to execute remotely or execute on a dissimilar system.

- Indeterminacy

  A distributed environment includes a certain set of execution characteristics that are not typically provided in the stand-alone environment. For example, if there are problems accessing main memory in the stand-alone environment, the system probably needs to be rebooted to correct the problem. In the distributed environment, however, there are a variety of problems that could temporarily affect access to the network.

Therefore, with respect to indeterminacy, it is necessary for the technology developer to make the following assumptions:

- All access to the network may result in failure and a subsequent access may succeed; this is typically handled by using retry mechanisms.

- Any access to the network may not relinquish control to the calling program; this is typically handled by using threads, polling, or time-out mechanisms.

- Scalability

  One of the most important objectives of a distributed service is to ensure acceptable operation as the number of users increase. This growth typically occurs because more systems are added to the current environment. In general, the issues associated with scalability tend to dictate what mechanisms are appropriate for a distributed application. That is, if the target audience for a given application is known to be small, there are a variety of approaches that can be used to solve a particular problem. When the target audience is known to be large, usually only a few methodologies will be appropriate. This is analogous to determining an appropriate search algorithm. When the sample is small, there is practically no difference in using a binary or a linear search. However, as the sample size increases, the appropriateness of a linear search decreases. Therefore, it is encumbent upon the technology developer to assume that the demand on the application will always be large. In general, mechanisms that work when the sample size is large will work when it is small. The reverse, however, usually is not true.

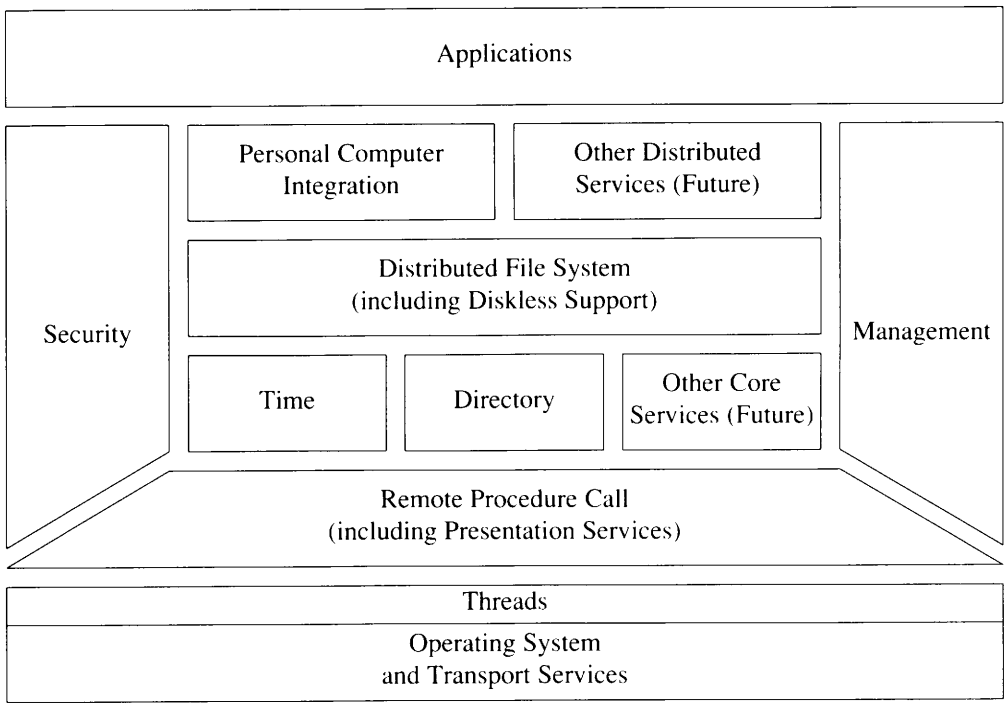


Figure 2: *OSF Distributed Computing Environment Architectural Framework*

4.3. Requirements of the Distributed System

Now the issue is how to put all of this information together in a way that will lead us to the appropriate architectural framework and the appropriate set of mechanisms to support this framework. There are two key goals in making this happen: define a minimal set of mechanisms and integrate them into a system.

The core of the distributed system needs to be defined as a minimal set of fundamental mechanisms. A mechanism is fundamental if it is required for the development of other distributed applications (such as distributed system management or distributed development tools). The core is expected to be that set of mechanisms which support the previously described application development paradigms. Figure 2 depicts the framework which defines this minimal set of fundamental mechanisms [OSF90a].

Several issues should be considered when reviewing the information in Figure 2.

- It is not an interface definition. For example, it is explicitly expected that, for example, distributed applications will make direct calls to the remote procedure call or threads interface(s).
- Threads is shown close to the operating system because it is an essential distributed system service but may be implemented in the local operating system
- The Other Core Services area is intended to be a repository only for other fundamental technologies, that is, those necessary for the development of other applications. An example is event notification.

The fundamental services such as file, time, directory, remote procedure call, management, and threads provide a layer between the

application and the operating system and network. These services are necessary for the development of applications in the distributed environment. The security, remote procedure call, and management services form the architectural infrastructure. The remote procedure call is the application development communication mechanism, and the security and management services are a part of all of the fundamental services. The directory service is central to the whole architecture because it provides the mechanism in which all objects are referenced and located.

The need to put this distributed environment together to create a distributed system, rather than a disparate set of technologies was articulated by Lampson, Schroeder, and Birrell. Basically, there needs to be one set of interfaces to the fundamental technologies. For example, there is only one set of exposed interfaces to the remote procedure call, to threads, or to the naming/directory service. Additionally, the fundamental technologies themselves need to use these same interfaces. For example, all of the fundamental technologies should use the threads technology to achieve concurrent programming benefits. Furthermore, all of the fundamental technologies should use the remote procedure call technology to achieve interprocess communication across systems. The information in Figure 3 indicates the level of integration that needs to exist in the fundamental technologies to support the distributed computing environment architecture.

	Threads	Time	Security	RPC	Directory
Threads	–	n.a.	n.a.	n.a.	n.a.
Time	x	–	x	x	x
Security	x	x	–	x	x
RPC	x	x	x	–	x
Directory	x	x	x	x	–
File	x	x	x	x	x

Figure 3: *Fundamental Technology Integration Matrix*

Legend:

x denotes an explicit integration point; for example, the Security, RPC, Directory, and the File services are integrated with the Threads service.

n.a. denotes a non-applicable integration point; the threads technology does not integrate with the other services because it executes only within a process on the local operating system (and therefore does not require the functionality provided by the other distributed services).

– denotes an intersection of a technology with itself.

4.4. The Makings of a System

Without this level of integration, the distributed environment becomes less predictable and more complex. This is true from both a statistical and an intuitive sense. Statistically the complexity of the environment increases with the number of interfaces because there are more potential outcomes during the execution of a process. For example, if each fundamental service had its own interface(s) (represented by n) to support threads, time, security, remote procedure call, and directory ser-

vices, the total number of interface invocations would (more likely) be on the order of $n*n$ rather than just n. Intuitively, the environment would be more complex because there are more potential failure points, debugging paths, and administrative issues. Additionally, it would be more likely that this collection of components would grow apart in time because each would be following its own evolutionary path rather than evolving as a system.

With this level of integration, the fundamental technologies have the qualitative characteristics and benefits of a system. This is analogous to the local operating system. That is, the local operating environment is viewed as a system which provides a common set of fundamental services, such as memory management, interprocess communication, and interprocess synchronization. Without these common services, the application developer is burdened with the task of providing some level of support for these mechanisms. Therefore, by designing an integrated system, the application developer is relieved of many programming complexities. For example, the RPC service will automatically use the security service to ensure both message integrity and privacy. It also will automatically use threads to ensure maximum throughput for multiple requests from a single process. Without an integrated system, the application developer needs to explicitly call all of these services to obtain the benefits that they provide. This additional programming makes it difficult to write distributed applications.

It should be noted that the comprehensive integration of the distributed technologies, and the adherence to a standard set of interfaces to them, is what distinguishes this type of system (that is, this architecture and the technologies that support it) from other distributed networking systems.[6] These are the attributes that make the system coherent and allow the system to account for the characteristics and challenges that distinguish a distributed environment.

5. Conclusion

We are compelled to think about and solve problems associated with distributed computing because we live in a distributed world and a distributed computing environment offers many potential benefits. In order to effectively deal with the complexity of a distributed environment, we need to define an architecture which accounts for a certain set of distributed environment issues. For example, it needs to support useful distributed application development models (such as the client/server model) and account for interoperability issues (such as heterogeneous software and hardware platforms). The architecture needs to be designed by considering and solving for key characteristics and challenges that are endemic to a distributed computing environment. Additionally, there is historical evidence which suggests that there are a certain set of necessary application development paradigms that apply to both the stand-alone environment as well as the distributed environment.

While it is certainly true that no one architecture will satisfy all of the needs of all of the people, the architecture developed by the Open Software Foundation is an example that satisfies the needs of many diverse organizations and many of the architectural demands of a distributed environment. To realize the potential of a distributed computing

6 Distributed operating systems have this level of integration among the fundamental services; however, this type of system is usually only applicable to a homogeneous distributed environment.

environment will require supporting such an architecture as well as providing implementations which satisfy two key requirements. One, the mechanisms that support the fundamental application development paradigms must account for the characteristics and challenges that are unique to a distributed environment. And two, these mechanisms need to be integrated with each other so that they behave like a system rather than a collection of disparate technologies.

6. Acknowledgements

I would like to especially thank Dr. Walter Tuvell for his keen insights into shaping the direction of this paper and Ann Hewitt for her valuable editing comments. Their input has significantly improved the quality and clarity of the paper. I would also like to thank Doug Hartman, David Lounsbury, and Jon Gossels for their reviewing comments. And finally, I would like to pay special tribute to the rest of the Open Software Foundation distributed computing environment evaluation team: Richard Mackey, Norbert Leser, Dietmar Fauth, Chi Shue, Jennifer Steiner, and Todd Smith. The efforts of this team made this paper possible.

References

[APM89a] APM, *Advanced Networked Systems Architecture*, Architecture Projects Management Limited, Cambridge, United Kingdom (1989).

[Apo89a] Apollo, *Network Computing Architecture*, Apollo Computing Inc., Englewood Cliffs, New Jersey (1989).

[Gas89a] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson, *The Digital Distributed System Security Architecture*, 1989.

[ISO90a] ISO, *Basic Reference Model of Open Distributed Processing ISO/IEC JTC1/SC21/WG7*, International Standards Organization, The Netherlands (1990).

[Joh89a] Brad C. Johnson and David M. Griffin, "Remote System Management in Network Environments," pp. 29-35 in *Digital Technical Journal*, Maynard, Massachusetts (June 1989).

[Lam81a] B. W. Lampson, M. Paul, and H. J. Siegert, "Motivations, objectives and characterization of distributed systems," pp. 1-19 in *Distributed Systems: Architecture and Implementation*, Springer-Verlag, New York (1981).

[Lam89a] Butler W. Lampson, Michael D. Schroeder, and Andrew D. Birrell, *A Distributed Systems Architecture for the 1990's*, 1989.

[Lau78a] H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," in *Proceedings of the Second International Symposium on Operating Systems*, IRIA (October 1978). reprinted in *Operating Systems Review* **12**(2) April 1979, pp. 3-19

[Mul89a] Sape J. Mullender, "Introduction," pp. 3-17 in *An Advanced Course on Distributed Computing*, ACM Press, Fingerlakes, New York (1989).

[OSF90a] OSF, *Distributed Computing Environment Request for Technology: DCE Framework – Preliminary Position Paper*, Open Software Foundation, Cambridge, Massachusetts (1990).

Integration Mechanisms and Communication Architecture in AxIS

Dario Avallone
Roberto Dottarelli Felice Napolitano
Ingegneria Informatica, Rome, Italy
dario@engrom.uucp

Abstract

This work presents the communication architecture of the configurable Software Engineering Environment (SEE) AxIS. The architecture goal is to allow the integration and the support of user-defined production tools in a distributed environment.

1. Introduction

The experience has pointed out that the software life-cycle model, the analysis and design methodology and the tools used to develop software products may differ according to the project needs and the organizational context.

The main goal of **S**oftware **E**ngineering **E**nvironments (**SEE**s) is to supply a synergic combination of methods, techniques and tools for supporting the software production [Jef88a].

AxIS is a research project for the construction of a *configurable* SEE [Pen88a].

This paper introduces the architectural and technological solutions to the problems of integration and communication among components in an AxIS *instance*. The first paragraph summarizes the architecture of the AxIS environment. The following paragraphs describe the communication system architecture and the integration policy.

2. AxIS Overview

AxIS is a configurable environment (i.e. meta-environment) that can be instantiated to build an environment for supporting specific software projects. The AxIS architecture can be seen as a virtual machine of five software layers (Figure 1).

The two lower layers are the AxIS host environment representing the basic software platform. This consists of the UNIX Operating System (System V or BSD 4.3), Object-Oriented DBMS (ONTOS by Ontologic), NFS (Sun Microsystems) and the X-Windows system.

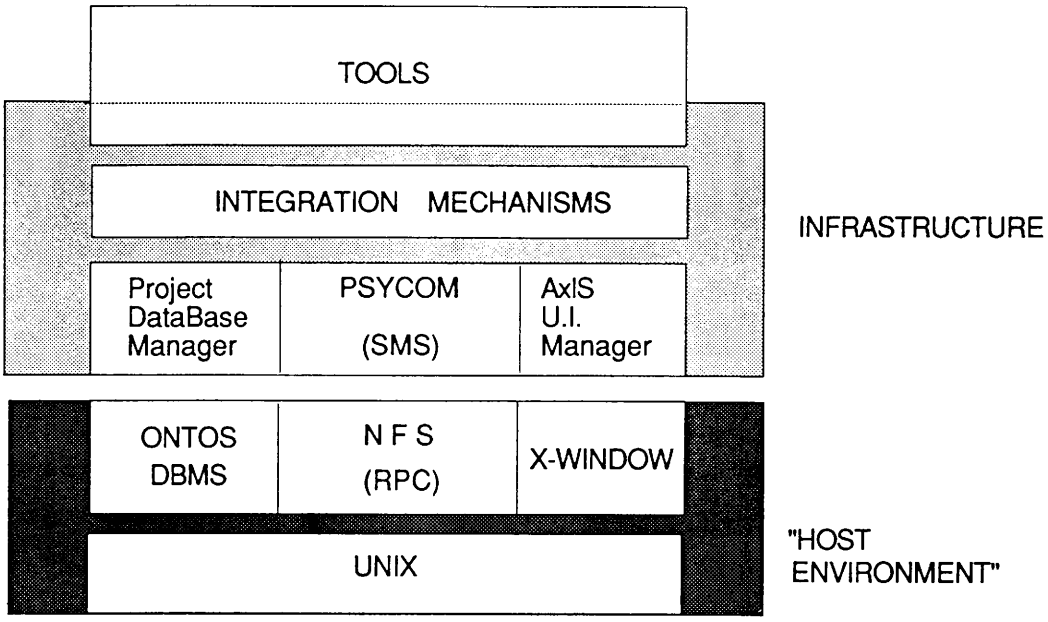


Figure 1: *AxIS Virtual Machine Architecture*

The middle layers are named the *infrastructure*. This part is the core of AxIS and can be configured to instantiate a specific environment through specialized tools called *configuration tools*. To build an instance, in this context, means to support a specific life cycle model and design methodology [Cor90a].

The top-most layer is a set of different tools for supporting a specific software project. We call these tools *production tools*.

Some of these tools are predefined, while the others depend on the activities stated in the chosen instance. The tools not predefined can be built through the AxIS provided toolkit [Cec89a, Ava89a, Nac89a, Roc89a] or imported by *integration mechanisms*.

The predefined tools directly communicate with the infrastructure components and provide a fixed set of operations, whose behavior depends on the instantiated environment. This kind of tool [Weg90a] can be called *polymorphic*. Some example of them are a *Project DataBase Browser* and a *Cooperative Support Tool* [Flo86a, Mal88a].

AxIS has a distributed architecture in that the tools and the infrastructure are organized as a set of interacting processes located on the different networked machines. The execution of an activity is carried out following a service request according to the client-server paradigm. Further, AxIS supplies a supervision system, named **PSYCOM** (**P**rocess and **SY**stem **CO**ntrol and **M**anagement), for controlling and supporting all the phases of the chosen life cycle model.

Finally, AxIS provides a project Object-Oriented DBMS for managing all the Project Data Bases of the generated instances.

3. Communication System Architecture and Integration Policy

PSYCOM is composed of two parts:

- The first one is devoted to software process management [Arm89a];

- The second one, called Service Management System (**SMS**), is in charge of the communications between the production tools and the infrastructure.

This paper details *Service Management System components and the integration mechanisms.*

SMS allows the transparent access to the services of the environment.

These services depend on the defined instance. For example, suppose that the entity-relationship diagramming is foreseen in a particular project. In order to draw E-R diagram a tool is necessary and the *service* "diagram|ER" will be made available in the AxIS instance.

Our aim is to ensure the independence between the *infrastructure* and the particular set of tools which provides the services.

The Service Management System architecture has two main features:

- A user defined allocation of the tools in the environment.
- The centralized access control to the services.

The latter lets have the automatic monitoring of the system workload. This means that there is a master component that receives all service requests and schedules them according to the *system state*. The system state is determinated by:

- The tools distribution through the workstations
- The number of times that every tool can be started on each workstation
- The degree of service usage at the request time.

The first two parameters are defined at configuration time and are stored in a system map, Services **M**ap (**SM**). When the system is running, Services Map contains all the information about the system state.

The components of Service Management System architecture are illustrated in Figure 2.

3.1. Service Management System components

We have pointed out three classes of problems, concerning communication in AxIS environment:

- To hide the communication mechanism;
- To provide both the scheduling of service requests and the starting of the processes that control each service, exploiting the distributed host environment features;
- To keep a system log and to assure some degree of fault tolerance for the system.

In order to answer the previous problems we designed SMS architecture which include the following components.

3.1.1. System Services Interface (SSI)

System Services Interface is the component that allows the interaction between the *User Interface* and Services **M**ap **A**dministrator (**SMA**: see below). System Services Interface receives the user's service requests and transmits them to the Services Map Administrator. In case of failure, SSI signals to Services **M**ap **A**dministrator **T**utor (**ST**: see below) the failure occurred.

System Services Interface is replicated on each workstation and serves many User Interfaces.

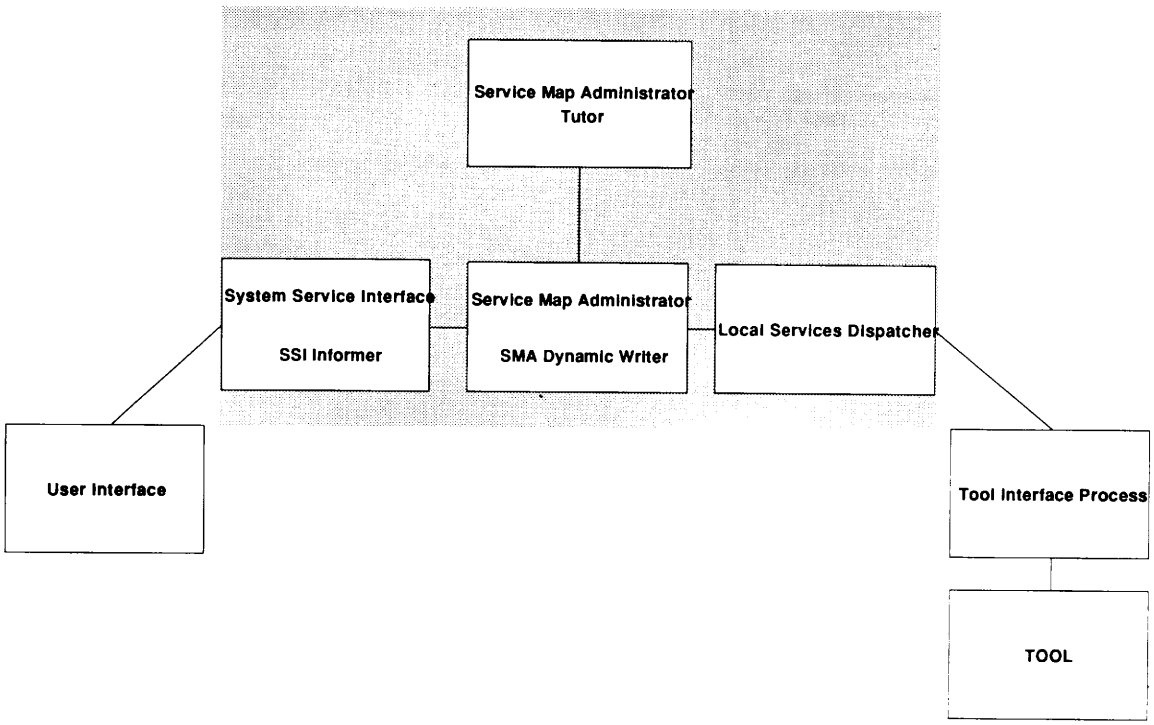


Figure 2: *SMS Architecture*

3.1.2. System Services Interface Informer (SI)

System Services Interface Informer is the process that transmits to System Services Interface the Service Map Administrator address and Service Map Administrator Tutor address. These informations are sent by Service Map Administrator Tutor.

System Service Interface Informer is replicated on every workstation.

3.1.3. Services Map Administrator (SMA)

Services Map Administrator is the master component of Service Management System. It administrates the Services Map, supplying to System Service Interface the list of available services and recording the services's usage. When a request of service arrives to Services Map Administrator, this one verifies the service's availability and determines the set of Local Service Dispatcher (**LSD**: see below) located on the workstations in which the required service resides. Then Services Map Administrator turns the request to the first Local Services Dispatcher of the set. In case of failure, Services Map Administrator tries with the others Local Service Dispatcher of the same set and if none of them responds, Services Map Administrator transmits a failure signal to the SSI that made the request.

Services Map Administrator is centralized and resides on a different machine from that on which Services Map Administrator Tutor runs.

3.1.4. Service Map Dynamic Writer (SDW)

Service Map Dynamic Writer is a process that stores the system state on a shared storage device every time Services Map is updated.

It is a centralized component that runs on the same workstation on which Services Map Administrator resides.

3.1.5. Local Services Dispatcher (LSD)

Local Services Dispatcher dispatches the services located on the machine which it controls.

It is replicated on every workstation.

3.1.6. Services Map Administrator Tutor (ST)

Services Map Administrator Tutor starts Service Map Administrator and restarts it in case of system crash. It has two main functionalities.

The first one is to run Services Map Administrator and Services Map Dynamic Writer at Service Management System start up time.

The second one is to broadcast the actual address of Service Map Administrator and its own address to all System Service Interface Informer, every time SMA is activated (at start up or after a crash).

It is a centralized component, which resides on a different machine from that on which Service Map Administrator is located.

Because of Services Map Administrator is the most critical component in our system, we have managed SMA crashes. Every time Services Map Administrator crashes, one of the System Service Interface processes or one of the **T**ool **I**nterface **P**rocess (**TIP**: see below) signals the failure to Service Map Administrator Tutor, that restarts Service Map Administrator and Service Map DynamicWriter on another machine.

If only Services Map Administrator Tutor crashes, this does not prevent the system continuing, but recovery is no longer possible.

We have not decided yet if an automatic management of this problem should be carried out, although techniques for this are known. An example is the solution adopted in the NIS system [Sun90a].

The Service Management System components are present in both the RPC upper level [Wei89a] and TCP transport protocol.

3.2. Integration Mechanisms

To support the life cycle model and design methodology for a AxIS instance it necessary to provide a set of specific tools. These tools have to be integrated so to make their services available to the environment. Data and functional integration are necessary to achieve this goal.

Data integration is necessary to convert the tool's data format into the project database format, that can be viewed and accessed from other tools.

Functional integration is required to have a transparent access to the tool's services from the AxIS environment.

3.2.1. Tool Interface Process (TIP)

Data and functional integration of a tool is subordinated to the construction of a Tool Interface Process [Dem88a]. These processes are specific for each tool to be integrated into the environment. To built a TIP for a tool means to allow communication between the infrastructure, i.e. PDBMS, UIMS, SMS. The coding of a TIP imply the specification the services the tool supply and the way to convert data from the tool internal format to the PDB format and from the PDB format to the internal format. This policy ensures uniform criteria for tools integration. Local Service Dispatcher, after a request of service, starts a Tool Interface Process, which advises Service Map Administra-

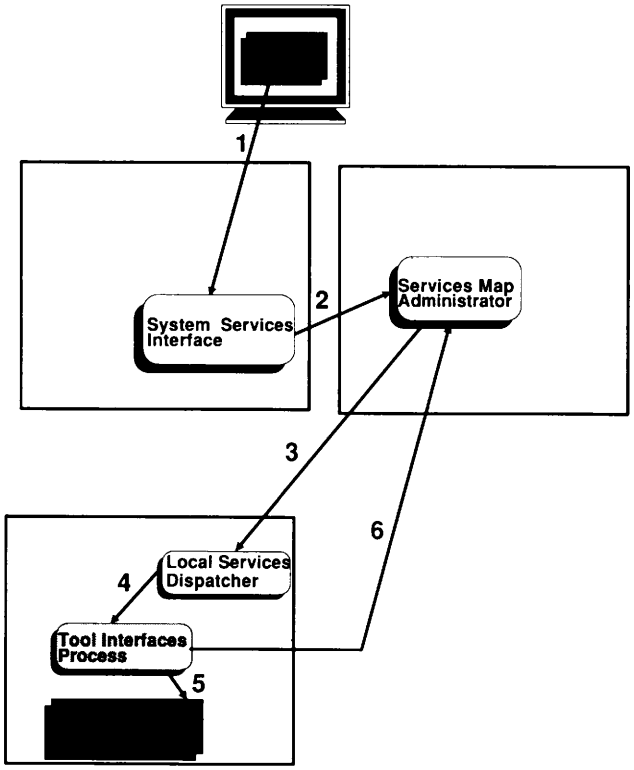


Figure 3: *Service request*

tor when the service is completed. In the case of failure, TIP communicates the occurred failure to Service Map Administrator Tutor.

Tool Interface Process must reside on the same workstation on which the tool is located.

3.3. Integration Mechanisms

User is in charge of the development of Tool Interface Process, in that the tool data format and the services supplied by the tool are known only when a specific instance is generated.

We are actually working on to the designing and implementation of a special purpose language, called **T**ool **I**nterface **L**anguage **(TIL)**, that will make Tool Interface Process coding easier. Our goal is to realize a unique language to hide the different languages for accessing the infrastructural basic systems (OODBMS SERVER, SMS).

Another functional integration facility is given by the **S**ervice **M**appe **R** **(SMR)**, whose aim is to generate Services Map.

3.4. SMS Dynamic Example

In this section we summarize the sequence of transactions activated by a service request (Figure 3) and the recovery procedure in case of SMA failure (Figure 4).

A service request is done to the local System Services Interface by a User Interface (Figure 3: arrow1). System Services Interface transmits it to the Services Map Administrator (Figure 3: 2).

Once SMA has received the request, it looks for the service availability and determines the set of Local Services Dispatcher of the machines on which the required service resides. Then, SMA turns the request to the first LSD of the set (Figure 3: 3).

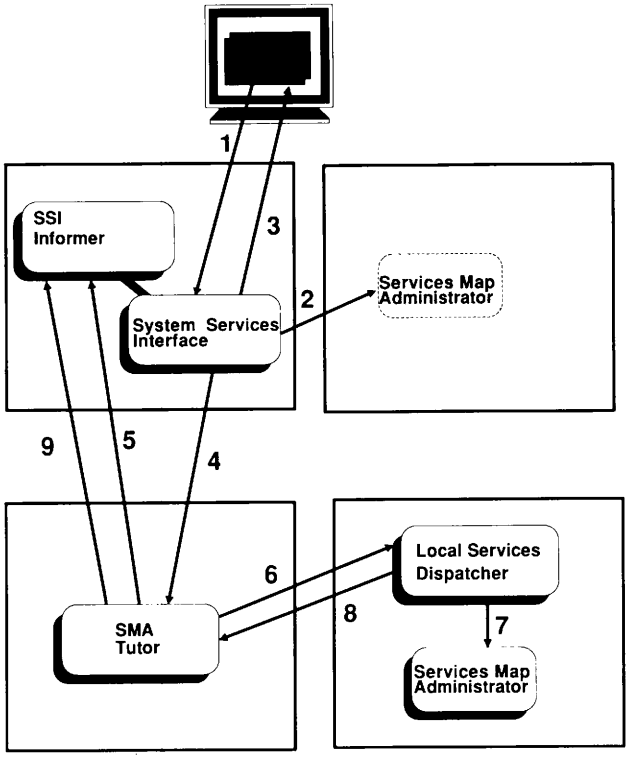


Figure 4: *SMS crash*

LSD activates the Tool Interface Process (Figure 3: 4).

TIP executes all the necessary operations related to the requested service and activates the Tool (Figure 3: 5), that serves the request. When the service is completed, TIP send a completion message to SMA (Figure 3: 6).

As said, SMS is implemented using RPC. This protocol allows to determine, on a remote request, if server process is alive. For this reason we have managed the possible failure of SMA, exploiting the processes that communicate with that component (SSI and TIP).

In the following we give an example of recovery procedure.

A service request is done to the local System Services Interface by a User Interface (Figure 4: 1).

SSI transmits it to the Services Map Administrator (Figure 4: 2).

Because of SMA doesn't reply, SSI sends User Interface a warning message, temporarily inhibiting the *services menu* (Figure 4: 3).

Then, SSI signals the SMA failure to Services Map Administrator Tutor (Figure 4: 4).

ST signals to all System Services Interface Informer the SMA failure (Figure 4: 5), so to inhibit the service request on every machine, then ST requires the activation, via Local Service Dispatcher (Figure 4: 6), of the new SMA (Figure 4: 7).

LSD signals SMA is on again to ST (Figure 4: 8). ST broadcasts the actual address of Service Map Administrator and its own address to all System Service Interface Informer (Figure 4: 9), that restores the services menu.

4. Conclusions

In this paper we presented the Service Management System architecture of the configurable SEE AxIS. The architecture goal is to allow the integration of user-defined production tools in a distributed environment and the transparent access to the tools' services.

Our future work will concern the design and the development of the special purpose language for building Tool Interface Process and the improvement of the fault tolerance of the system.

References

[Arm89a] P. Armenise, "Software Process Machines: a framework for future software development environments," pp. Springer-Verlag in *Proc. of ESEC '89 2nd European Software Engineering Conference. Lecture Notes in Computer Science*, University of Warwick, Coventry (September 1989).

[Ava89a] D. Avallone, G. La Rocca, F. Nachira, and O. Viele, "AXE in AxIS: Un tool per la definizione e la manipolazione di linguaggi grafici collocato nell'ambiente di progettazione software AxIS," in *Proc. Conf. AICA*, Trieste (October 1989).

[Cec89a] M. Morandi Cecchi, F. Nachira, and O. Viele, "AXE: The Syntax Driven Diagram Editor for Visual Languages Used in the Software Engineering Environment AxIS," pp. Springer-Verlag in *Proc. STACS '89, Conf. 6th Annual Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science*, Paderborn (February 1989).

[Cor90a] P. Corte, S. De Panfilis, and D. Presenza, "A Conceptual Modelling Enhanced Environment," in *CNR Project "Sistemi Informatici e Calcolo Parallelo, Tech. Rep. 5/21* (October 1990).

[Dem88a] S. A. Demurjiam, "A Data Base System Environment for Supporting Computer-Aided Software Engineering," in *Tech. Rep. Computer Sience and Engineering Department, University of Connecticut*, Storrs, Connecticut (1988).

[Flo86a] F. Flores and T. Winograd , *Understanding Computers and Cognition*, 1986.

[Jef88a] D. R. Jeffrey and V. R. Basili, "Validating the TAME Resource Data Model," in *Proc. 10th Intl. Conf. on Software Engineering*, Singapore (1988).

[Mal88a] T. W. Malone, "What is Coordination Theory?," in *Proc. of National Science Foundation Coordination Theory Workshop*, MIT Cambridge Massachusetts (February 1988).

[Nac89a] F. Nachira, S. De Marchi, A. Inferrera, E. Morandin, and O. Viele, "Un tool per la definizione dei linguaggi grafici," in *Proc. Conf. AICA*, Trieste (October 1989).

[Pen88a] M. H. Penedo, "Guest Editors' Introduction – Software Engineering Environment Architecture," in *IEEE Transactions on Software Engineering* (June 1988).

[Roc89a] G. La Rocca, S. De Marchi, A. Inferrera, E. Morandin, and O. Viele, "Object-Oriented Paradigm in AxIS User Interface," in *Proc. of 1st Int. Conf. TOOLS '89. Technology of Object-Oriented Languages and Systems*, Paris (November 1989).

[Sun90a] SunMicrosystems, "Network Information Service," in *Sun Manual: System & Network Administration* (March 1990).

[Weg90a] P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," in *OOPS Messenger* (August 1990).

[Wei89a] W. E. Weihl, "Remote Procedure Call," pp. 65-86 in *Distributed Systems, edited by Sape Mullender, ACM Press*, New York (1989).

Incorporating Multimedia into Distributed Open Systems

Gordon S. Blair Geoff Coulson
Nigel Davies Neil Williams

Lancaster University, UK
mpg@comp.lancs.ac.uk

Abstract

In the near future, system software must adapt to the needs of open systems and also to the desire to incorporate multimedia capabilities. This paper describes an approach to reconciling the apparently conflicting demands of heterogeneity and multimedia. Our solution is based on a configurable add-in module with multimedia peripherals together with the necessary processing power and network interfacing. Application and host workstation access to this module is via an ODP compatible base services interface. Higher level software is also described which will complement the base services to provide a comprehensive platform for open multimedia applications.

1. Introduction

Operating systems have had a long history of development, with early batch systems giving way to large time sharing systems and eventually being replaced by more lightweight kernels such as UNIX. In more recent times, the field has reacted to falling hardware costs and the evolution of networking technology by developing distributed operating systems (e.g. [Tan86a, Che84a]) running over local area networks. The current state of the art is that areas such as communications networks, protocols, network management and distributed operating systems are relatively well understood. Products such as Ethernet, X-25 and Mach [Ras86a] are available in the market place and standards to provide open communications are generally agreed.

However, the operating systems field is far from static. In particular, recent years have seen two important developments both of which are stimulating new research in the operating systems community. These developments are:

- Multimedia

  The emergence of multimedia distributed systems has been driven by both user requirements and technology developments. From the user perspective, there is a realisation that collaborative facilities such as electronic mail and remote access to computing services are not enough and that collaborative working can be considerably enhanced by the integration of a range of communi-

cations media such as voice, image and video, in addition to text. A variety of application areas such as office information systems, scientific collaboration, conferencing systems and distance learning have been identified.

From the technological perspective, recently available local area network technologies such as FDDI, the fast Cambridge Ring and wide area Integrated Service Digital Networks (ISDN), have the potential to support the demands of multimedia communications [Heh90a]. In addition, there are a number of research projects looking at the production of multimedia workstations, e.g. Multiworks.

- Open Systems

  Heterogeneity has long been an issue in systems design. The designers of many distributed operating systems have avoided the problem by developing kernels which run on homogeneous processors over local networks. However, this approach is rapidly becoming infeasible and there is an ever increasing need to provide consistent platforms over multivendor processors and wide area networks. These needs are being addressed by several standards bodies, for example OSI (through their Open Distributed Processing initiative) and the European Computer Manufacturers Association (ECMA).

  The introduction of multimedia computing exacerbates this problem in two main ways. Firstly, although standardisation in communications protocols has been making progress, multimedia has additional communications requirements (especially real time requirements) which may not be satisfied by current standards [Cou90a]. Also, the desire to distribute multimedia applications over wide area networks requires such protocols to be available in an internet environment. Secondly, the stringent performance requirements of multimedia dictate that solutions to the heterogeneity problem must not be too computationally expensive.

The above considerations have led the distributed multimedia research group at Lancaster University to address the problem of integrating media such as video and audio into operating systems structures while being aware of the importance of the open systems requirements. This paper reports on the initial results of this research. Sections 2 and 3 set out in more detail the objectives of the research and outline our general approach. This is followed in section 4 with details of our implementation work. The hardware infrastructure of our experimental setup is described in section 4.1 and the basic software infrastructure in section 4.2. The software description is broken down into sections on distributed systems support, the basic multimedia objects, structured objects, support for co-operative working and finally tools and applications. Section 5 then reports on the current implementation status of the work and section 6 offers some concluding remarks.

2. Objectives of the Research

The main aim of the research at Lancaster is to develop techniques to support interactive multimedia applications in a distributed environment. Furthermore, these techniques are being developed in the context of emerging work on Open Systems. As outlined above, the problem of integrating multimedia services into an open environment is currently not well understood and our research is addressing this

deficiency in the state of the art. In particular, the research has the following goals:

i) To develop a range of multimedia services (including the storage of multimedia objects) in a distributed environment,

ii) To provide access to the range of services through an Open Systems interface,

iii) To provide real time performance in both access to stored multimedia objects and the transmission of multimedia objects,

iv) To ensure that the solution is suitable for a heterogeneous environment consisting of a range of different workstations with different capabilities, and

v) To allow existing workstations to be integrated into the environment, i.e. specialised multimedia workstations will not be required.

3. General Approach

The work at Lancaster is ambitious in that it is tackling two of the recognised difficult problems of distributed systems, namely real time performance and heterogeneity. Researchers have been aware of these problems for a number of years but satisfactory solutions have proved to be elusive. However, with the emergence of multimedia computing and the need for Open Systems, solutions to both questions are urgently required.

Multimedia computing demands real time performance to provide support for the range of interactive applications such as distance learning and conferencing. Moreover, this level of performance is required throughout the system, i.e. in communication subsystems, storage facilities, window management and so on, to ensure proper support for the full range of interactive tasks.

Similarly, Open Systems make considerable demands of computing technology. The ultimate aim of an Open System is to interconnect a range of different computer systems each with their own operating system, hardware configuration and application software into a complete and coherent environment. The normal solution to accommodating this heterogeneity is to layer a platform on top of the host environment which provides a unified abstraction of the underlying system and hides aspects of distribution. This approach must however be at the expense of performance.

It would seem therefore that there is an apparent contradiction between the requirements of multimedia computing and those of Open Systems. On one hand, multimedia computing has stringent performance requirements whereas on the other hand Open Systems would appear to impose an extra computational overhead. This is precisely the problem to be tackled in the research at Lancaster. The work aims to develop engineering techniques to support real time multimedia computing while remaining in the context of an Open System. We believe that a solution to this problem can only be found by providing external support (in terms of hardware and software) for both multimedia computing and Open Systems. To test this hypothesis, we are developing a *multimedia enhancement unit* to provide support for a range of multimedia services and to provide a basic Open System infrastructure. The enhancement unit is independent of the host workstation and operating system and hence provides a solution to handling hetero-

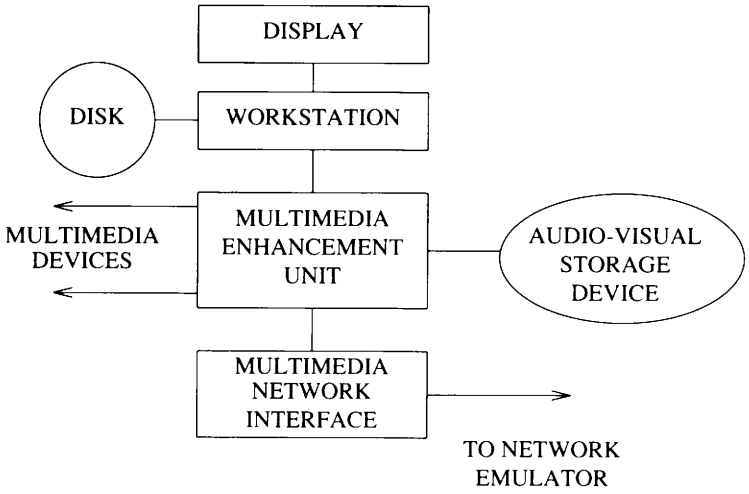


Figure 1: *Multimedia Enhancement Unit*

geneity. In addition, many of the computationally intensive tasks can be carried out on the enhancement unit and hence real-time performance becomes a more realistic goal.

The multimedia enhancement unit interfaces directly to a high speed network emulator being developed in the MNI project at Lancaster [Bal90a]. This facility provides a real time emulation of a high speed multimedia network thus allowing realistic experiments to be carried out. In addition, the network interface also directly implements a stack of protocols providing services such as remote control protocols, group execution protocols, voice and video channels (of varying characteristics) and synchronization across channels.

The overall approach is depicted in Figure 1.

4. Implementation Details

4.1. Hardware Infrastructure

A transputer based approach has been adopted in the development of the enhancement unit. There are two main reasons for taking this approach. Firstly, transputers allow processing power to be provided where necessary to ensure the necessary real time behaviour of a system. Secondly, the approach is extensible in that new devices and increased processing power can readily be introduced.

The enhancement unit consists of a small number of transputers together with a number of transputer based device interfaces. The precise configuration is shown in Figure 2.

The existing experimental environment consists of a number of IBM-compatible personal computers (currently two) running MS-DOS, a SUN workstation running UNIX and a multimedia storage node. All are connected to the network emulator via instances of the multimedia enhancement unit. Each workstation also has a number of multimedia peripherals including microphones, speakers and cameras. They also have specialised displays in order to support video windowing capabilities. The current configuration is summarised in Figure 3.

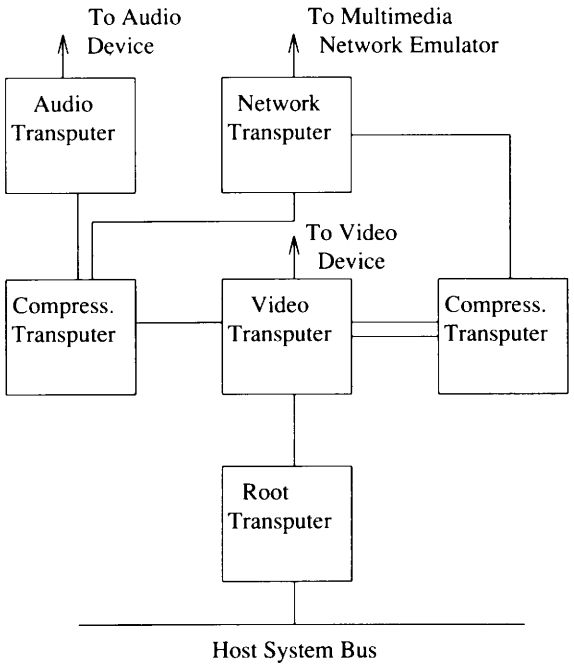


Figure 2: *Hardware Architecture*

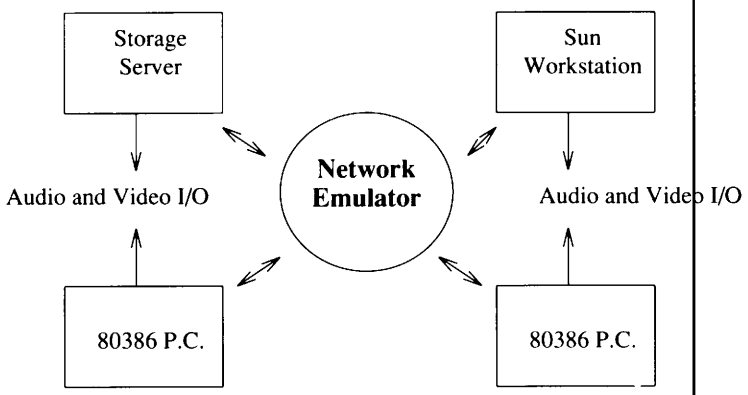


Figure 3: *Existing Hardware Configuration*

4.2. Software Infrastructure

The computational model for the multimedia environment is based directly on work from the Open Systems community and in particular from the ANSA project. All services are treated uniformly as objects, i.e. encapsulations of state and operations defined on that state. Objects are made available for access by exporting an object interface to a *trader*. The trader therefore acts as a database of services available in the network. Each entry in this "database" describes an interface in terms of an abstract data type name for the object and a set of attributes associated with the object. A process wishing to access an object must then *import* an object interface by specifying a set of requirements in terms of a type name and attribute values. This will be matched against the available services in the trader and a suitable candidate selected. Note that an exact match is not required; ANSA specifies a subtyping policy whereby an interface providing at least the required behaviour can be substituted. Finally, once an interface has been selected, the

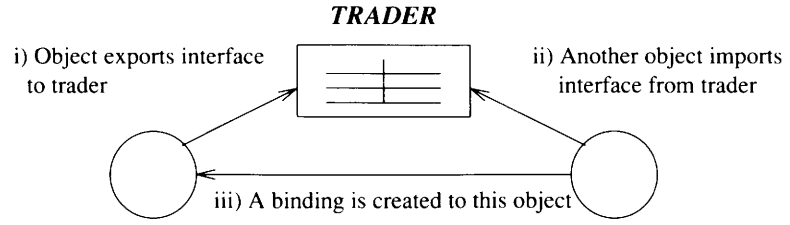


Figure 4: *Trading and Binding*

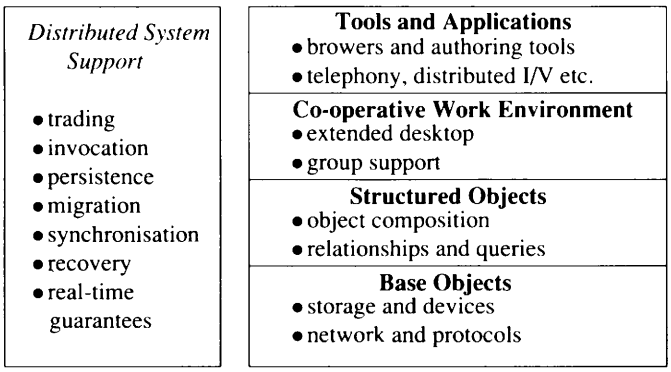


Figure 5: *Software Infrastructure*

system can arrange a *binding* to the appropriate implementation of that object and thus allow operations to be invoked. This binding is typically provided by the ANSA remote procedure call protocol, REX. This process of trading and binding is depicted in Figure 4.

The ANSA architecture provides a basic infrastructure for the multimedia development. However, this general model must be refined and, in some cases, modified to incorporate the particular characteristics of multimedia objects. Figure 5 illustrates the particular infrastructure adopted in the research at Lancaster.

This diagram shows a layering of objects within the overall environment, from base objects such as storage mechanisms through to multimedia applications. The diagram also shows the distributed systems support which provides the basic ANSA infrastructure.

In terms of implementation, the distributed system support is replicated on both the enhancement unit and the host workstation. This provides a common platform for all objects in the system whether they reside on the enhancement unit or the host workstation. Critical services are typically provided directly on the enhancement unit with less time-critical objects residing on the host environment. In practice, the approach we have taken is to implement the base objects directly on the enhancement unit thus guaranteeing high performance implementations of multimedia protocols, multimedia storage, etc..

The various elements in Figure 5 are discussed in more detail below.

4.2.1. Distributed System Support

As mentioned above, the distributed system support is based closely on the ANSA architecture. More specifically, the implementation relies heavily on the facilities offered by the ANSA testbench [ANS88a]. The ANSA testbench is a suite of software which provides a partial implementation of the ANSA architecture. In particular, the testbench imple-

ments the concepts of trading, binding and remote procedure calls as discussed above. It also provides facilities such as lightweight threads and basic synchronisation primitives (eventcounts and sequencers).

The ANSA testbench runs on a number of environments including UNIX, VMS and MS-DOS, and thus provides the necessary platform to construct distributed applications in a heterogeneous environment. For the purposes of our experimentation, the testbench was also ported to the transputer environment in order to have a common platform across the entire configuration described in section 4.2. It is therefore possible to access objects in the distributed environment whether they reside on the host workstation, on the multimedia enhancement unit or on any of the other nodes in the distributed environment.

The testbench provides the necessary infrastructure for the multimedia developments. However, it was necessary to make a number of modifications and extensions to the basic testbench to meet the particular requirements of multimedia. The additional features are described below.

- Generalised Invocation

  The first, and perhaps most important, alteration is the incorporation of a generalised invocation service within the ANSA testbench. In distributed object-oriented environments, invocation is traditionally seen as mapping directly on to a remote procedure call protocol. However, in our architecture, invocation is considered to be a conceptual idea which does not imply any particular implementation strategy. Invocation may be implemented by remote procedure calls; however, a number of other mechanisms can also be used (e.g. distributed virtual memory [Tav87a] or proxies [Sha89a]). This flexibility is important in distributed multimedia systems where appropriate mechanisms must be selected to provide the required level of performance (or more generally, quality of service).

  A complete description of the invocation approach is beyond the scope of this paper. Briefly, however, invocation involves the following steps. Firstly, it is necessary to locate the object in the distributed system. A decision must then be made as to where the object should reside and how it should be accessed. Following this, the object may have to be migrated to the chosen location. It is also necessary to ensure that the object is currently active (i.e. ready to respond to invocations). If not, the system must activate the object by creating a process and retrieving the state of the object from persistent storage.

  In summary, invocation deals with a number of issues in a distributed system including the location and access of objects, migration of objects and persistence. Further details of the approach to invocation can be found in [Bla90a].

- Flexible Trading

  Another area of ANSA which is affected by the nature of multimedia systems is trading. The nature of multimedia services means that many similar services with slightly differing characteristics will be present. There may be services that provide *at least* the functionality required by a client (plus some additional functionality not required for the current interaction). In addition, there may exist services with slightly different specifications to those required which otherwise are not observably different from the service required. For example, slow scan video (10 frames/sec say) and full speed video (25 frames/sec)

can still have the same operations invoked on them (play, stop etc.); it is just that the frames/sec parameter differs slightly.

In the first case where services with at least the functionality required exist, the use of a subtyping policy [Bla90b] at the abstract data type level can be used to cope with this proliferation of services (at present, ANSA has a rather crude approach to subtyping [Mac89a]). In the latter case where specifications differ slightly, the use of facilities such as compression algorithms can be used to transparently fit potential services to the particular requirements of the client. These mechanisms are the basis of what we call flexible trading. Further details of the approach to flexible trading can be found in [Mac90a].

- Flexible Transactions

  Transaction mechanisms have traditionally been concerned with the maintenance of the consistency of a group of objects in spite of concurrent access and the possibility of partial failure of parts of the system. The emphasis has been on providing preventive mechanisms to ensure that inconsistencies do not occur. A wide portfolio of techniques (two phased locking, timestamps, logs, etc.) have been developed to assist in this task. More recently however there has been greater interest in less rigid techniques which are more geared towards the semantics of the application domain and hence tailored towards the cooperative nature of the task. This is having a major impact on the design of transaction mechanisms with techniques such as non-serialisable transactions starting to appear. The emergence of multimedia systems will promote this activity and inevitably other alternative designs for transaction services will appear.

  The biggest impact of multimedia is that transaction mechanisms will have to be more flexible. More specifically, it will be necessary to tailor particular transaction mechanisms for the requirements of given classes of application. This introduces the notion of application dependent transactions, where the actions of the transaction management are dependent on the precise context of execution. The issue of flexibility in transactions is currently being examined at Lancaster in the context of the multimedia developments [Bla90c].

- Communications Abstraction

  At present, most distributed object-oriented systems hide communications behind the notion of invocation, i.e. objects invoke other objects with the underlying passing of messages being hidden. However, with the introduction of multimedia objects, it is not possible to hide communications completely. Invocation alone is not sufficient to express the full generalities of a multimedia system. Invocation is sufficient to express the calling of operations on objects but cannot naturally describe operations which result in a continuous flow of information, e.g. a *play* operation on a camera. In general, this situation occurs with all the continuous media types (video, animation, audio, etc.). It was therefore necessary to add extra features to the testbench implementation to handle continuous media types. Further details of the extensions can be found in section 4.2.2 below and in [Cou90a].

Our implementation of ANSA also includes facilities to create objects in the distributed environment. Object creation is carried out by factories which generate new instances of objects according to a particular set of

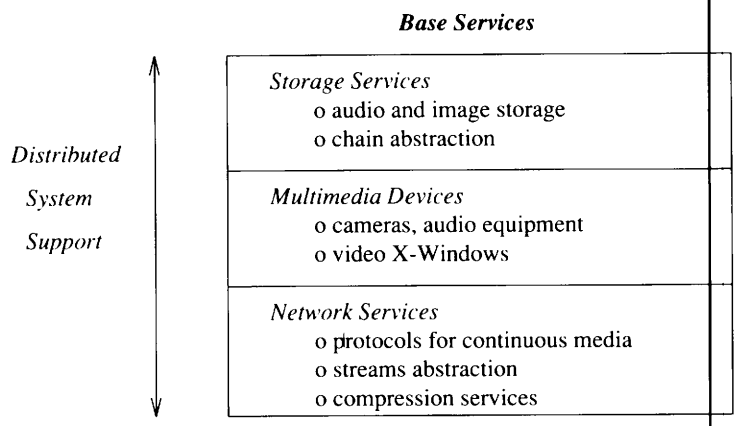


Figure 6: *Software Architecture*

requirements (the requirements are given as parameters to the factory). In a given configuration, a number of factories will exist, corresponding to the various object types described in the rest of section 4.2.

The factory concept is actually part of the ANSA architecture but is not implemented in current releases of the testbench. It has therefore been necessary to extend the testbench with the factory concept and to develop a number of specialised factories for multimedia objects.

4.2.2. Base Objects

The base objects can be subdivided into three general categories as follows: storage services, multimedia devices and network services as shown in Figure 6.

The various components highlighted in Figure 6 are described in more detail below.

- Network Services

  The main task of the network services layer is to provide a range of protocols (of varying qualities of service) to handle multimedia communications. This required the provision of *stream* services through the ANSA architecture (this is in addition to the standard RPC protocols already supported by ANSA, as discussed in section 4.2.1). Streams represent high speed, one way connections between an information source and sink. They are therefore abstractions over transport protocols and map on to various protocols provided by a protocol stack. This protocol stack is implemented on one of the transputers in the multimedia network interface. Further details of the protocol stack and multimedia network interface can be found in a companion paper [Sco91a]. The network services layer also provides a number of compression services which can be used to reduce the bandwidth requirements of multimedia data.

- Multimedia Devices

  Each device provides an interface consisting of two parts: a device dependent part and a device independent part. The device dependent part offers a number of operations specific to that device. For example, the camera device has operations to focus, tilt, etc.. The device independent part has a number of operations to create communication *endpoints* (which are also invokable objects) and to send and receive information. At present, a small number of devices have been incorporated into the system,

including a camera service and an interactive videodisk player (as a source of still and moving image). A windowing system supporting video window capabilities is also provided at this level. To achieve this, the X-server component of X-Windows was ported on to the transputer environment and then modified to support video windows.

- Storage Services

  The approach taken at Lancaster is to provide a number of specialised storage services for each media type. This allows each storage service to be tailored to the characteristics of a particular media type, which may involve specialised hardware support or a particular approach to storing and retrieving information. Work on storage services is at an early stage although some work has been done on the conceptual model provided by the storage services. It is proposed that each storage service manages a number of separate units of information (c.f. files). Each of the units of information will be an object in its own right and will provide the abstraction of a *chain*. Chains are a generalisation of the "voice ropes" adopted in the Etherphone project for storing audio information [Ter88a]. A chain consists of a number of individual links, e.g. single picture frames, connected together into a sequence. It is perfectly valid to have a degenerate chain consisting of one link.

4.2.3. Structured Objects

One of the main features of multimedia computing is that objects tend to be complex, structured entities consisting of a number of component parts. For example, multimedia documents typically contain a number of text paragraphs, some bitmap diagrams, some vector drawings and possibly sound or animation. Furthermore, the various component parts are linked together in an intricate manner. Similarly, applications such as Hypermedia [Mey86a] require rich structuring of objects. The provision of structured objects is therefore an important part of the research. This work has taken input from such standards activities as ODA and SDML [Bro89a] but has developed structuring techniques more suited to multimedia objects.

Most of the work on structured objects has been carried out under the auspices of the Zenith project (in collaboration with the University of Kent). The aim of this project is to develop a generalised object management system to support design environments (especially those involving multimedia information). The work on this project has resulted in the specification of an object model which implicitly assumes that objects are structured. This model has been adopted for the general multimedia work at Lancaster.

An individual Zenith object consists of a number of visible components which can be of a variety of media types. Components can then be linked together by a number of relationships to form a generalised lattice structure. Two fundamentally different types of relationship exist within the system, namely logical relationships and conceptual relationships. Logical relationships define the structure of objects in terms of their composition. They therefore map closely on to "component of" relationships as found in many data models. In contrast, the conceptual relationships are user-defined higher level relationships which are used to enrich the semantics of a particular structured object. For example, a multimedia document could have a conceptual relationship defining the authors of the document. Note that it is often possible to use either log-

ical or conceptual relationships to define structure. The difference between the two is that conceptual relationships can have user-defined semantics.

Constraints are placed on the structure of an object. This is seen as important to preserve the consistency of a structured object. For logical relationships, this is implemented by a regular grammar which limits the components that can exist in a particular structured object. Conceptual relationships however require a more complex constraining mechanism to capture the semantic constraints on an object. For conceptual relationships, constraints are realised by a predicate calculus which provides a set of rules which must be maintained by that object.

Creation of structured objects is managed by one or more structured object factories. These factories take as parameter the grammar and the set of predicates and generate a structured object which maintains these constraints. In this way, it is possible to define specific object structures tailored towards particular applications.

4.2.4. Co-operative Working Environment

It is expected that the integration of multimedia into a distributed environment will have a radical impact on the future use of information technology. The addition of audio and video communication to distributed systems will provide the support necessary to allow remotely situated users to work together within an environment which has the required audio/visual functionality for successful computer supported group working (or CSCW). Due to the close relationship between multimedia and cooperative working, it is important that a distributed system supporting multimedia should also provide system support for group working. It is not until such combined infrastructures are available that distributed systems can achieve their full potential as totally integrated information systems.

The Lancaster architecture is aiming to provide an environment which understands the nature and requirements of multimedia and group working and, therefore, support these aspects at the system level. Building on top of the group execution protocols provided by the communication system, the higher levels provide *group* objects which support the construction of group working environments. Group objects maintain information on the current status of a group including information on group members, applications being used by the group, etc.. The mechanisms for creating and using these group environments are provided by an extended desktop user interface which in a similar way to a normal desktop allows objects to be created, invoked, and destroyed, but also allows these operations to be performed within the context of user groups. For example, the execution of a multimedia editor within a group environment would result in all available group members receiving an interface to the editor.

4.2.5. Tools and Applications

One of the important considerations of the approach taken at Lancaster is that the development of the distributed multimedia support system must take on board end user requirements. The assessment of these requirements, however, is difficult when end users have no experience of the technology and its potential. To help in this task several tools and applications are being developed which range from browsing and authoring utilities to video conferencing. Through experience of these applications, it is intended to both gain experience of what can be

achieved with the available technology and provide realistic testing of the underlying communications and storage systems. The multimedia applications will be used extensively by the research staff and also by a small group selected from one of our industrial partners.

5. Implementation Status

The research described in this paper is now entering its second year, and significant progress has been made in implementing the ideas set out above. In hardware terms, both the multimedia network interface and the network emulator are complete and the experimental set-up illustrated in Figure 3 is in place. The ANSA testbench has also been ported onto the transputers of the network interface.

The base services are largely in place. The X-Windows service, enabling video windows to co-exist with ordinary text and graphics windows, is complete and the cameras and interactive video disc are accessible as ANSA services. The stream communications abstraction is in place, and provides access to a basic set of communications protocols. The current emphasis of the low level work is on the implementation of a fully comprehensive multimedia protocol stack. Work will also start shortly on the implementation of prototype storage services including a high quality audio store.

A number of sample applications have been developed to exercise and validate the underlying system. For example, disc jockey and production room consoles have been written which allow the compilation, maintenance and playback of stored music and video tracks in a distributed environment. In addition, we have a network telephone application embedded in the co-operative working environment which allows both point to point and conference calls. These applications make full use of the user interface desktop, communications abstractions and ANSA based multimedia services.

Other work in the higher layers of the architecture is progressing. For example, a generic factory service is under development, from which arbitrary structured objects can be generated. Also, extensions to the trading and object location functions are underway. These include an implementation of the flexible trading ideas and, in connection with the Zenith project, services to provide generalised queries in a large database of structured multimedia objects.

6. Conclusion

This paper has described an approach to the problems of heterogeneity and real-time performance in distributed, multimedia systems. This approach is based, at a low level, on a transputer based multimedia enhancement unit which manages multimedia services in a distributed environment. To cater for heterogeneity, this multimedia network provides a standard ADT interface based on ANSA. Higher level distributed systems support is tailored for maximum flexibility, which is an essential requirement in our target environment.

7. References

[ANS88a] ANSA, "The ANSA Reference Manual Ed. 00.05 (Parts 1, 2, 3)," *APM Ltd, Poseidon House, Castle Park, Cambridge, UK* (1988).

[Bal90a] F. Ball, D. Hutchison, A. C. Scott, and W. D. Shepherd, "A Multimedia Network Interface," *3rd IEEE COMSOC International Multimedia Workshop (Multimedia'90)*, Bordeaux, France (November 1990).

[Bla90b] A. Black and N. Hutchinson, "Types and Polymorphism in the EMERALD Programming Language," *Dept. of Computer Science, University of Washington, USA* (June 1990).

[Bla90a] G. S. Blair, "Distributed Invocation in Multimedia, Object-Oriented Systems (Position Paper)," *Workshop on Object Orientation in Operating Systems*, Ottawa, Canada (October 1990).

[Bla90c] G. S. Blair and B. F. de Toledo, "Flexible Transactions in Distributed, Multimedia Systems (Extended Abstract)," *Computing Department, Lancaster University* (September 1990).

[Bro89a] H. Brown, "Standards for Structured Documents," *The Computer Journal* **32**, pp. 505-514 (1989).

[Che84a] D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software* **1**, pp. 19-42 (1984).

[Cou90a] G. Coulson, G. S. Blair, N. Davies, and A. Macartney, "Extensions to ANSA for Multimedia Computing," *Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK.* (September 1990).

[Heh90a] D. B. Hehmann, M. G. Salmony, and H. J. Stuttgen, "Transport services for multimedia applications on broadband networks," *Computer Communications* **13**, pp. 197-203 (1990).

[Mac89a] A. Macartney and G. S. Blair, "Conformance in Object-Oriented Systems: Problems and Solutions," *Proceedings of the First Maghrebin Conference on Artificial Intelligence and Software Engineering*, Constantine, Algeria (1989).

[Mac90a] A. Macartney and G.S. Blair, "Flexible Trading in Distributed Multimedia Systems," *Computing Department, Lancaster University* (Sept 1990).

[Mey86a] N. Meyrowitz, "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, Portland, Oregon **21**, pp. 186-201, Special Issue of ACM SIGPLAN Notices (1986).

[Ras86a] R. F. Rashid, "Threads of a New System," *UNIX Review* **4**, pp. 37-49 (1986).

[Sco91a] A. C. Scott, F. Ball, D. Hutchison, and P. Lougher, "Communications Support for Multimedia Workstations," *to appear in Proceedings of 3rd IEE Conference on Telecommunications (ICT'91)*, Edinburgh, Scotland (1991).

[Sha89a] M. Shapiro, P. Gautron, and L. Mosseri, "Persistence and Migration for C++ Objects," *Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP '89)*, Nottingham, England (1989).

[Tan86a] A. S. Tanenbaum and S. J. Mullender, "The Design of a Capability-Based Distributed Operating System," *The Computer Journal* **29** (1986).

[Tav87a] A. Tavanian, "Architecture Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach," *Dept. of Computer Science, Carnegie-Mellon University* (1987).

[Ter88a] D. B. Terry and D. C. Swinehart, "Managing Stored Voice in the Etherphone System," *ACM Transactions on Computer Systems* **6** (1988).

Scalable Mainframe Power at Workstation Cost

J-P. Baud J. Bunn F. Cane F. Hemmer
E. Jagel G. Lee L. Robertson
B. Segal A. Trannoy I. Zacharov

CERN, Geneva, Switzerland.
ben@cernvax.cern.ch

Abstract

Until very recently, big mainframes were considered essential for the provision of large-scale scientific batch computing services, requiring intensive tape and file space management, high throughput and maximum reliability.

However, RISC-based workstations have outstripped mainframes in CPU price/performance by an order of magnitude for some years, and workstation-class disk systems have finally become cost-competitive with mainframe disk storage. The last missing elements preventing mainframe replacement by cheaper distributed systems have been the software to enhance their (UNIX-based) operating systems, and a powerful enough networking technology.

Over the last year at CERN, two services have been established to prove the feasibility of physics batch processing using off-the-shelf high-end workstations. The initial service, called "HOPE", uses a single 4-CPU HP-Apollo DN10000 workstation, and achieves well over 80% utilization for periods of many months. The second service addresses the scalability issue, using a modest number of heterogeneous workstations on a high-speed network, connected and managed by a portable distributed set of user-level software. The CPU, disk, and tape services are functionally separated, and individually subject to price-performance optimization. This system is already comparable in CPU capacity to the full CERN computer center (including a Cray X-MP/48 and an IBM 3090-600E), and is scalable to several times that size.

The system is called the "Scalable Heterogeneous Integrated FaciliTy" or "SHIFT".

1. Introduction

Until quite recently, say five years ago, UNIX was considered to be limited in scope to "small" computer systems, whereas "big mainframes" running serious amounts of scientific/engineering problems would run only proprietary operating systems. There were good reasons for this state of affairs, primarily the lack of important system features in the

UNIXes of that period. To cite the most crucial: lack of acceptable FORTRAN; lack of a batch scheduler subsystem; lack of magnetic tape handling and scheduling features for a multiple job stream; lack of management and backup features for very large disk file systems; lack of serious user accounting features.

The first computer manufacturers to remedy some or all of these shortcomings were Amdahl (UTS) and Cray (UNICOS). The presence of a Cray X-MP/48 system running UNICOS at CERN since early 1988, and the joint development work that CERN carried out with Cray on the enhancement of UNICOS around that time, proved to us that the addition of "big mainframe features" to UNIX was not only possible but actually straightforward.

Over the same five-year period we have witnessed an extraordinary surge in the CPU power and price-performance of workstation-class systems, and over the last year or two a less spectacular but just as significant advance in the price-performance of workstation-class disk systems. Thus it has recently become possible to think of providing "mainframe computer power" at workstation cost; the major question to be answered was "how?".

Certain groups in the High Energy Physics (HEP) community have pursued the idea of "computer farms", taking advantage of one basic characteristic of their computing load: it is presented in units of work called "events" that are largely independent of each other and can be processed by a set of uncoupled or loosely-coupled parallel processors. In the early days, such processors were often special-purpose, but recently UNIX workstations with RISC CPU's have been used in farms. Some farms also explored the use of heterogeneous elements, including mixtures of UNIX and VMS systems.

However, computer farms were usually just simple task-dispatching systems, lacking the full robustness, flexibility and operability of a mature batch system. As relatively special-purpose systems they could generate impressive performance over limited periods but required considerable attention and support from their users in order to be properly exploited. In particular, they usually needed to be "fed" their data in a non-automatic fashion from a mainframe or other separate data storage facility.

2. What is "Mainframe Power"?

2.1. What is "CERN"?

CERN is the European Laboratory for Particle Physics, a collaboration between 15 European member states to carry out fundamental research in the physics of very high energy particle interactions. This research uses very large accelerators, in conjunction with extremely complex detectors and chains of data reduction hardware and software. CERN's facilities are not restricted to member-state physicists but are also open to the world-wide physics community. CERN's users normally operate in "collaborations", each of which will carry out a given set of experiments and typically consist of 50-200 physicists from 5-20 universities in 2-10 countries. Around the latest (and largest) CERN accelerator, known as LEP, there are four such large collaborations with the names ALEPH, OPAL, L3 and DELPHI; much of the work described in this paper was carried out in conjunction with OPAL.

2.2. CERN's Mainframes

Currently CERN's central computing service consists of four major elements (see Table 1).

Mainframe	CPU(CERN Units)	Disk(GBytes)
Cray X/MP-48	32	50
IBM 3090-600E	39	250
Siemens 7890	13	5
Vax 9000-410	9	50
Total	93	355

Table 1: *CERN – Central Mainframes*

Note that a "CERN CPU Unit" is equivalent to an IBM 370/168, a Vax 8600, or about 4 Vax 11/780's. Note also that only **scalar** CPU power is compared in this paper, as HEP problems are not very susceptible to vectorization.

2.3. LEP's Computing Requirements

Each LEP collaboration records its physics measurements on magnetic tape, and requires that this data be processed (sometimes repeatedly) in various "offline" modes to extract the physics results that are published. Each collaboration also generates large quantities of *simulated* physics data, plus *reconstructed* data from partial analysis steps. The total quantity of such recorded data for 1991 will be 20 Terabytes, mostly held on about 100,000 IBM 3480 cartridges. Occasional access to the whole of this vast data store, together with frequent access to a few hundred Gigabytes of it, must be provided by any computer system proposed for its reduction or analysis.

The requirements estimated for LEP data reduction (in 1991) amount to 100 CERN Units and 200 Gigabytes of online disk storage, whereas only 40 CERN Units and 50 Gigabytes of central mainframe capacity have been made available to LEP experiments. The LEP collaborations have therefore equipped themselves with additional private computing facilities to the extent they could. However, the quantity of data that will arrive from LEP in future years will completely swamp the computing resources now available, and exceed any extensions that are affordable at current mainframe prices.

3. The "HOPE" Project

In mid-1989, CERN's Computing & Networking Division and the OPAL physics collaboration made a proposal to HP/Apollo for a joint project to develop a reliable batch service for physics production computing on an HP/Apollo DN10000 system with four tightly-coupled **Prism** RISC processors and 4 Gigabytes of disk space. CERN benchmark codes show that each such CPU is equivalent to between 4 and 5 CERN Units; in what follows we rate a 4-CPU system conservatively at 16 Units. The resulting project was named the "H-P Opal Physics Environment" (or "HOPE").

3.1. Goals

The "HOPE" project's goals were as follows:

- Achieve "mainframe-class" service quality overall
- Port and run realistic physics production codes
- Port and develop the public domain batch system NQS
- Provide IBM 3480 and Exabyte magnetic tape support
- Establish high-speed access to a central data robot
- Integrate HOPE operation and accounting into CERN central services
- Later expand service to manage multiple clones

3.2. Timetable

The DN10000 machine for the "HOPE" project was delivered to CERN in January 1990. The first steps taken were to set up the environment for CERN HEP physics computing, including installation of the CERN Program Library and other local utilities. Physics users from the large LEP collaborations OPAL and L3 were then allowed to get familiar with the system and port their applications. The public domain version of NQS, a batch system for the UNIX environment, was ported to the DN10000 in early May, 1990. Accounting procedures were established by mid-May at which time the batch facility was officially opened to the public. In June, an FDDI connection was installed, enabling faster access to data on Cray disks via NFS and ftp. Over the summer months, the accounting scripts were improved and work began on providing local tape support software.

In September, the HOPE1 service was integrated into the main Computer Center operation. 24-hour operator surveillance together with a file backup program were initiated. System and user accounting were also merged with those of the Cray, IBM and DEC central computers.

In early October, an 8mm Exabyte unit was installed on the HOPE1 workstation. Production analysis of data stored on 8mm tapes began shortly thereafter. In mid-November, dual IBM 3480 compatible tape drives (STK 4280) were installed on HOPE1. Work is now in progress to provide a complete set of software library routines for support of locally attached tape units. Also in November, NQS job submission scripts for VMS and VM/CMS platforms were written, and a distributed version of NQS which provides for intelligent load-sharing across a network of NQS machines was developed.

3.3. Results

Over the last seven months of 1990, the HOPE1 workstation delivered an average of 69% of its available CPU cycles. Over the last three and a half months of 1990 this availability figure was 84%, with only 12% of available CPU time going idle; the mean time between failure was 8 days, the mean time between interruptions 4.5 days and the mean time to repair 30 minutes. These figures fully attain the project's goals, in fact exceeding typical experience with the central mainframe batch systems at CERN. The total CPU delivered from September 1990 to mid-January 1991 was 39700 CERN Unit-hours, equal to 29% of the CERN IBM 3090-600E and Cray X-MP/48 combined.

The batch work on HOPE1 has been primarily submitted by physics users running simulation, reconstruction and analysis jobs. HOPE1 has

also served the CERN Program Library with approximately 400 CERN Unit hours for interactive code development and debugging. The "HOPE" project has attracted a wide interest at the CERN Computer Center, within the broad community of CERN physicists, and at other HEP institutes around the world.

4. The "SHIFT" Project

While the early results of HOPE1 were extremely encouraging, amply confirming the feasibility of running a high-quality batch service on a workstation, the service still provided less than 20 CERN CPU Units, and well under 10 GigaBytes of disk space. Simply cloning HOPE1 would not provide an integrated solution offering hundreds rather than tens of CERN CPU Units and disk Gigabytes. A much more ambitious set of goals was drawn up for a new project, named the "**S**calable **H**eterogeneous **I**ntegrated **F**acili**T**y", or "**SHIFT**".

4.1. Goals

- Provide an *integrated* system of CPU, disk and tape servers capable of supporting a large-scale physics batch service
- The system must be constructed from *heterogeneous* but *open* system components, to retain flexibility towards new technology and products
- The system must be *scalable*, both to small sizes for individual collaborations/small institutes, or upwards to at least twice the current size of the CERN computer center
- The architecture should also be capable of supporting interactive physics applications
- The batch service quality should be at least as good as mainframe batch quality
- The batch system must operate in a distributed environment, using a single set of queues for each class of CPU servers, and a unified priority scheduling scheme
- Automated disk file space control to be provided, including a tape staging service
- Support for 3480-compatible cartridge tapes as well as Exabyte tapes must be provided, including access to the CERN automatic cartridge-mounting robot
- SHIFT operation and accounting to be integrated into the CERN central computer services

4.2. Architecture

The SHIFT system consists of sets of *CPU servers, disk servers and tape staging servers*, with *distributed software* which is responsible for managing the disk space, moving data between tape and disk, locating staged files, batch scheduling and accounting. The main motivations in choosing this design, which stresses separation of function, are simplicity, flexibility, and the ability to optimize the various components for their specific functions.

The servers are interconnected by two networks: the *backplane*, a very fast network used for optimized special purpose data transfer, and the *secondary network*, used for control, operations and general purpose

file access. The backplane is connected to the site's general purpose network infrastructure by means of an *IP router*, thus providing file and batch services to workstations distributed throughout CERN with the same functionality (though with lower performance) as the services available to systems connected directly to the backplane.

4.3. Strategy and Feasibility

The strategy adopted with the above architecture involves *staging* the very large offline physics data sets upon demand between their magnetic tape "homes" and the fast SHIFT online disks, permitting them to be accessed via the *backplane* by any of the SHIFT CPU's. The staging process is performed automatically, efficiently, and in as transparent a manner as possible for a physics user.

However, the multiple transits of these large data sets across the SHIFT backplane produce a potential system bottleneck. Calculations, supported by a detailed simulation of the SHIFT design, showed that backplanes built using conventional LAN's would not be adequate: for a moderate size system of 100 CERN CPU Units, consuming 20 KBytes/s of physics data per CPU Unit per second, we computed peak backplane rates approaching 18 MBytes/s. Additionally, to keep the number of SHIFT components for this system to a manageable number (e.g. 3 tape servers, 3 CPU servers and 6 disk servers) we required peak per-server backplane rates of around 3 MBytes/s. Even if these estimates are for I/O intensive jobs, they are very revealing: 3 MBytes/s. interface rates are far beyond Ethernet and currently just attainable on FDDI at the price of a full RISC CPU for protocol processing. This led us to another unexpected conclusion: with conventional LAN's, *the associated amount of protocol processing* was shown to absorb *an unacceptable fraction* of SHIFT's CPU resources.

4.4. The Backplane

To permit SHIFT to handle a balanced load, including I/O intensive work, the solution found for the backplane is a proprietary hub-based network called "UltraNet". This provides a switched total sustained bandwidth of over 100 MBytes/s, per-interface sustained rates of between 3 and 12 MBytes/s, and built-in protocol processing with particular efficiency for stream-type applications. The associated CPU load of an Ultra-connected node is under 10% of that used by current FDDI implementations, while the application interface is completely standard (TCP/IP stream sockets). UltraNet hardware interfaces exist for most of the currently desirable SHIFT components. Finally, an IP gateway exists to connect an UltraNet backbone to external conventional LAN's.

4.4.1. Fast Remote File Access

SHIFT supports two modes of file access between its CPU and disk servers:

- Transparent access using NFS; this also applies to files external to SHIFT.
- Very high performance access using *Remote File Access* routines operating across the backplane (preferentially), or other available socket-based transport services. A *Remote File Access Library*, must be used explicitly by applications wishing to benefit from this high performance, low overhead facility. Both C and FOR-

TRAN library calls are supported. A *Remote File Access server* process runs on every SHIFT disk server.

4.5. File Base and File Pools

The SHIFT file space is organized as a series of logical *pools*. A SHIFT disk pool is a set of UNIX file systems on one or more disk servers. Each of these file systems is mounted by NFS on every CPU server, using a convention for the names of the mount points which ensures that files may be referred to by the same UNIX pathname on all CPU, disk and tape server nodes in a SHIFT configuration.

As mentioned above, the SHIFT file space is intended principally for *physics data* files. The pools of data files are *managed* by special SHIFT software utilities, under user and system control. A distinction is made between such data files and *user files* (programs, scripts, source code, etc.).

4.5.1. User Files

User files normally reside on remote file systems, but if they are of general interest (e.g. library files) they could reside on SHIFT servers. (CPU servers automatically mount both the SHIFT and any remote file systems constituting the user file base). Apart from optional file quotas, implemented using standard UNIX facilities, no special SHIFT management is done for user files.

4.5.2. Physics Data Files

SHIFT utilities manage the SHIFT data file pools, allocating files within pools, locating previously staged files on behalf of users, moving data between magnetic tape and disk, and performing garbage collection within each pool. Each staged disk file consists of a copy of all or part of an existing tape, or of data which will eventually be written to a pre-defined file on a specific tape.

Pools may be *public* or *private* for accounting purposes; *permanent* and *temporary* pools are also distinguished. Temporary files are eventually deleted by a garbage collector using a "least-recently-used" algorithm to maintain adequate free space within the system, but a user may specify that the file should be deleted at the end of a job if it is known that it will not be re-used. Files in permanent pools are not subject to automatic garbage collection.

4.6. SHIFT System Components

The SHIFT system software has been implemented as a set of separate components in the UNIX spirit. These are the *Disk Pool Manager*, the *Tape Copy Scheduler* and the *Batch Scheduler*, which in turn rely on the *Tape Allocator* and *Remote Tape Copy* utilities, together with the Remote File Access routines. The user interface is via a small set of *SHIFT Commands*.

4.6.1. Disk Pool Manager

The Disk Pool Manager (DPM) allocates files to pools, creates files and associated directories, locates existing files, deletes files on request and performs garbage collection on temporary pools. The DPM user interface is through the *sfget, sfrm, sfsh* commands, which are described below.

An important component of the Disk Pool Manager is the *Table Manager*, which maintains static and dynamic configuration information. It is responsible for locating SHIFT files, making disk pool selection decisions and serializing access to the data when necessary. In the initial implementation the Table Manager is centralized, but the design permits functions to be distributed in later versions, if necessary for performance or reliability reasons. In general the data used by the Table Manager can be recreated in the event of a serious failure of the node on which it executes.

4.6.2. Tape Copy Scheduler

The Tape Copy Scheduler organizes the copying of data between disk files and magnetic tapes. On request from a user through a *tpread* or *tpwrite* command, it selects an appropriate tape server depending on the device type required, location of the tape and current tape activity. It then initiates a tape copy using the SHIFT *cptpdsk* or *cpdsktp* program on the tape server node. The Tape Copy Scheduler informs the user when the operation is complete, queues concurrent requests for the same tape, and deals with error recovery. The *tpread* and *tpwrite* commands are described below.

4.6.3. Remote Tape Copy

The *cptpdsk* and *cpdsktp* commands copy data from tape to disk or disk to tape, respectively. They execute on the computer to which the tape unit is attached, but can access a local or remote disk file, normally using the Remote File Access system. These commands are described below.

4.6.4. Tape Allocation and Control

The remote tape copy programs use the facilities available on the local machine to allocate a tape unit, mount the tape, and handle labels. These facilities are generally provided in the form of a *tape daemon* in conjunction with a *tape driver*, and a user interface. For example under UNICOS (which is used as a SHIFT tape server at CERN) the user interface is called *tpmnt*. SHIFT user commands permit the passing of certain arguments directly to the tpmnt interface.

4.6.5. Batch Scheduler and Accounting

The batch scheduler used by SHIFT is the *Network Queueing System (NQS)*, originally developed for NASA and which is now in the public domain. Although NQS is an important facility used by SHIFT, it is entirely independent of it. CERN has extended NQS to cater for centralized batch queues which feed clusters of CPU servers.

A number of utilities are also available to provide resource consumption reports and controls.

4.6.6. SHIFT User Interface

4.6.6.1. Disk Pools and Files

The pathname used to access files uniformly over a SHIFT configuration is referred to as the *nfs_pathname*, and has the form:

`/shift/`*shift_node/file_system_name/group/user*`/usppn`

e.g.

```
/shift/shift1/data1/c3/les/tape23
```

Here *usppn* is a "user_supplied_partial_pathname": this may be a single term (e.g. tape23), or it may include directories e.g.

```
period21/run7/file26
```

Except for *usppn*, all other components of the *nfs_pathname* are managed for the user by SHIFT via the following user commands.

The **sfget** command is used to allocate a file. This is a call to the Disk Pool Manager (DPM), which selects a suitable file system within the pool specified by the user, creates any necessary directory structure within the file system, creates an empty file, and echoes its full *nfs_pathname*. Thus the *nfs_pathname* returned by:

```
sfget -p opalpool period21/run7/file26
```

might have the form:

```
/shift/shd01/data3/c3/les/period21/run7/file26
```

This name may be used by the user in subsequent commands, such as *tpread* or a call to a FORTRAN program. The user does not need to remember the full pathname between sessions, as a subsequent sfget call using the same *usppn* will locate the file and return the full pathname again. The sfget command sets the UNIX command status to indicate if the file was created (status value 1) or simply located (value 0). This may be useful in shell scripts. If the user only wishes to locate the file if it is present, but not to create it, he may use the **-k** option of sfget:

```
sfget -p opalpool -k period21/run8/file41
```

This example will return a status of 0, and echo the *nfs_pathname* if the file exists in the specified pool, but merely return a status of 1 if it does not exist.

A user may list ("show") all of his files in a particular pool by means of the **sfsh** command e.g.

```
sfsh -p opalpool ls period21/run7
```

This *sfsh* command performs a *cd* to the user's directory in each of the file systems in the pool in turn, issuing the *ls* command in each of these directories. (The *sfsh* command is actually much more general, and can issue any shell command specified by the user in each of the file systems in the pool).

Files are removed from SHIFT pools by means of the **sfrm** command e.g.

```
sfrm -p opalpool period21/run7/file26
```

In addition to the **-p** option, specifying the disk pool to be used, all of the DPM commands support **-g** and **-u** options, which allow the caller to specify the group and user associated with the file. By default these are the group and user names of the user who issues the command.

4.6.6.2. File Access

Once a file has been created or located by the the Disk Pool Manager, and the user knows the full *nfs_pathname*, it may be used exactly like any other file. For example, suppose that we have a program called *gendata* which writes to FORTRAN Unit 10. We can allocate a file and connect it to the program as follows:

```
ln -s 'sfget sample45' fort.10
gendata
```

The first line uses *sfget* to generate the full pathname for the file, and creates a logical link to it called *fort.10* in the current directory. (FORTRAN automatically connects Logical Unit 10 to any file named *fort.10*). Normally, however, the SHIFT Remote File Access System (RFAS) will be used to access these files, in order to obtain the best possible performance. The RFAS routines are callable by the user, and have also been incorporated into certain popular CERN Program Library packages. In order to connect a file to RFAS, the **assign** command would be used. In fact, assign sets up a connection for both RFAS and for FORTRAN. The above example becomes:

```
assign 'sfget sample45' 10
gendata
```

4.6.6.3. File Staging and Tape-Disk Copying

As mentioned above, users often need to stage their data files to and from tape; on a typical mainframe job control commands are provided for this. SHIFT provides **stagein** and **stageout** commands in the form of a shell scripts that simply issues calls to the SHIFT commands *sfget*, *assign*, *tpread* and *tpwrite*. This approach is again in the UNIX spirit: taking these scripts as examples, individual users can develop their own scripts to customize their particular staging requirements or develop their own file and tape handling techniques.

4.7. SHIFT Project Status

The configuration of the pilot SHIFT system in the CERN computer center is show in Table 2.

Node Type	Name	CPU(CERN Units)	Disk(GBytes)
Apollo DN10K	hope1	16	4
Apollo DN10K	hope2	16	5
Apollo DN10KTX	hope3	32	5
SGI 340S	shift1	24	1
SGI 320S	shd01	12	20
DEC5000/200	shd03	4	10
DEC5000/200	shd04	4	10
Total		108	55

Table 2: *CERN – Pilot SHIFT System*

The addition of further system types to this pilot configuration is under review, including IBM RISC System/6000 and Sun machines. No difficulties are foreseen in incorporating these or other UNIX based systems to SHIFT.

The system is running a wide variety of physics production jobs, and has confirmed our belief that such an approach is entirely practical and economic for many physics computing applications. We intend to develop the SHIFT approach to handle interactive as well as batch applications, and work has already begun in this direction.

Design and Implementation of an Experimental Load Balancing Environment

Wouter Joosen Bruno Vandenborre
Pierre Verbaeten

K. U. Leuven, Belgium
wouter@cs.kuleuven.ac.be

Abstract

Many load balancing strategies have been proposed and simulated, but a significant amount of design issues have only been dealt with in an intuitive way. We have created a distributed load balancing system on top of UNIX. Our system is currently used as a base for evaluating design issues in load balancing strategies.

This paper describes the design and implementation of the load balancing environment. It is a prototype to experiment with, in order to produce a flexible and adaptive system, with user transparency.

1. Introduction

Many load balancing strategies have been proposed. However, a lot of design issues have been dealt with only in an intuitive way. It is our goal to experiment with a variety of strategies to assess the relative merits of a range of design issues. Therefore we have implemented a *realistic testbed* on top of BSD UNIX.

Our first experiments as well as other research on load balancing strategies have shown that no single load sharing policy is ideal in all circumstances. An *adaptive* load balancing mechanism is therefore needed.

In this paper we describe the design and implementation of a load balancing software layer in a distributed system. The environment we have implemented is both a *realistic testbed* and a firm base for an *adaptive* load balancing server.

Section two gives a general introduction to the subject. In section three, we describe the design of our load balancing environment. We justify the decision to build the testbed as a load balancing *server* (load manager). We analyse the functionality of the load balancing software, describe a general framework for our software and illustrate its value by showing how two rather different load balancing strategies can be implemented.

Section four discusses the software infrastructure we need, justifies our decision to implement our load manager in a UNIX environment and

describes the packages we have used to create the appropriate software basis. Section four also discusses the first and the actual implementation and gives a typical configuration in which our prototype is used. First experiences are summarised. We conclude by summarising the work we will do in the near future.

2. Load Balancing in Distributed Systems

2.1. Distributed Systems

We define a distributed system as a collection of computers (processor-memory pairs), interconnected by a network, and equipped with software that manages this hardware substrate. One can distinguish between this general definition of distributed systems and distributed operating systems (DOS) specifically. The latter refers to the management of all system resources in order to offer a convenient working environment for the user. We will call a system a true DOS only if the software offers the whole system as a virtual uniprocessor to the user community. As another extreme, a network operating system is implemented by a software layer on top of independent systems that run on each of the computers. In this case the physical nature of the underlying hardware model is not hidden for the users.

Between these two hypothetical models, a variety of hybrid operating systems implemente some degree of transparency.

2.2. Load Balancing

Distributed systems are attractive for economic reasons: the (theoretical) price/performance ratio is much better for a set of interconnected processor-memory pairs (*nodes*) than it is for a large mainframe with the same processing power.

These economical advantages are enforced only if we are able to make *maximal* use of the available processing power in a distributed system. This leads to the idea of load balancing: when we have several processors, we want to continuously use all the processing power that is available. We certainly do not want one processor to be idle while others are heavily loaded.

Note that we use the word *processor* or *node* where we mean *processor-memory pair*. The load on a processor-memory pair, represented by a *load index*, reflects the use of CPU and memory on that node.

The word load *balancing* sounds ambitious. We do not want to reach a perfect balance for several reasons:

- The cost for obtaining this optimum could become very high. Intuitively, one might expect a large incremental cost for evolving from an almost balanced situation to a perfectly balanced situation.
- Reaching a perfect balance is one thing, keeping the situation that way is another. Fluctuations in the environment make it rather difficult to keep the system load completely balanced. The perfect balance is inherently an unstable situation.

We want to reduce (and not necessarily *minimise*) the average job turnaround time in a system. The realisation of load sharing is a part of the general distributed scheduling problem.

2.3. Load Sharing Strategies

A number of design issues of load sharing strategies have been discussed in [Joo88a]. We use the word design *choices* to indicate those issues for which the strategy designer has to choose between a limited set of options (often only two), for instance: centralisation or not, cooperation or not, sender- or receiver-initiated strategy, deterministic or probabilistic decision rules, dynamic reallocation of jobs or not, and so on.

Although it is our intention to evaluate the merits of design issues in load balancing strategies by experimenting with them, we have already evaluated some design choices from a more theoretical point of view:

- First, the load sharing strategy cannot be implemented by a central decision maker. Such a process would introduce a serious bottle-neck in a system. Another problem would arise if such a central component fails (e.g. if the subsystem on which it resides crashes): system availability would be harmed. These observations justify the decentralised approach.

- Second, a study of existing process migration facilities [Joo88b] made us decide to base load balancing algorithms on a unique assignment of a process on a processor, primarily because process migration (dynamic reassignment) without residual dependency is an expensive operation.

The examples of strategies we use in this paper are all based on a distributed approach without reallocation of jobs. However, it remains our intention to confirm the conclusions above by experimentation. The design of the load balancing software must result in a framework in which *any* load balancing strategy can be implemented.

3. Design

The load balancing environment has been designed to simulate load balancing strategies in the first place. One might expect such a testbed to be based on a simulation programme in a traditional simulation language as for instance in [Sta85a]. Two major reasons justify the development of a distributed system to perform our simulations:

- Applying a load balancing strategy, the transfer of load from one processor to another involves two types of costs: a communication and a computation cost. One could estimate these costs and try to simulate the distributed system by inserting the values into some simulation programme. This is less accurate than building a distributed testbed to do the simulations, because *all costs* will effectively be incorporated during the experiments. No estimations have to be made at all, no cost factors are neglected.

- We want to integrate the load balancing subsystem into our "daily use" programming environment. The testbed that is used for the comparison of load balancing strategies is the prototype of a user friendly, transparent layer in our environment.

The following section explains why the distributed testbed is developed according to the client/server model.

3.1. Servers in Distributed Systems

In this section we briefly summarise the server concept. This work has been discussed extensively in [Ber87a].

Extensibility, resource sharing and reconfiguration possibilities are requirements of distributed systems that have consequences for the system's structure. In classical operating systems and in some DOS, for example Locus [Pop85a], all (or most) operating system facilities are provided in a kernel. Such a kernel is one big complicated programme from which modules cannot easily be isolated. In distributed operating systems it should be easy to add and to remove modules while a host is running. It should also be easily possible to configure the system software for a specific host from all the modules needed in this host. The model where all the system software is put together in a complex kernel is therefore not the appropriate one. The modular organisation of the system software is the main difference between these classical systems and *server oriented systems*. System resources are managed by different *servers*, which cooperate to provide all the operating system facilities. A straightforward way to implement servers is as a process running on top of a kernel.

Servers offer services to their clients. One can distinguish classes of servers according to the type of request they handle: **Independent servers:** The most simple server is one that can directly answer a request without any help of other servers (we call it therefore an *independent server*), and for which every request is independent of previous requests of the same client (we call such requests *independent requests*).

Sometimes a server needs to keep information between different requests of the same client. An example of such a server is one that manages objects for its clients; for every client the server must keep the current state of the object they access. We will call requests that are dependent on previous requests *dependent requests*. Such requests are not considered for the load balancing server because maintaining state information in the load manager would cause too much overhead.

Dependent servers: Another case arises when servers are clients of other servers. We call such servers *dependent servers*. The corresponding services are *dependent services*.

If dependent servers are constructed like independent ones and handle one request at a time, they will be idle or blocked while they wait for the answer to a request that they send to another server. If this other server runs on another host, the delay can be significant. Servers perform important tasks in the system and may become bottlenecks. Therefore these idle times must be eliminated and it must be possible to run *several requests in parallel*. A server that forwards a message instead of replying to the client acts as an independent server.

3.2. Functionality of the Load Balancing Server

The word load *manager* will be used to refer to one component (on a single node) of the distributed load balancing server.

The load manager handles an incoming job in two steps. (1) When a job has to be executed by a distributed system, it must be allocated to a specific processor. As the job is always generated at a particular processor, the first subproblem is: "Should the job be transferred to another processor in order to improve the load distribution?". This question will be referred to as the *initiation problem*. The answer is

generated, using of some *initiation policy*. (2) If the initiation policy determines that job transfer is desirable, then a *location policy* will determine to which processor the job has to be moved. This second subproblem will be referred to as the *location problem*.

Once the location problem has been solved, the job has to be transferred. This third element, the *job transfer*, is not really part of the load sharing strategy. The available software infrastructure must provide mechanisms to transfer jobs. It is however the task of the load manager to accept jobs that have been forwarded from another node, and cause the start of such job on the local node.

Two additional tasks of the load balancing server support the execution of a load balancing strategy. (1) On each node, the load index must be distributed to other nodes and (2) load indices from other nodes must be collected and interpreted. These task are not necessarily executed upon request from a client: they may be *internal* to the distributed load manager.

To summarise, at each node the load manager has to deal with four activities:

- handling user requests to start a job,
- handling requests from other load managers to start a forwarded job,
- distributing information about the local load,
- managing load information from other processors.

3.3. Framework of the Load Balancing Server

Let us apply the classification of section 3.1 to the load balancing server.

- The command interpreter of a user acts as the client of the load manager and sends a request. We can consider two cases:
  1. The load manager executes the initiation policy and concludes that the job should be executed on the local machine. This means that the location strategy will not be executed by the load manager; the request is forwarded to the process manager to start the job. From this point of view, the service as implemented by the load manager is an *independent service*, if the initiation policy is performed without any call on an underlying server.
  2. The initiation policy decides to offload the job to another machine in the system. The location policy will handle this request. This always implies interaction with a load manager on a remote host, a *dependent service*.
- The request to accept a forwarded job is an independent request.

  The normal service call on a load manager behaves as an independent service when the job is executed on the local machine. When the job is offloaded, the load manager acts as a dependent server. Consequently, several user requests must be handled in parallel. The load managers are dependent on each other, we call them *cooperatives* (see [Joo88c]).

  Several techniques can be used to implement this parallelism [Ber87a]. We prefer a lightweight process infrastructure where several threads coexist within a process. They share data and can perform communication operations independently from each other.

The thread concept is therefore used to describe the framework. Several types of logical threads coexist in the load manager.

1. A first thread will treat incoming requests from local users to start jobs. This thread will be called the "SENDER". Several threads of this kind may coexist, since it is possible that a dependent service call is executed while the next one will be independent (local execution). We should take care not to do too much redundant work. This can happen when the location strategy is executed simultaneously by two threads. The SENDER will also transfer jobs when required.

   Two threads deal with information management. Load indices must be exchanged. For example, in a receiver-initiated strategy, load indices of remote hosts can arrive at any time. The load manager must also be wary of the configuration of distributed systems: nodes may be added or removed.

2. The global information manager catches this type of information and stores it.

3. A local load manager receives information from the local load monitor, which is not part of the load manager. The local load monitor should report any relevant fluctuation in the local load index. Possibly, a tolerance on this value could be tunable. The existence of the local information manager is essential because a service call by the SENDER on the local load monitor would probably turn any call on the load manager into a dependent service call. The load monitor calls the load information manager: otherwise many of the service calls on the load monitor would be redundant.

4. Finally, one thread, the "RECEIVER" is responsible for the agreement with a sender thread to start a job. The request will be forwarded from a SENDER to a RECEIVER that finally forwards the request to the local process manager. In a sender-initiated approach, the RECEIVER may also handle incoming requests for load indices. One could also implement this activity within an information manager.

Conclusion: we need four types of threads, of which the SENDER could be instantiated more than once. This is illustrated in Figure 1.

3.4. Some Strategies

The value of our framework is illustrated by describing its use in the realisation of two quite different load sharing strategies.

3.4.1. Example 1: A Sender-Initiated Approach

In a sender-initiated approach, a sender thread is started for every incoming request to start a user job. The initiation policy determines whether the job is to be executed on the local node. In that case, the job is forwarded to the process manager.

Otherwise, the location policy is executed. Hereby, the sender thread requests the load indices from several remote sites and selects an interesting destination for the job. In the receiver thread the remote sites will receive this "request for bids" and answer the request (and possibly reserves some resources). Hereby the receiver uses the local load index that is received by the local information manager from the load monitor. When the sender has chosen a destination, the job is forwarded to the receiver thread on that site.

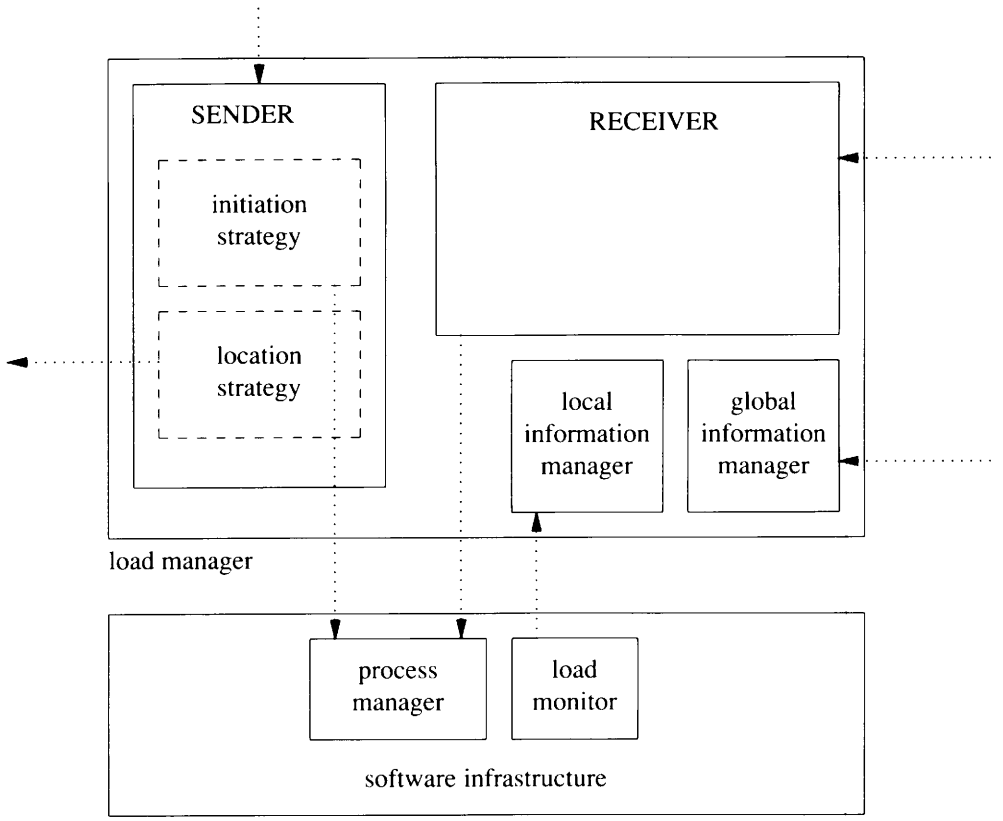


Figure 1: *Basic software architecture*

The global information manager is not handling load indices but keeps track of the system's configuration (e.g. in a distributed system, processors may be added to or removed from the system).

This scheme is represented in Figure 2.

3.4.2. Example 2: A Receiver-Initiated Approach

In a receiver-initiated approach, the sender thread executes the initiation policy as described in the former section, but it does not perform a request for bids.

It is the local information manager that distributes the local load index when it changes significantly (e.g. towards a less-loaded state). The global information manager receives such information and keeps in this way an up to date view on the system wide load distribution. The sender can use this view to select a target and only interacts with the receiver for the final agreement.

This approach is represented in Figure 3.

4. Implementation

4.1. Software Infrastructure

In this chapter, we briefly discuss some elements of the software infrastructure that is needed to implement the load balancing server. In this section, we only address the issues that are the result of the study of the framework. Other relevant elements in the software infrastructure are

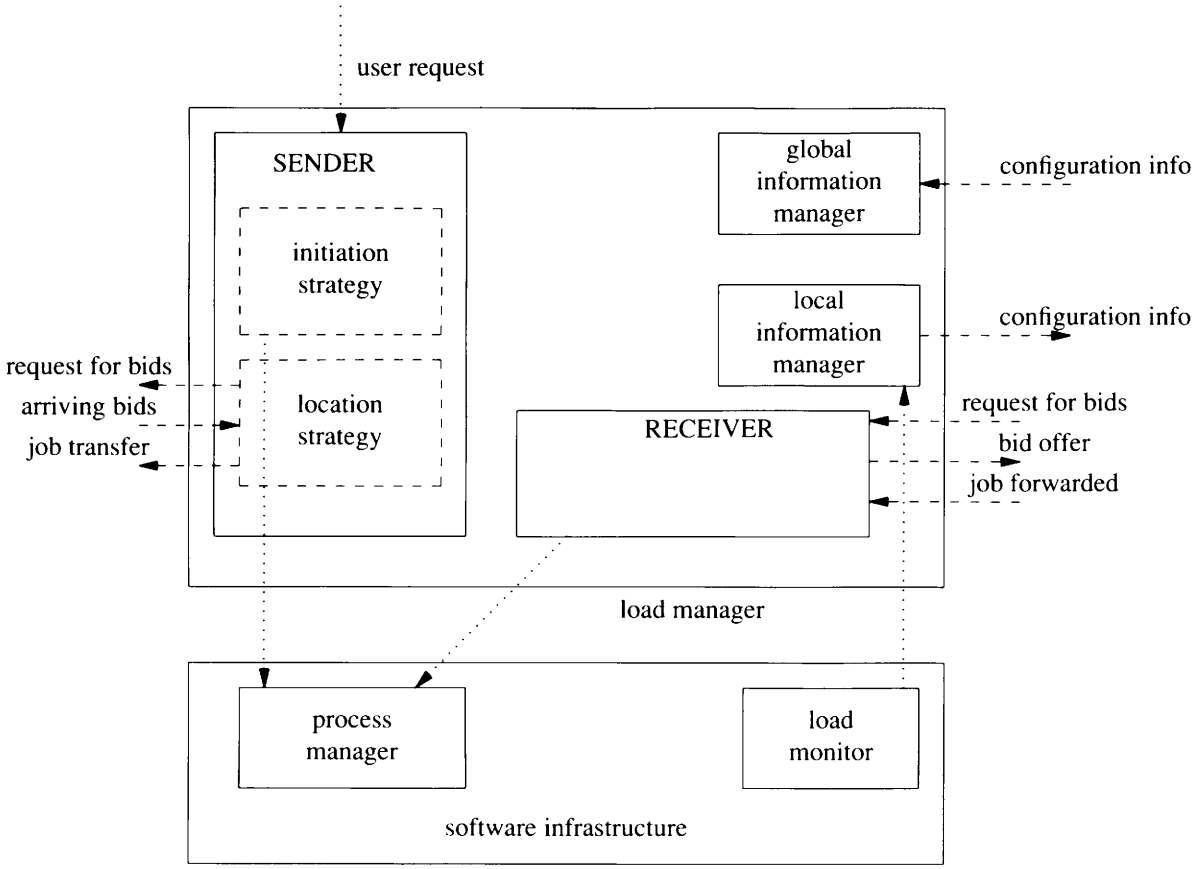


Figure 2: *Load manager: sender-initiated*

process management (a process migration facility can be considered as a part of the process manager) and a monitor for the local load (see section 4.3).

1. The load managers must be able to handle several requests in parallel. A lightweight process infrastructure is preferred: several threads coexist within a process; they share data and can perform communication operations independently from each other.

2. The load manager must be able to do set communication. Set communication is treated in more detail in [Joo88c].

   A global state must be known by all processes that are part of the distributed server: this global state is characterised by measures of the load on the different machines in the system, and possibly some knowledge of the expected fluctuation of this load in the near future. This state information should be as up to date as possible on the one hand, and the distribution of the state information should be as cheap as possible on the other hand.

   It is our belief that available communication primitives should keep as much parallelism as possible in the sending of local load information to other nodes so that updates of (or requests for) load indices happen as simultaneously as possible in the different nodes.

   For instance, when the load manager is looking for another, less loaded machine than the one it is running on (in order to start a job there), it can request bids from other machines; these are

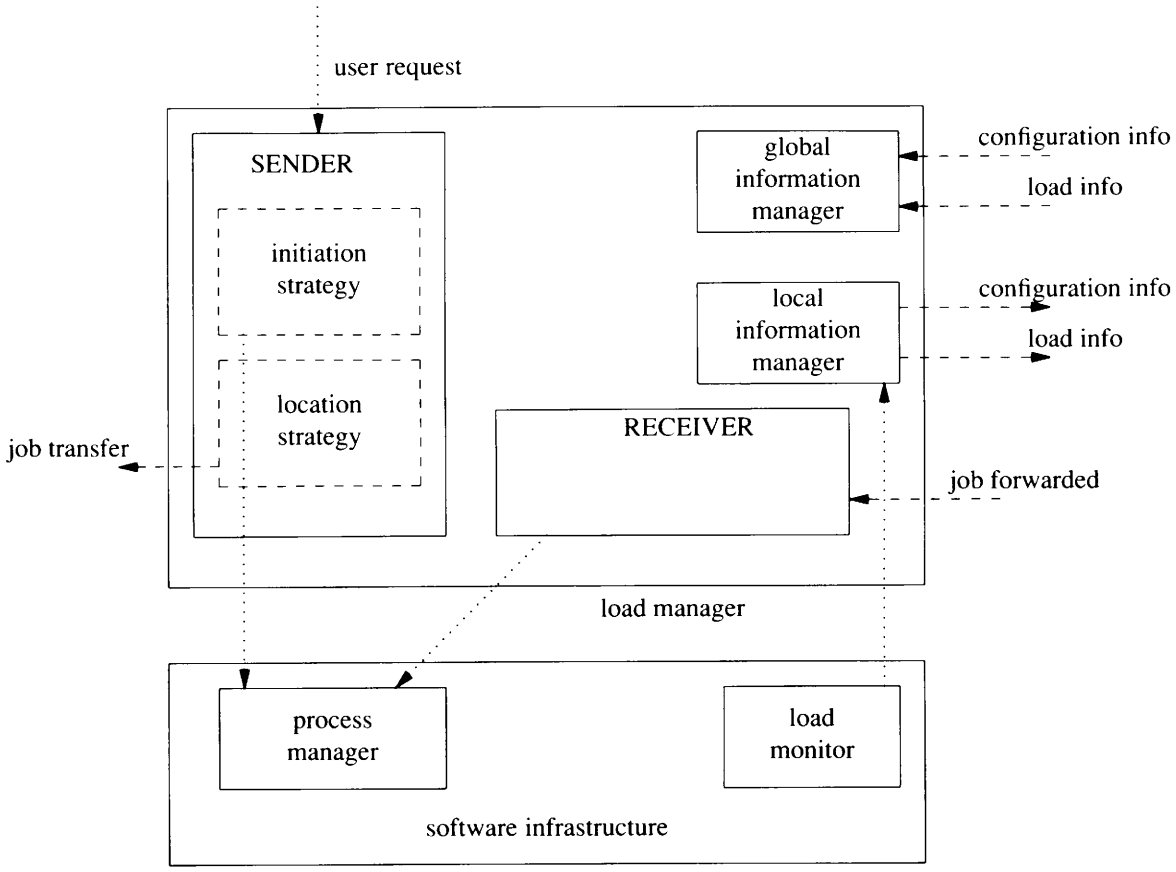


Figure 3: *Load manager: receiver-initiated*

measures of their actual load. The more parallelism in the distribution of this request, the better the arrival order of the responses becomes an indication of the system's state. This property *might* allow us to use rather simple load indices for the bids.

3. Finally, synchronisation between the different threads is required when accessing the global data in the server. A synchronisation facility makes the essential infrastructure complete.

The V-kernel is one of the few experimental distributed operating systems that gives support for both lightweight processes and set communication [Che84a, Che85a].

However, for executing numerous load balancing experiments, we prefer a system we are more familiar with in daily use. This will simplify the retrieval of appropriate load indices from the kernel.

Therefore we attempted to work on top of UNIX. Additional software libraries were made available by M. Satyanarayanan from Carnegie Mellon University. The packages we describe next have been used there for the implementation of the Andrew distributed file system [Mor86a, How88a]. Some aspects of this software have been discussed in [Sat90a].

4.1.1. Packages

4.1.1.1. The Lightweight Process (LWP) Package

This package consists of modules at two levels.

- At the lowest level is the basic lightweight process package.

  The LWP package implements primitive functions that provide basic facilities to enable procedures written in C to proceed in an unsynchronised fashion. These separate threads of control may effectively progress in parallel and more or less independently of each other. This facility is meant to be general purpose with emphasis on simplicity. Interprocess communication facilities can be built on top of this basic mechanism.

  The process model supported by the basic operations is based on a non-preemptive priority dispatching scheme. Once a given lightweight process is selected and dispatched, it remains in control until it voluntarily relinquishes its claim on the CPU. Relinquishment may be either explicit or implicit through the use of some of the basic operations. The process executing such operation will be descheduled when a process with a higher priority is ready: the priority dispatching mechanism takes over and dispatches the highest priority process automatically.

- Above this basic layer are a set of routines that only work in the context of the lightweight process package: an I/O manager package, a preemption package, a timer package and a fast-time of day package. The combination of all of these packages gives a very powerful lightweight process facility.

4.1.1.2. Set Communication

The communication facilities are RPC based. The package offers primitives that enable the user to implement remote procedure call. Set communication is possible through the *multisend* primitive. Hereby the user specifies a list of destinations and a routine (*handler*) that has to be called when a reply arrives. The use of *multisend* does not influence the LWP semantics. Servers do not know whether clients call an RPC or a *multisend*.

4.1.1.3. Locking

The lock package contains a number of routines and macros that allow C programmes using the LWP abstraction to place read and write locks on data structures shared by several lightweight processes. Like the LWP package, the lock package was written with simplicity in mind; there is no protection inherent in the model.

4.2. The Prototype

4.2.1. The Load Manager

The prototype implementation was implemented on a Sun workstation (running SunOS 3.4) and written in C. Our current organisation (see Figure 4) differs from the model in section 3 in three aspects:

1. There is no receiver thread in the load manager. Remote jobs are forwarded to the remote execution daemon *rexd*. This actually has an impact when implementing a sender-initiated strategy: handling a "request for bids" is currently implemented in the global information manager.
2. Instead of forwarding the client's request, the sender thread replies to the command interpreter specifying the node where the job is to be executed. In the case of remote execution, the com-

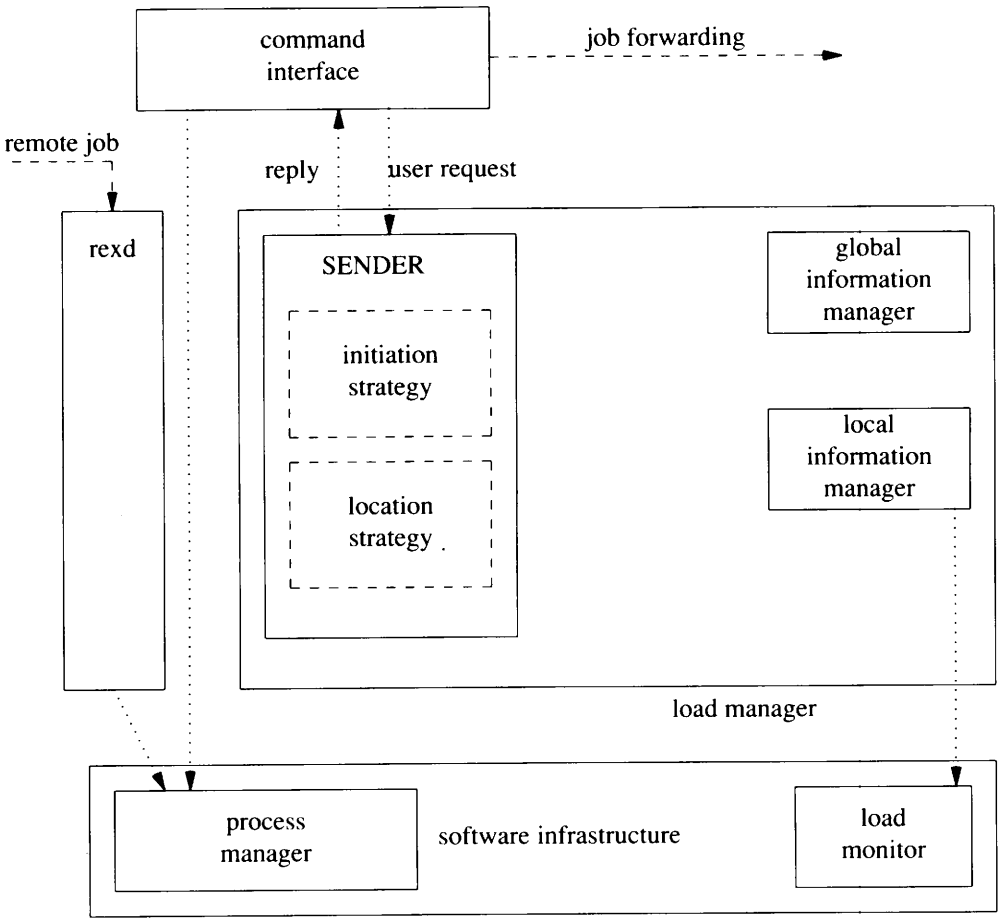


Figure 4: *Implementation*

mand interpreter uses the command *on* (see SunOS reference manual).

It is clear that this construction introduces more overhead than we would like, but it allows rapid prototyping since the remote execution daemon is present in the standard system. On the other hand, the measurement of the response time of a particular job becomes easy.

3. The local load index is requested explicitly from the local system kernel. We use the smoothed average queue length (over one minute) of the processor's ready queue. We have not yet experimented with an alternative load index but the one we use now has been recommended in [Fer88a].

   Asking the local load index regularly – instead of receiving it when it has changed significantly – is of course a disadvantage in comparison with our original proposal. It raises for instance the number of system calls the load manager makes. Implementing this in another way turns out to be a difficult problem [Kup85a].

4.2.2. A Typical Test Configuration

Figure 5 shows the test environment that has typically been used: 5 diskless workstations (the processor pool), do each run the load manager and a command interpreter. This one reads commands from a script and sends the response time measurement of a job to the coordinator machine of the experiment. The coordinator machine is a com-

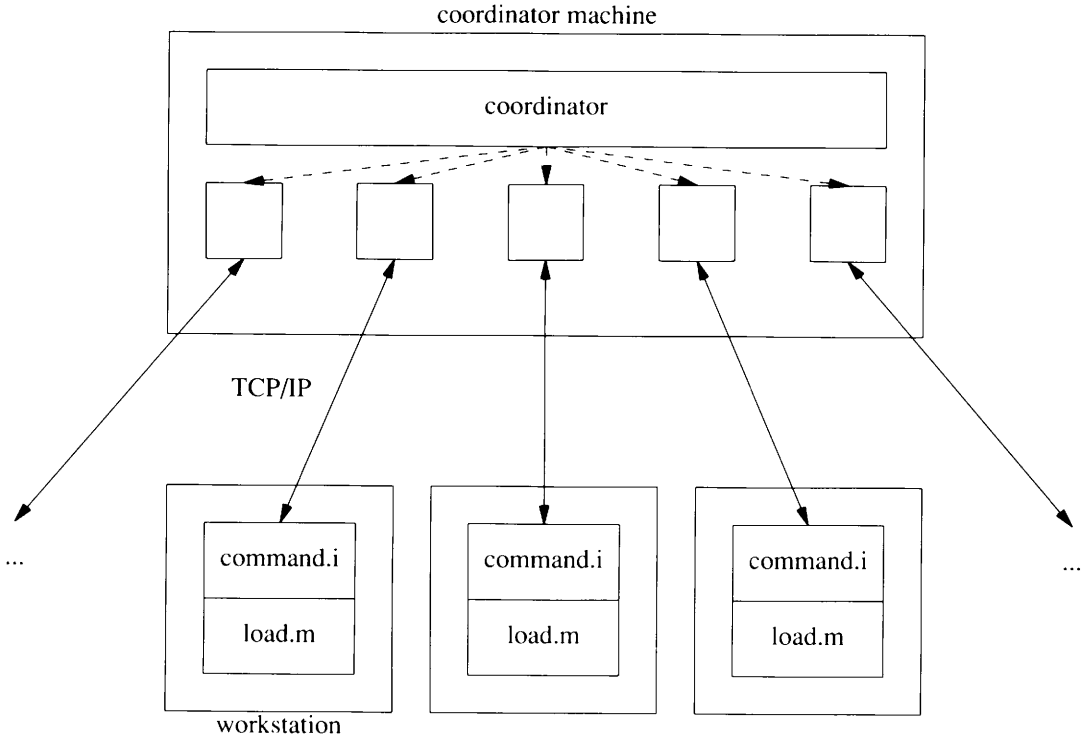


Figure 5: *Test configuration*

puter that is not in the pool and that contains all the files that are needed in the experiment. File transfer is an identical cost for all the nodes.

For each workstation, a process is running on the coordinator machine to keep track of the observed response times. All communication between the coordination machine and the processor pool is done through TCP/IP connections.

The coordinator is a process that sets up these connections and provides a synchronised start of the experiments.

4.2.3. First Experience

Several load sharing strategies have been implemented in the testbed, mainly to observe the behaviour of the environment itself.

We started by observing strategies that had been studied in earlier research [Eag86a, Hsu86a]. In a first phase, 6 strategies have been used.

Experiments have been done with two objectives in mind.

- Comparison with other results: are our observations acceptable in relationship with other results? Differences must be explained.
- Optimisation of individual strategies: each strategy is written in terms of a number of variables, e.g the period of information exchange, the number of nodes to cooperate with, threshold values on which the initiation policy relies etc. Such *parameters* must be tuned to obtain optimal results with a given strategy.

The detailed interpretation of our measurements is not the subject of this paper. We only indicate some general conclusions:

- First experiences approve the generality of the testbed.

- A lot of experiments are required to optimise a single load sharing strategy. This indicates that a careful selection must be made to limit the experiments to the "useful" ones.

  Design *choices* will be treated first in order to reduce the set of strategies we want to work with in the long run. For example, experiments with alternative load indices will be done with a selected set of high quality strategies.

- Overhead and delays are often underestimated in traditional simulations. The benefits we obtain with the given strategies are smaller as predicted in the papers mentioned above.

These comments are not totally precise. We have to continue with our experiments and will soon report detailed information.

4.3. The Experimental Environment

4.3.1. Porting the Prototype

We have ported the testbed to the AIX environment. First we ported the lightweight process package, together with the locking package and the RPC code. Afterwards, the transfer of the load manager was straightforward.

The port has two major consequences.

- First, the organisation of the testbed is different. We run AIX on 5 IBM PS/2 stations. Each of them has a hard disk. Therefore, the testbed writes the response time of a job on the disk local to the node on which the job has been generated. This reduces network traffic substantially since the prototype implementation ran on a collection of diskless workstations.

- Second, the possibilities of the testbed are enlarged by installing TCF [Wal89a] (Transparent Computing Facility) on our machines. TCF is a Locus based [Pop85a] extension of AIX in which process migration is supported. This facility allows the investigation of load balancing strategies that are based on the dynamic reallocation of jobs.

4.3.2. Improvements

The efficiency of the load manager is being improved by working on two aspects. (a) The retrieval of a load index is done efficiently, actually by making a device driver, the load monitor, that looks in the kernel's statistics. The number of system calls that is needed to obtain the load index is reduced to one instead of two. (b) The receiver code is being integrated in the load manager. This will allow the effective forwarding of jobs from one load manager to another.

The current implementation as described is sufficient as it is our objective to compare the relative merits of certain design issues in a load sharing policy. Improvements as explained in this section are only important in the long run.

4.4. Future Work

4.4.1. Adaptive Load Sharing

One of the main objectives of our research is to know the relative merits of a number of design issues. Experiments have to tell us which

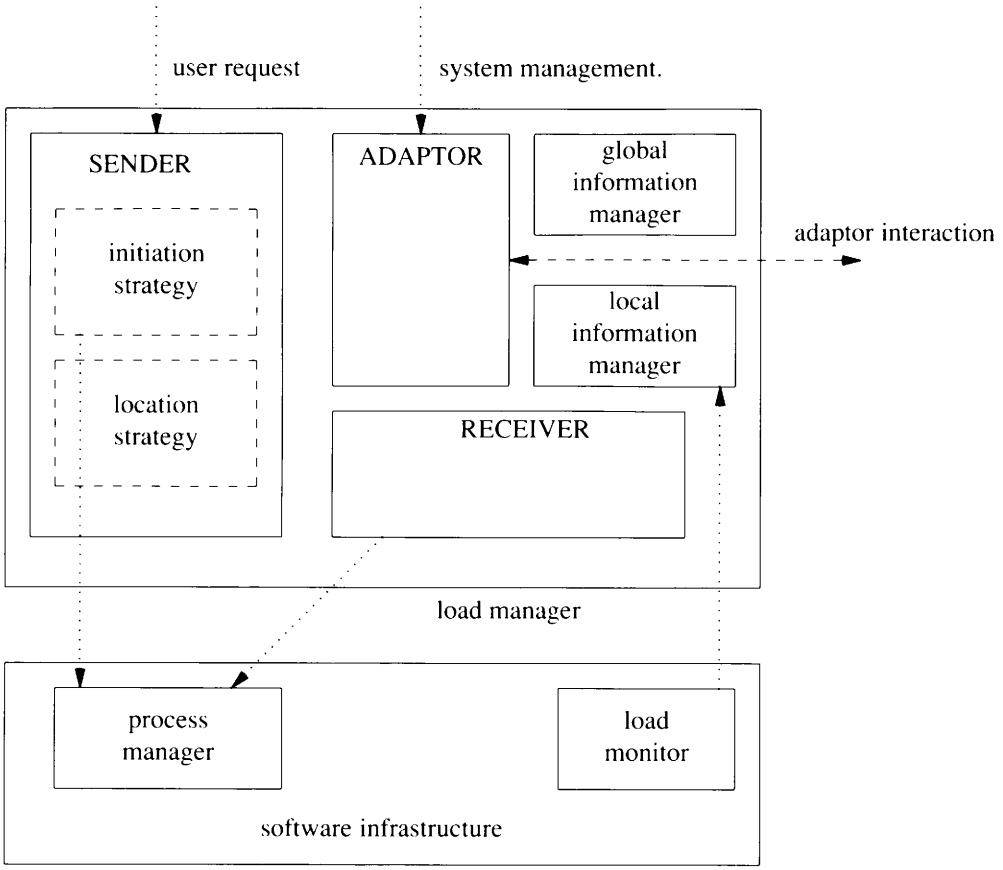


Figure 6: *Adaptive load manager*

strategy is appropriate in given circumstances. This knowledge is the base for adaptive load sharing software: strategies may vary as the system state evolves.

This means that there is another level of control inside the load sharing software: a thread that is authorised to change the policies. It is called the *adaptor*. The adaptor can for instance select another routine that implements the location strategy (e.g. from a deterministic into a probabilistic one).

In this concept, the adaptor will *initialise* the load balancing server and subsequently observe the system's behaviour in order to adapt to the evolution.

Some changes only affect the local load manager and can therefore be make at any time, but more global adaptations may require a consensus between adaptors. Such changes are not intended to occur frequently since they will involve quite some overhead.

Figure 6 shows the extended model.

This topic is still under development and partially dependent on the results of the experiments. In the actual implementation, the adapter only initialises the load manager by providing functions (code) for each of the threads.

4.4.2. Extended Command Interpreter

Our distributed system consists of the physical layer, equipped with the basic software infrastructure (AIX/TCF extended with the libraries from CMU). The load sharing software is situated on top of this native sys-

tem. The user's command interpreter acts as a client of the load manager. The latter only acts as a dependent server when interaction with remote components of the cooperative is required.

In order to exploit the full power of the load balancing server, the command interpreter should be able to recognise large grain parallelism in a job. For example, many shell scripts contain a list of command lines that are executed sequentially, even though many of these commands could be executed simultaneously. Currently, parallelism can only be caused by starting background jobs; but this method excludes the possibility to synchronise.

The command interface is responsible for passing such parallel subtasks separately to the load manager. It is however, up to the writer of the script to express the potential parallelism in the job.

This leads to an extension of the syntax of the shell. We intend to adapt the Korn shell for this purpose. Some ideas are found in the CDL (component description language) of Helios [Per89a].

5. Conclusion

Our design describes the load balancing software as a subsystem with a limited interface to the other system components. It is suited for a very general system model. The actual implementation enables us to integrate relevant load balancing strategies into the framework. The steps towards an adaptive subsystem that will be integrated in our "daily use" distributed environment are undertaken. Other aspects still have to be worked out: how can we cope with large scale systems, and with a de facto heterogeneous environment?

A series of experiments will assess the merits of a number of design choices in load balancing strategies. This work will deliver a guideline in the development of an adaptive distributed server, in which the adaptor selects policies out of a set of well experimented strategies.

References

[Ber87a] Y. Berbers and P. Verbaeten, "Servers, Processes and Subprocesses: a Critical Evaluation," *Report CW 60*, Leuven, Department of Computer Science K.U.Leuven (May 1987).

[Che84a] David R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software* **1**, nr. 2 (April 1984).

[Che85a] David R. Cheriton and Willy Zwaenepoel, "Distributed Process Groups in the V-Kernel," *ACM Transactions on Computer Systems* **3**, nr. 2 (May 1985).

[Eag86a] Derek L. Eager, Edward D. Lazowska, and John Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Performance Evaluation* **6**, pp. 53-68, Elsevier Science (1986).

[Fer88a] D. Ferrari and S. Zhou, "An Empirical Investigation of Load Indices for Load Balancing Applications," *Performance'87*, pp. 515-528, Elsevier Science Publishers B.V. (1988).

[How88a] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West,

"Scale and Performance in a Distributed File System," *ACM Trans. on Comp. Systems* **6**, pp. 51-81 (February 1988).

[Hsu86a] Chi Yin Huang Hsu and Jane W.-S. Liu, "Dynamic Load balancing Algorithms in Homogeneous Distributed Systems," *Proceedings of the 6th International Conference on Distributed Processing*, pp. 216-223 (May 1986).

[Joo88a] W. Joosen and P. Verbaeten, "Design Issues in Load Balancing Techniques," *Report CW81*, Leuven, Department of Computer Science K.U.Leuven (April 1988).

[Joo88b] W. Joosen and P. Verbaeten, "On the Use of Process Migration in Distributed Systems ," *Report CW83*, Leuven, Department of Computer Science K.U.Leuven (November 1988).

[Joo88c] W. Joosen and P. Verbaeten, "Set Communication in Distributed Systems ," *Report CW82*, Leuven, Department of Computer Science K.U.Leuven (November 1988).

[Kup85a] M. D. Kupfer, "An Appraisal of the Instrumentation in Berkeley Unix 4.2BSD.," *Report UCB/CSD 85/246* , Computer Science Division, University of California, Berkeley (1985).

[Mor86a] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM* **29**, pp. 184-201 (March 1986).

[Per89a] Perihelion, "The Helios Operating System," *Perihelion Software Ltd.*, Prentice-Hall (1989).

[Pop85a] Gerald Popek and Bruce J. Walker, "The Locus Distributed System Architecture," *MIT Press*, Cambridge, Massachusetts (1985).

[Sat90a] Mahadev Satyanarayanan and Ellen Siegel, "Parallel Communication in a Large Distributed Environment," *IEEE Transactions on Computers* **39**, pp. 328-348 (March 1990).

[Sta85a] John A. Stankovic, "Stability and Distributed Scheduling Algorithms," *IEEE Transactions on Software Engineering* **11**, pp. 1141-1152, no. 10 (October 1985).

[Wal89a] Bruce J. Walker and Richard M. Mathews, "Process Migration in AIX's Transparent Computing Facility (TCF)," *Operating Systems Technical Committee Newsletter* **3**, pp. 5-7 (Winter 1989).

A Decentralized and Efficient Algorithm for Load Sharing in Networks of Workstations

Guy Bernard

Institut National des Télécommunications
Evry, France
guy@bdblues.altair.fr

Michel Simatic

Alcatel TITN Answare
Massy, France

Abstract

This paper presents the design and evaluation of a decentralized load sharing algorithm for networks of workstations, RADIO. With respect to general distributed computing environments, networks of workstations have some peculiarities. First, the global computing power is most of the time much underutilized. Second, users of workstations occasionally need a peak of computing power. Third, workstations are often diskless, so that running a process on one workstation or another does not add file migration overhead. Fourth, network interfaces often provide a broadcast capability, which may be used to reach several destinations in a single message. Last, workstations are often dedicated to an "owner", so that a workstation may only be used for running foreign processes only when the workstation's owner does not use it (or at least when running foreign processes would not increase the owner's programs response time by a significant amount). The first three points make load sharing very attractive for a network of workstations. The fourth point may be used for simplifying the design of load sharing algorithms, but broadcasting is expensive. The goal of RADIO is to provide the benefits of a decentralized load sharing algorithm while preserving the personal character of workstations and providing good performance results, in particular with respect to extensibility.

The key feature of the RADIO load sharing algorithm is that it is decentralized but involves expensive broadcast messages only occasionally. The design choices for information policy, location policy and transfer policy are described and motivated. RADIO has been implemented on a network of Sun workstations, and runs entirely outside of the kernel. Experimental results show that the extensibility of RADIO is better than that of previous decentralized algorithms, based on broadcast messages.

1. Introduction

There are basically three ways of improving performance in loosely coupled distributed computing systems. The first one consists in implementing a file location/migration policy. The second one consists in taking benefit of the parallelism inherent to some applications by running in parallel, on different processors, the different tasks that constitute the application program. The third one, usually referred to as "load sharing", consists in taking benefit of the fact that, in the network, some machines are less loaded than others (or even totally inactive), by running some processes on a less loaded machine. This paper focuses on a load sharing policy.

Among loosely coupled distributed computing environments, a network of workstations has some peculiarities. First, global computing power is underutilized most of the time [Mut87a, Stu88a, The89a]. Second, the owner of a workstation needs occasionally a peak of computing power, high enough to lead to slow response times if the processes are run simultaneously on his/her workstation. Third, workstations are often diskless, so that running a process on one workstation or another does not add file migration overhead. Fourth, network interfaces often provide a broadcast capability, which may be used to reach several destinations in a single message. Last, workstations are often dedicated to an "owner", so that a workstation may only be used for running foreign processes when its owner does not use it (or at least when running foreign processes would not increase the response time of the owner's programs by a significant amount). This paper focuses on a load sharing policy in a network of personal workstations. The goal is to provide the benefits of load sharing (that may be large for underutilized systems) while preserving the personal character of workstations and providing good performance results, in particular with respect to extensibility (defined as the maximum number of workstations that may be part of the system without consuming too much CPU or network bandwidth).

The problem of performance and extensibility of load sharing algorithms has been addressed in several papers [Zho88a, The89a]. The main conclusions are: (i) broadcasts on a local area network are expensive; (ii) centralized algorithms are, surprisingly, the most extensible; (iii) failure detection and recovery are expensive for centralized algorithms.

The goal of this paper is to describe and evaluate a decentralized algorithm that involves broadcast messages very occasionally, and thus provides large extensibility and good performance results.

2. The RADIO Algorithm

A load sharing algorithm is composed of three parts [Zho88a]. The information policy specifies which information is used in deciding a process migration, and the way this information are distributed in the system. The location policy decides on which machine an eligible process should be migrated. The transfer policy determines the eligibility of a process for migration. We now describe these three parts for the RADIO algorithm and then the broadcast cases.

2.1. Information Policy

In RADIO, information and decision are both decentralized and centralized. A workstation is *available* when it can run remote processes. At any time, RADIO handles the following data structures:

1. Every workstation W_i keeps in memory the identity of the last workstation L_i that accepted a process from W_i, i.e. that was available at that time.
2. The available workstations are linked in a distributed ordered list, the "available list" (each available machine A_i knows only the identity of its predecessor A_{i-1} and successor A_{i+1} in the list).
3. A machine called the "manager" plays a special role. It knows the identity of the first workstation A_1 of the available list. The identity of the manager is known by all the workstations of the network. Workstation A_1 knows it is the first workstation of the available list as its predecessor is the manager itself.

When workstation W_i wishes to run a process P on a remote machine:

1. It first polls its (supposed) available partner L_i. This is based on the assumption that an L_i that was available before is still available now. If L_i is really available, it accepts process P (see Figure 1).
2. Of course, this assumption does not always hold. If L_i is not available, it forwards the request to the manager (see Figure 2). If the available list is not empty, the manager indicates the identity of the first workstation of the available list, A_1, to the requesting workstation W_i. If the available list is empty, the manager notifies W_i that no workstation is currently available so that W_i executes process P locally.

When a workstation A_i switches from available state to non-available state, it notifies its predecessor A_{i-1} in the available list that the succes-

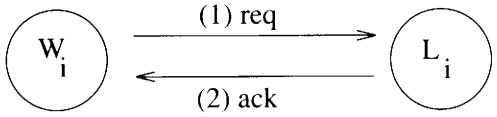


Figure 1: *Finding an available workstation in two messages*

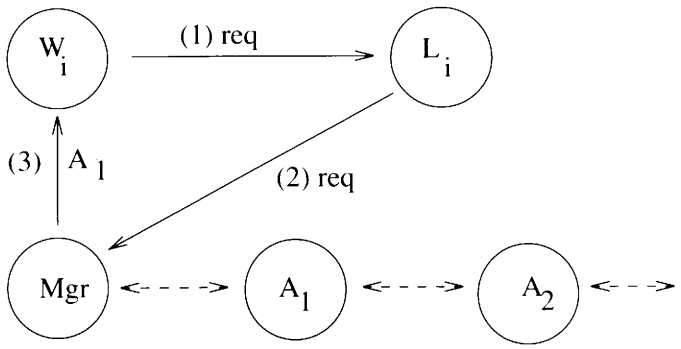


Figure 2: *Finding an available workstation in three messages*

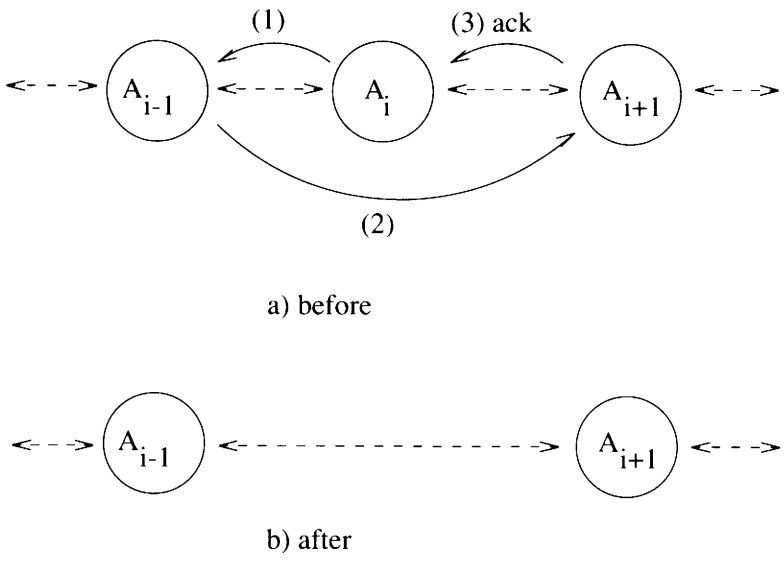


Figure 3: *Switching from available to non-available state*

sor of A_{i-1} is now A_{i+1}. Afterwards A_{i-1} notifies A_{i+1} that it is now its predecessor (see Figure 3).

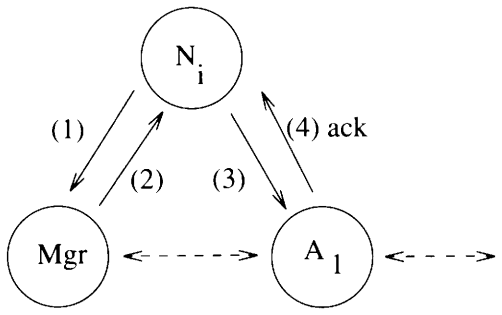
When a workstation N_i switches from non-available state to available state, it notifies the manager, which sends back the identity of the first available workstation A_1. Then, N_i notifies A_i that it is now its predecessor. Thus, N_i is inserted at the head of the available list, in location A_1 (see Figure 4).

2.2. Location Policy

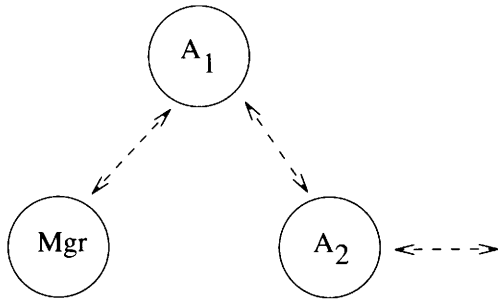
The goal of a load sharing policy may be to attempt to balance machine loads ("load balancing") at any time over the network, or simply to ensure that no machine is idle when others are overloaded ("load sharing" strictly speaking) [Kru87a]. For networks of personal workstations, load balancing is not only useless, but undesirable, for two reasons. First, since most of the time the workstations are underloaded, load balancing would most of the time migrate a process from a workstation less loaded to another even less loaded, so that migration overhead would dominate the gain in execution time [The89a]. Second, since workstations are personal, the owner would not tolerate a significant degradation of his/her response times because another user started many cpu intensive computations.

Several ways for taking into account the personal nature of a workstation have been proposed. In Butler [Nic90a], a workstation is unavailable for foreign processes as soon as the number of logged in users is greater than some threshold. This criteria is very restrictive, because if users neglect to log out when they do not use the workstation, it will appear busy whereas it is in fact available. In Condor [Lit88a], a workstation is declared available when there have been no keyboard or mouse activity for some duration, and the average CPU utilization has been less than some value for a certain amount of time. This criteria is very restrictive too, because if the workstation is used only to run an editor, it will appear as busy, while running a foreign process would not slower the editing process.

In RADIO, as in [Alo88a], a workstation is declared available when its current load is less than a threshold T_{low}. The load index used is the



a) before



b) after

Figure 4: *Switching from non-available to available state*

UNIX 4.3BSD index, namely a mix of averaged CPU and IO queue lengths. The value of T_{low} need not be the same on every workstation, and it may be changed dynamically by the system administrator if needed. Empirically, the value of 0.8 appeared to be suitable. In our experiments, users did not notice that their workstation was executing foreign processes with this value.

2.3. Transfer Policy

As stated before, the transfer policy determines the eligibility of a process for migration. In RADIO, a process may be migrated to an available workstation (if such a workstation exists) only when the workstation becomes overloaded, namely when its current load is above a second threshold, T_{high}. In order to prevent boomerang effects, the value of T_{high} must be at least $T_{low} + 1$. The value of 1.8 appeared to be suitable. However, as for T_{low}, T_{high} need not be the same on every workstation. To summarize the influence of T_{low} and T_{high}, a workstation may be in one of three states, according to the value of its current load:

1. $load < T_{low}$: the workstation is "available". It may receive foreign processes.
2. $T_{low} < load < T_{high}$: the workstation is in "normal state". It will not accept new foreign processes.
3. $T_{high} < load$: the workstation is "overloaded". It will try to send one or more process to an available machine.

This double threshold scheme is very flexible. For instance, if T_{low} is set to 0 on a workstation, this machine will never accept remote processes.

Furthermore, in RADIO the processes are not all eligible for migration. Cabrera has observed that a large majority of UNIX processes consume less than $1s$ CPU [Cab86a]. Hence, a filtering policy must be enforced. This filtering may be done on the process type, such as in [Sve90a]. However, while a compilation lasts longer than $1s$ CPU in average, some compilations are very short, in particular because of errors. In RADIO, we chose manual filtering. By default, processes are not candidates for migration. If a user wishes that some process be a candidate (because a long execution time is expected), he/she starts the process by a special command at shell level.

2.4. Broadcast occurrences

In RADIO, broadcasts are occasional. Under normal system behaviour (no failures), broadcasts occur in the following cases:

1. When a workstation joins the process migration facility (at boot time, for instance). This workstation broadcasts a message requesting the identity of the manager. If no answer is received, this workstation becomes the manager.
2. When the manager becomes overloaded. In this case, the workstation acting as the manager experiences larger response times not only for the processes started by his/her owner, but for the load sharing algorithm too. For these two reasons, the manager has to be replaced. The overloaded manager sends a message to the workstation at the head of the available list (A_1). A_1 then becomes the manager, and broadcasts this fact on the network.

In a few cases of workstation failure, RADIO requires broadcasts too, as will be described in Section 3.1.4.

3. Evaluation of RADIO

In this section we evaluate RADIO algorithm, first by looking at its properties, and then by reporting experience with running RADIO.

3.1. Properties

3.1.1. Quality

If there are available workstations in the network, RADIO is able to find one, either because the workstation presumably available is really available, or when the manager is involved, because the manager points at the head of the available list. Notice that in the latter case the available workstation selected is the "best" choice, since the head of the list is the machine having switched to available state the most recently, thus its probability of switching back to unavailable state in the meanwhile is low.

3.1.2. Efficiency

The location policy involves two messages (request and ack) when the workstation supposed to be available is really available, and three messages when it is not (request, forward to manager, identity of A_1). The

underlying assumption for our design is that the workstation supposed to be available is often really available (this assumption is validated by our observations as will be seen in Section 3.2). Thus, the average number of messages for selecting an available workstation is less than three. Furthermore, the overhead imposed on the manager by the algorithm is lower.

3.1.3. Extensibility

Extensibility (defined as the maximum number of machines that the algorithm can reasonably take into account) is studied with the parameter values set by Theimer and Lantz in [The89a]. In order to be efficient, a load sharing algorithm should select a receiving machine for a process in less than $100ms$, consume less than 1% of CPU cycles on any machine, and consume less than 1% of network bandwidth. Furthermore, Theimer and Lantz assume that program generation leads to running the process assignment algorithm once per minute in average on every machine, and that, for algorithms involving periodic information emission, the interval between two emissions is $10s$.

With these parameters, Theimer and Lantz compare two algorithms: CENTRAL and DISTRIBUTED (also described in [Zho88a]). CENTRAL is a centralized algorithm: when an overloaded workstation wishes to transfer a process, it asks a server for an available workstation, if there is one. The server is informed of the availability of any workstation in the system by means of messages sent to it by every workstation in the system, either periodically [Hag86a, Bon89a], or when its load has changed by a significant amount [The89a]. DISTRIBUTED is a fully decentralized algorithm: when an overloaded workstation wishes to transfer a process, it polls all the workstations in the system by a broadcast message. The workstations reply by sending their current load, thus the requesting workstation may find a suitable receiver, if any. In order to avoid buffer overflow that may result from the reception of many simultaneous answers, Theimer and Lantz propose the following mechanism: only the workstations with reasonable load reply to polling messages, and the reply is delayed by a small time increasing with the local load, so that the first replies received are probably the most interesting.

With a simple model, Theimer and Lantz show that the maximum number of workstations that CENTRAL and DISTRIBUTED can handle are 700 and 85, respectively.

Under the same assumptions, we found the maximum number of workstations RADIO can handle is 282. This result is quite sufficient for most workstation networks. The reason why extensibility of RADIO is better than extensibility of DISTRIBUTED is that broadcasts are occasional.

3.1.4. Robustness

All workstation failures are detected by a timeout mechanism.

- If the failure occurs on a non-available workstation, this failure has no consequence on the load sharing facility.
- If the failure occurs on an available workstation member of the available list, this failure is detected by its predecessor A_{i-1} or its successor A_{i+1} when one of those two workstations wants to notify it has switched from available to non-available state. In the case of A_{i+1} detecting the failure, this workstation broadcasts

a message requesting predecessor of A_i to become the predecessor of A_{i+1}.

- In case of manager failure, the workstation that detects the failure broadcasts a message requesting the manager replacement. Then, workstation A_1 in the available list (this workstation knows that it is at the head of the list as its predecessor is the "–defunct–" manager) becomes the new manager and broadcasts this fact on the network.

As in CENTRAL, this case of failure is the worst one. But, in CENTRAL, recovery after a manager failure is complex, since the (centralized) information about system state is lost. In RADIO, recovery is much simpler, since the (distributed) available list need not to be reconstructed. The simple election algorithm described above sets up again the migration facility in two broadcast messages. Notice that the migration facility is not interrupted at all for a significant number of workstations, since the manager is not always involved in the location mechanism.

3.2. Experience

RADIO has been implemented and monitored on a network of 8 workstations Sun-3 running SunOS 3.5.

First, an artificial load was generated in order to observe the behaviour of RADIO under severe conditions:

- On three of the workstations, we simulated heavy users running compilations (taking $50s$ on an empty workstation) every $3min$ and trying their CPU-intensive program consisting in a set of sinus computations (requiring $60s$ on an empty workstation) every $2min$. The editing sessions were simulated by sleep mechanisms.
- On another workstation, we simulated a user making data analysis by first entering data with an editor (simulated by sleeping), making some computation (requiring $60s$ on an empty workstation) every $6min$ and running his/her analysis (made of 2 simultaneous CPU-intensive processes requiring $5min$ on an empty workstation, each), every $70min$.
- The four remaining workstations were unused and thus available.

The results are the following:

- About 60% of locations were resolved in two messages. This validates the "caching" mechanism: most of the time, the last workstation that accepted to receive a process is available again.
- The average rate of broadcasts is one per $266s$. Under the same conditions, DISTRIBUTED would lead to a rate of one per $51s$ (every request for remote execution requires a broadcast).
- The overhead of running RADIO is negligible: less than $10ms$ CPU per hour (0.0002% of CPU cycles), and one 10-bytes message every $4s$ in average (0.0002% of network bandwidth).
- The measured recovery procedure duration is about $500ms$, to be compared to $18s$ with CENTRAL algorithm [The89a].
- The average time for locating an available workstation is $30ms$, and the average time for knowing that no workstation is available is $500ms$. The latter result is large, because when no workstation is available, all the requests are forwarded to the manager, thus its load is high and its response time is large.

- When a process is to be executed remotely, the initialization time (transferring process information and setting up the communication channels for interactivity) is about $500ms$. This shows that load sharing is efficient as soon as process execution time is larger than a few seconds.

However, the results above were obtained under high artificial load. In normal use, one can expect a larger efficiency of the "caching" mechanism, and thus better results.

For instance, compiling the 7 modules of the RADIO source files in parallel on several workstations requires $59s$, to be compared to $104s$ locally. Four CPU intensive processes (a set of sinus computations) are placed by RADIO on four workstations and executed in $263s$, whereas the elapsed time is $1035s$ locally. This result shows that load sharing with RADIO can lead to a speedup close from the optimum for long, CPU-intensive, applications.

4. Implementation details

RADIO runs entirely outside the kernel. The RADIO daemon (running on every workstation) is written in C by the means of an automaton.

Interactivity is preserved by a mechanism analogous to UNIX remote shell: when a process runs remotely, keyboard input (including signals) and screen output are done locally, in a way transparent for the user.

Basically, the load index used for deciding whether a workstation is in "available", "normal" or "overloaded" state is the UNIX 4.3BSD load index (average of the number of processes ready to run or waiting for disk I/O to complete, as sampled over the previous 1-minute interval). However, using this index directly cannot take into account in a correct way several requests for process execution on the same workstation in a short time interval, because the load index of the available workstation does not react quickly enough: the last requests will consider the workstation as still available whereas it has just accepted to run several remote processes. Thus, in our implementation the load is computed on every workstation by a dedicated process, which computes the average load exactly as in UNIX 4.3BSD, except that when a remote process is accepted the current load is incremented immediately by one unit.

5. Conclusion

The design and implementation of RADIO has shown that it is possible to build a load sharing policy that is decentralized (thus robust with respect to failures) while avoiding expensive broadcast messages. Extensibility of RADIO is better than extensibility of previous decentralized algorithms, based on broadcasts. Thus, RADIO is a good candidate for medium/large configurations when robustness is an important factor.

The double threshold scheme used for location and transfer policies is very flexible and is well adapted to networks of workstations, because it can conciliate load sharing and personal use of workstations. In RADIO, the threshold values are set empirically. It would probably be possible to tune these values dynamically, according to system state.

RADIO is currently being evaluated on a larger number of workstations (about twenty) with trace-driven process patterns. We are also thinking about how to take machine heterogeneity into account, not only in

terms of CPU speed, but also available memory size and local disks usage.

References

[Alo88a] R. Alonso and L. L. Cova, "Sharing Jobs among Independently Owned Processors," in *Proc. 8th Int. Conf. on Distributed Computing Systems*, San Jose, California (June 1988).

[Bon89a] F. Bonomi, P. J. Fleming, and P. D. Steinberg, "Distributing Processes in Loosely-Coupled UNIX Multiprocessor Systems," in *Proc. USENIX Summer '89*, Baltimore, Maryland (June 1989).

[Cab86a] L. F. Cabrera, "The Influence of Workload on Load Balancing Strategies," in *Proc. USENIX Summer '86*, Atlanta, Georgia (June 1986).

[Hag86a] R. Hagmann, "Process Server: Sharing Processing Power in a Workstation Environment," in *Proc. 6th Int. Conf. on Distributed Computing Systems*, Cambridge, Mass. (May 1986).

[Kru87a] P. Krueger and M. Livny, "The Diverse Objectives of Distributed Scheduling Policies," in *Proc. 7th Int. Conf. on Distributed Computing Systems*, Berlin (September 1987).

[Lit88a] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor – A Hunter of Idle Workstations," in *Proc. 8th Int. Conf. on Distributed Computing Systems*, San Jose, California (June 1988).

[Mut87a] M. W. Mutka and M. Livny, "Profiling Workstations' Available Capacity for Remote Execution," in *Proc. Performance '87, 12th IFIP WG7.3 International Symposium on Computer Performance* , Brussels (December 1987).

[Nic90a] D. A. Nichols, "Multiprocessing in a Network of Workstations," *Ph.D. Thesis, CMU Report CMU-CS-90-107, Carnegie-Mellon University* (February 1990).

[Stu88a] M. Stumm, "The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster," in *2nd IEEE Conference on Computer Workstations* (March 1988).

[Sve90a] A. Svensson, "History, an Intelligent Load Sharing Filter," in *Proc. 10th Int. Conf. on Distributed Computing Systems*, Paris, France (May 1990).

[The89a] M. M. Theimer and K. A. Lantz, "Finding Idle Machines in a Workstation-Based Distributed System," *IEEE Trans. on Software Engineering* **15**(11) (November 1989).

[Zho88a] S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," *IEEE Trans. on Software Engineering* **14**(9) (September 1988).

Distributed Applications in Heterogeneous Environments

Bertil Folliot

Laboratoire MASI, Paris, France
folliot@masi.ibp.fr

Abstract

In a system where many hosts are connected by communication networks, the choice of programs placement allows to benefit from the processing power of all hosts (idle or unused) and thus to reduce the program response time. In this paper we present a solution to execute and to control distributed applications in heterogeneous environment. We consider an application as a set of programs linked by a precedence graph. Each program may be allocated on different heterogeneous hosts and may specify different allocation criteria.

We have implemented the GATOS system to automatically distribute parallel applications among heterogeneous hosts and to provide a software layer in order to easily write new distributed applications in a heterogeneous environment. GATOS has been developed to be portable and to work in a network containing a large number of hosts.

Introduction

Modern computer systems are often built with many workstations connected by communication networks. In such workstation networks, many hosts are unused or idle at a given time. Some may be free, others may be unused during thinking time of their users. However in most present systems, users who need a large amount of processing power cannot *automatically* benefit from the idle workstation power [Mut88a, Lan88a, Sut89a]. Moreover, computers are usually not homogeneous: there are various architectures running different operating systems. In such a situation, when a specific distributed application is to be written, several problems arise:

- Communications between distributed programs,
- Access to remote files,
- Management and monitoring of the distributed executions.

A distributed application/system *GATOS* has been developed to solve the above problems. Its two main goals are:

- To automatically distribute parallel applications among heterogeneous hosts and to manage their executions,
- To provide a software layer in order to easily write new distributed applications in a heterogeneous environment.

First, we present the application model and the environment model. Then, we show how a multi-criteria load balancing can be used to choose appropriate hosts. Finally we give a survey of the GATOS server and the way users can manage distributed applications.

1. Applications

An application is a set of UNIX programs connected by a precedence graph. This graph defines a strict order for the executions of the programs, and gives qualitative information about the needed resources. Those resources are the files used and produced, and also the communications with other programs. This description is done both by a *static* graph: a file containing the graph description using the GATOS parallel application grammar, and by *dynamic* operations that allow to modify the execution graph while running.

1.1. Application and Environment

Beside the application model, GATOS manages the environment model that specifies the environment for the application executions. In heterogeneous systems, the surroundings are very important for allocations, in order to classify the different physical entities.

1.1.1. Execution Graph

The execution graph contains all the information to execute a parallel application. The strict order between programs is given by two relations. Let us consider two programs A and B:

- *Serialism*: B must wait for the end of A to be executed.
- *Parallelism*: A and B are executed in parallel.

The qualitative information also gives a strict order between programs:

- *Communication*: all communicating programs must be parallel,
- *Produced resources*: if not shared, all programs producing a resource must be serialised,
- *Used resources*: a resource cannot be used before being produced (resource synchronization).

In the example (Figure 1), the graph is made up of five programs connected by precedence arcs (bold lines). A and B are first parallelized, then C is executed after the end of A, D after both the end of A and B and finally E after the end of C. The qualitative information is: A uses f1 and produces f2 (grey lines), B produces f3, C uses f2, D uses f2 and f3 and produces f4, D and E communicate together (doted line).

The syntax used to specify an application is given in Program Listing 1. As an example the description of program D may be:

```
PROGRAM D
        AFTER A B USE f2 f3 PRODUCE f4
        COMMUNICATE E
ENDPROG
```

1.1.2. Environment

The graph execution involves one or more processor allocations. These allocations can only be made on the existing networks of hosts. To take into account heterogeneous system, all specifications about hosts and networks must be known. The main problems are:

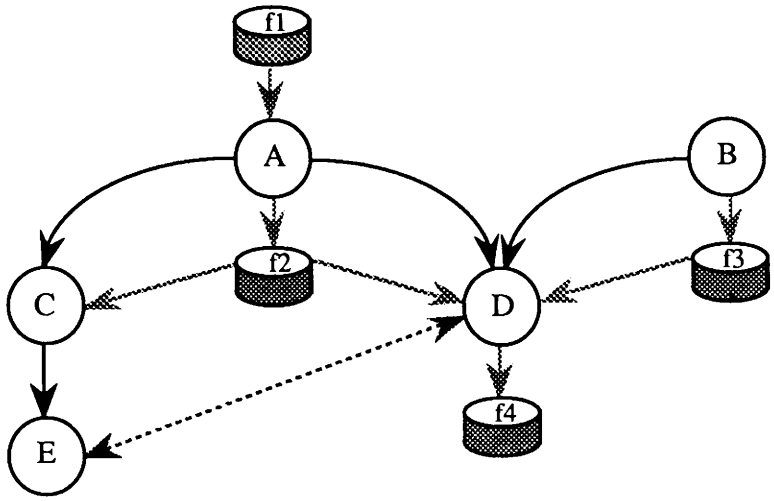


Figure 1: *Example of execution graph*

1. What are the sets of compatible hosts?
2. What is the network topology?
3. What are the different speeds, the memory capacities?

```
Application:
        [name]          # By default the file name containing the graph
        [ProgDef]+
ProgDef :               # Definition of a program
        [PROGRAM
                ChampsLoc
                [OPTIONS [i|I] [t|T] [s|S] [u|U]]
                                        # interactive, trace, statistic
                                        # use statistic
                [SYMBOLIC symbolicName]     # For user purpose
                [ARGUMENTS ArgList]         # Program arguments
                [ALLOCATION Allocations]    # Allocation algorithm
                [AFTER ProgramList]         # Precedences
                [COMMUNICATE ProgramList]   # Qualitatives informations
                [USE ResourceList]
                [PRODUCE ResourceList]
        ENDPROG]*

ProgramList :
        [programName [EXCLUSIVE] [RelativeLoc]]+
ResourceList :
        [[ChampsLoc]+ [EXCLUSIVE] [RelativeLoc]]+
ChampsLoc :
        OneLoc | LOCALIZATION [OneLoc]+ ENDLOC
OneLoc :
        [NET netName] [CLASS | TYPE | HOST name] resourceName
RelativeLoc :
        [SAME_HOST] | [SAME_TYPE] | [SAME_CLASS] | [SAME_NET]
        [DIFF_HOST] | [DIFF_TYPE] | [DIFF_CLASS] | [DIFF_NET]
Allocations:
        TIME &| LOAD &| MEMORY &| FILE &| COMMUNICATION
```

Program 1: *Graph Syntax*

The answers to questions one and two give the set of hosts that may participate in the load distribution. The answer to question three gives the necessary information for the program allocations.

1.1.2.1. Environment Description

The environment is made of three sets: a set of hosts, a set of bidirectional links connecting hosts and constituting a local area network (LAN) and a set of gateways connecting LANs.

A host is defined by:

- The *name*, unique in a LAN,
- The *processor type*, program executions can be made indifferently on any processor of the same type,
- The *class*: regrouping a set of compatible types,
- The *processor speed*: this is a subjective factor, but it permits to compare speeds of different hosts,
- *Physical information*: memory, disks, speed to access to resources.

A set of links constituting a LAN is defined by:

- A *type*: Ethernet, Token ring,
- A *speed*: according to the bandwidth of the link,
- A *name*.

A gateway is defined by:

- The *names* of the connected LANs.

The data needed by GATOS are quite bigger than what we can usually find in a distributed environment. For this reason, a special database containing one entry for each host must be created by the operators. Of course the configuration is static and must only be updated when there is any physical change in the system. This is the GATOS responsibility to manage automatically the dynamic part of the system configuration. This is done by an automatic recognition of start and stop (or crash) of hosts.

We may consider that a distributed application only works on a LAN. But sometimes it happens that vital computers involved in an application can be in other networks. To give to GATOS users a uniform access to all the available resources, we allow to use net names in their description.[†] Of course we do not consider this mechanism as "usual case" because its overhead is very important. Further discussion will consider an application as being executed in a local area network.

1.1.2.2. Application Impact

All components of a graph intrinsically have localization notions. Programs are located on the hosts where they are running, files are located on some disks in some hosts and communications between programs must be available anywhere on the networks (or at least in any LAN). Moreover programs can not be executed indifferently on any host due to the problem of heterogeneity. This problem may be solved by two different ways:

1. The system is able to build a program that can be executed on any host. This implies to give the sources, and for each different

[†] This mechanism is currently in development.

heterogeneous host to give (in a formal language) the necessary commands used to build the program [Bad88a].

2. The user builds himself the various version of the programs according to the various heterogeneous hosts and to his needs.

The second solution keeps us in the general way of distributed applications: if user wants a program to be executed indifferently on **n** kinds of incompatible computer, he has to build **n** different versions of his program. So, each entity of a program may have the followings *localization criteria*:

Host name, type name, class name, net name.

This allows programs and resources to be located in different hosts with different names, and enables GATOS to choose the correct program name or file name according to the allocated host characteristics. For example, the localization criterion for a program that needs to run on both SUN or VAX (belonging to the LAN) may be:

```
Program
      /users/example/code.vax on Class VAX or
      /users/example/code.sun on Class SUN
```

In the above example the different versions of a same program are supposed to be local. In fact the user can *prefix* the localization constraint with the name of the host containing the program, and if the program is not found on the allocated host, it is first migrated. In the same way, resources of a program (both files and communication programs) may contain one of the following *relative localization*:

Same host, same type, same class, same net,

Different host, different type, different class, different net.

This permits to specify constraints of resource localizations according to the program placement. For example, a file which must always be located on the program host, may be described by:

```
Program
      X
      File foo.file on Same_Host
```

GATOS may work with all the existing shared file systems. In practice, the user has some way to share files among hosts and has the same view of his files in any hosts where he can work (by example by using an NFS† partition mounted on all hosts). In such a case, all the existing programs may run transparently with GATOS.

All the information described above allows the users to benefit from any heterogeneous system. Practically, users can make as many versions of their programs as there are different host types. Of course, this is not necessary. If no localization criterion is given for a program, we assumes that it has to be executed only on the set of hosts of its original class (compatible hosts).

Relative Localization Compatibility

The relative localizations described above cannot be used without constraints. For a file, it makes no sense to specify a "different" localization as the local one (or the initial localization). For that case, we do not consider the "different" localization criteria for files. For the other resources the compatibility is shown in Figure 2.

† Network File System [Lyo86a].

SAME \ DIFF	NOTHING	HOST	TYPE	CLASS	NETWORK
NOTHING					
HOST		☒	☒	☒	☒
TYPE			☒	☒	
CLASS				☒	
NETWORK					☒

☒ forbidden
☐ coherent

Figure 2: *Localization constraints*

Coherence control results in one of the following situations:

a) The set of authorized hosts is empty: application execution is stopped and the user is informed.

b) The set of authorized hosts contains a single element: program placement is immediate (allocation totally constrained).

c) Several hosts are authorized to run the program: the placement algorithm finds out the best choice according to the specified criteria.

1.2. Distributed Operations

The graph description presented above allows to build *static* distributed applications. The precedence orders are strict, and there is no explicit conditional execution. Furthermore, we have not yet explained how a program may access transparently a remote file, and how two programs may communicate together with no localization dependency.

To solve the above problems, we supply to the programmers a library of operations:

- **Graph operations**: dynamic creation of new tasks, suppression of running or waiting tasks, stop/abort of the running application,
- **Resources operations**: file accesses, communication between programs,
- **Special operations**: files migration, remote processes management, launch of new applications.

1.3. Multi-Criteria Allocation

An application may be fully constrained, which means that a user may specify a unique host for each program. Most of the time, the host loads are quite different: some may be overloaded while some others are unused. To take benefit of the idle hosts of the system, GATOS has an allocator that can choose an optimal placement from a list of available hosts [Abr82a, Shi85a, Zah86a]. The set of informations specified in the models (localization and execution graph) is taken into account by the program allocator [Bou90a]. Application programs need

resources (CPU, memory, file, communication) which vary from a program to another. In the application description, the user can specify appropriate policies for each program. The specific *allocation criteria* are one or several of the following:

- *Response time*: optimize the response time,
- *Memory used*: optimize both response time and host memory available,
- *File access*: optimize the program placement according to the file localizations; file migration is possible,†
- *Communications*: optimize the communication cost between programs.

Such criteria already give a good information about the optimal placement to be done according to the use of programs. The user may specify either one allocation criterion or a list of criteria sorted by significance order.

A important problem for those algorithms is the amount of information about the various programs in order to make the "best" placement:

- *For response time and memory placement*: the processor time and the memory needed,
- *For file access*: for each file, the number of accesses,
- *For communications*: for each communicating program, the number of packets.

Two solutions may be used to get this information:

1. To let users provide themselves all the figures.
2. To keep tracks of the precedent executions.

The last solution is generally used by GATOS. When a new program is run for the first time, the placement is not optimal (as constants are used instead of real data) but the placement becomes better in accordance of the updated figures. When the user has a very good knowledge of his programs, he can use the first solution and enter himself the data that will be used by the allocator.

2. GATOS

GATOS has been built above the UNIX operating system, and works in a set of SUN 3x, SUN 4x and SPARC workstations connected by a LAN. It consists in servers and a set of commands and functions to communicate with servers. The whole constitutes the GATOS system. A GATOS server is running on every host of the system that may participate in the load distribution. They cooperate together by message exchanges, to execute user defined applications. Each server keeps a list of all the other active servers. This ensures the system dynamic configuration (stop/restart or crash of hosts).

2.1. Exchange of Informations

A critical problem for the load distribution is the amount of data exchanged between the hosts belonging to a LAN in order to allocate the programs [Lan88a, Mut88a].

† If a file is migrated for the needs of the allocation, it is then migrated again to its original place.

Two main methods are used:

- *Asking*: a server who needs to allocate some programs asks for information to all the other servers.
- *Periodically*: all servers periodically exchange their information.

The second method is used in GATOS to update the informations between servers. The informations needed by GATOS are: the *host load*, the *available memory* and the *number of active processes*. We consider the load as the average number of both active processes and processes waiting less than a limit.

When a server needs to allocate some programs, it has **already** an approximation of the load and of the available memory of all the hosts. A good approximation is given by the time between two load sending: the shorter it is, the better the approximation but the more overcrowded by messages the network is. In order to optimize the load exchange between servers, a second mechanism is integrated: the *piggy-backing*. When a server communicates with an other server, the load is automatically incorporated to the message. The overhead of this mechanism is quite null, and thus it allows to decrease the time delay between load update.

A threshold mechanism is also incorporated to limit the exchange of messages. When one host is overloaded it is no use to continue to broadcast the load: this host will never be chosen as an allocated processor. So, when a host load is above threshold, it just informs others servers that it is overloaded and stop transmitting its load until load decreases. In fact there are two thresholds: the **overloaded threshold** and the **limit threshold.** The first one indicates when a host cannot be used. It will become usable again when the load decreases under the limit threshold. The difference between the two thresholds gives some tolerance for the fast load variation.

2.2. Execution of a Distributed Application

For a single program there are two ways to manage its distribution:

1. Totally transparent: users are not concerned with the distribution.
2. On user request: users specify when they want their programs to be distributed.

As GATOS offers much more than single program distribution, users must be involved by the specification of a distributed application. This does not exclude the first solution. A software layer (such as a command interpreter) may be added to transparently distribute the more common "big" programs. This is not as simple as it seems: which programs are *benefic* to distribute? what about the crash of a remote host executing a program for an unconcerned user?

All the operating of an application can be defined by users. The static part of an application is given in a file containing the execution graph. The user initializes the application by means of the **gato command.** This command invokes a GATOS server (usually the local one). This server becomes the *master server* for that application. It is the only server that decides for the application programs allocation. This mechanism appears as a gateway between the user and GATOS [Sha86a]. The command creates a special task that will be the link between the user console and the executions of the different programs (Figure 3). So the user can be informed of the evolution of its application and the various hosts chosen by the allocator, get back the output made from its programs (normal or error messages) and can interac-

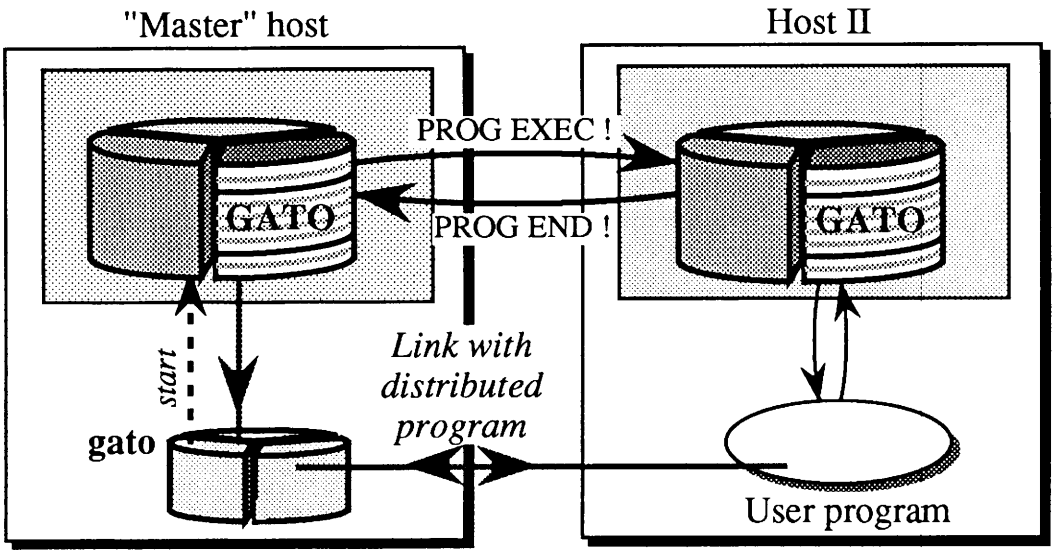


Figure 3: *Execution mechanisms*

tivly write to remote programs (for instance, it is possible to launch a text editor like emacs).

The main options available for this command are:

g <file>	:	file containing the static graph
i | I	:	interactive application
t | T	:	trace of application
s | S	:	update the statistic
u | U	:	use of the statistic by the program allocator
r <allocation>	:	allocation names
m <host>	:	master host
d | D	:	dynamic choice of the master host

In Figure 3, the master server sends a request to an other server (PROG EXEC!). This one executes the program (possibly after migration if a compatible code version is not available locally). When the execution is finished it informs the master (PROG END!). The control process for a given user is directly in connection (if the interactive option is specified) with the user window. There is absolutely no difference between a local and a remote execution.

2.3. Managing the Executions

Once the application has been launched, each program of the application may use special GATOS functions as described in I.2.

Users manage the execution of distributed applications by using a set of special commands. We have already described the main command: *gato*. Other are available:

- List of running applications/programs,
- Destruction of owning application/programs,
- Information on the various hosts of the system.

Conclusion

The GATOS distributed task manager improves the utilization of the various network hosts of a heterogeneous system. This project has been developed under the UNIX BSD 4.1 operating system. It runs on a LAN composed of SUN3x, SUN4x and SPARC workstations.

All existing programs can be transparently executed on the less loaded hosts. Moreover, GATOS provides to programmers powerful functions in order to easily build complex distributed applications in heterogeneous environment.

GATOS collects a lot of informations about the system state and the execution needs of the applications, thus making them available to the allocation algorithms. Furthermore, GATOS allows to manage distributed applications in heterogeneous environment (transparent program execution, access libraries, etc.). All applications can benefit from the multi-criteria dynamic placement, according to the wishes of the users.

References

[Abr82a] Timothy C.K. Chou, Jacob A. Abraham, "Load Balancing in Distributed Systems," pp. 401-412 in *IEEE Transactions on Software Engineering* (July 1982).

[Bad88a] Samir Al Badine, "STMM: Soumission de Travaux en Mode Messagerie," in *Thesis, University of Paris VI* (June 1988).

[Bou90a] Raouf Boutaba, "Placement Optimal de Taches sur un Réseau Hétérogène de Stations de Travail," in *Rapport de D.E.A. du laboratoire MASI* (September 1990).

[Lan88a] Marvin M. Theimer, Keith A. Lantz, "Finding Idle Machines in a Workstation-based Distributed System," in *8th International Conference on Distributed System*, San José, California (June 1988).

[Lyo86a] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, Bob Lyon, "Design and Implementation of the Sun Network Filesystem," in *Sun Microsystems, Technical Report* (1986).

[Mut88a] Michael J. Litzkow, Miron Livny, Matt W. Mutka, "Condor – A Hunter of Idle Workstations," in *8th International Conference on Distributed Computing Systems*, San José, California (January 1988).

[Sha86a] Marc Shapiro, "SOS: a Distributed Obect-Oriented Operating System," in *2nd European SIGOPS Workshop, on "Making Distributed Systems Work"*, Amsterdam (September 1986).

[Shi85a] Amnon Barak, Amnon Shiloh, "A Distributed Load-balancing Policy for a Multicomputer," pp. 901-913 in *Software-Practice and Experience* (September 1985).

[Sut89a] Alain Billionnet, Marie-Christine Costa, Alain Sutter, "Problèmes de Placement dans les Systèmes Distribués," in *Technique et Sciences Informatiques* (1989).

[Zah86a] Derek L. Eager, Edward D. Lazoska, John Zahorjan, "Adaptative Load Sharing in Homogeneous Distributed

Systems," pp. 662-675 in *IEEE Transactions on Software Engineering* (May 1986).

Process Sleep and Wakeup on a Shared-memory Multiprocessor

Rob Pike Dave Presotto
Ken Thompson Gerard Holzmann

Bell Labs, New Jersey, USA
{rob | presotto | ken | gerard}@research.att.com

Abstract

The problem of enabling a "sleeping" process on a shared-memory multiprocessor is a difficult one, especially if the process is to be awakened by an interrupt-time event. We present here the code for sleep and wakeup primitives that we use in our multiprocessor system. The code has been exercised by months of active use and by a verification system.

The Problem

Our problem is to synchronise processes on a symmetric shared-memory multiprocessor. Processes suspend execution, or *sleep*, while awaiting an enabling event such as an I/O interrupt. When the event occurs, the process is issued a *wakeup* to resume its execution. During these events, other processes may be running and other interrupts occurring on other processors.

More specifically, we wish to implement subroutines called `sleep`, callable by a process to relinquish control of its current processor, and `wakeup`, callable by another process or an interrupt to resume the execution of a suspended process. The calling conventions of these subroutines will remain unspecified for the moment.

We assume the processors have an atomic test-and-set or equivalent operation but no other synchronisation method. Also, we assume interrupts can occur on any processor at any time, except on a processor that has locally inhibited them.

The problem is the generalisation to a multiprocessor of a familiar and well-understood uniprocessor problem. It may be reduced to a uniprocessor problem by using a global test-and-set to serialise the sleeps and wakeups, which is equivalent to synchronising through a monitor. For performance and cleanliness, however, we prefer to allow the interrupt handling and process control to be multiprocessed.

Our attempts to solve the sleep/wakeup problem in Plan 9 [Pik90a] prompted this paper. We implemented solutions several times over several months and each time convinced ourselves – wrongly – they were correct. Multiprocessor algorithms can be difficult to prove correct by inspection and formal reasoning about them is impractical.

We finally developed an algorithm we trust by verifying our code using an empirical testing tool. We present that code here, along with some comments about the process by which it was designed.

History

Since processes in Plan 9 and the UNIX system have similar structure and properties, one might ask if UNIX `sleep` and `wakeup` [Bac86a] could not easily be adapted from their standard uniprocessor implementation to our multiprocessor needs. The short answer is, no.

The UNIX routines take as argument a single global address that serves as a unique identifier to connect the wakeup with the appropriate process or processes. This has several inherent disadvantages. From the point of view of `sleep` and `wakeup`, it is difficult to associate a data structure with an arbitrary address; the routines are unable to maintain a state variable recording the status of the event and processes. (The reverse is of course easy – we could require the address to point to a special data structure – but we are investigating UNIX `sleep` and `wakeup`, not the code that calls them.) Also, multiple processes sleep "on" a given address, so `wakeup` must enable them all, and let process scheduling determine which process actually benefits from the event. This is inefficient; a queueing mechanism would be preferable but, again, it is difficult to associate a queue with a general address. Moreover, the lack of state means that `sleep` and `wakeup` cannot know what the corresponding process (or interrupt) is doing; `sleep` and `wakeup` must be executed atomically. On a uniprocessor it suffices to disable interrupts during their execution. On a multiprocessor, however, most processors can inhibit interrupts only on the current processor, so while a process is executing `sleep` the desired interrupt can come and go on another processor. If the wakeup is to be issued by another process, the problem is even harder. Some inter-process mutual exclusion mechanism must be used, which, yet again, is difficult to do without a way to communicate state.

In summary, to be useful on a multiprocessor, UNIX `sleep` and `wakeup` must either be made to run atomically on a single processor (such as by using a monitor) or they need a richer model for their communication.

The Design

Consider the case of an interrupt waking up a sleeping process. (The other case, a process awakening a second process, is easier because atomicity can be achieved using an interlock.) The sleeping process is waiting for some event to occur, which may be modeled by a condition coming true. The condition could be just that the event has happened, or something more subtle such as a queue draining below some low-water mark. We represent the condition by a function of one argument of type `void*`; the code supporting the device generating the interrupts provides such a function to be used by `sleep` and `wakeup` to synchronise. The function returns `false` if the event has not occurred, and `true` some time after the event has occurred. The `sleep` and `wakeup` routines must, of course, work correctly if the event occurs while the process is executing `sleep`.

We assume that a particular call to `sleep` corresponds to a particular call to `wakeup`, that is, at most one process is asleep waiting for a par-

ticular event. This can be guaranteed in the code that calls `sleep` and `wakeup` by appropriate interlocks. We also assume for the moment that there will be only one interrupt and that it may occur at any time, even before `sleep` has been called.

For performance, we desire that multiple instances of `sleep` and `wakeup` may be running simultaneously on our multiprocessor. For example, a process calling `sleep` to await a character from an input channel need not wait for another process to finish executing `sleep` to await a disk block. At a finer level, we would like a process reading from one input channel to be able to execute `sleep` in parallel with a process reading from another input channel. A standard approach to synchronisation is to interlock the channel "driver" so that only one process may be executing in the channel code at once. This method is clearly inadequate for our purposes; we need fine-grained synchronisation, and in particular to apply interlocks at the level of individual channels rather than at the level of the channel driver.

Our approach is to use an object called a *rendezvous*, which is a data structure through which `sleep` and `wakeup` synchronise. (The similarly named construct in Ada is a control structure; ours is an unrelated data structure.) A rendezvous is allocated for each active source of events: one for each I/O channel, one for each end of a pipe, and so on. The rendezvous serves as an interlockable structure in which to record the state of the sleeping process, so that `sleep` and `wakeup` can communicate if the event happens before or while `sleep` is executing.

Our design for `sleep` is therefore a function

```
void sleep(Rendezvous *r, int (*condition)(void*), void *arg)
```

called by the sleeping process. The argument `r` connects the call to `sleep` with the call to `wakeup`, and is part of the data structure for the (say) device. The function `condition` is described above; called with argument `arg`, it is used by `sleep` to decide whether the event has occurred. `Wakeup` has a simpler specification:

```
void wakeup(Rendezvous *r)
```

`Wakeup` must be called after the condition has become true.

An Implementation

The `Rendezvous` data type is defined as

```
typedef struct{
        Lock    l;
        Proc    *p;
} Rendezvous;
```

Our `Locks` are test-and-set spin locks. The routine `lock(Lock *l)` returns when the current process holds that lock; `unlock(Lock *l)` releases the lock.

Figure 1 is our implementation of `sleep`. Its details are discussed below. `Thisp` is a pointer to the current process on the current processor. (Its value differs on each processor.) Figure 2 is `wakeup`. `Sleep` and `wakeup` both begin by disabling interrupts and then locking the rendezvous structure. Because `wakeup` may be called in an interrupt routine, the lock must be set only with interrupts disabled on the current processor, so that if the interrupt comes during `sleep` it will occur only on a different processor; if it occurred on the processor executing `sleep`, the spin lock in `wakeup` would hang forever. At

```
void
sleep(Rendezvous *r, int (*condition)(void*), void *arg)
{
        int s = inhibit();      /* interrupts */
        lock(&r->l);

        /*
         * if condition happened, never mind
         */
        if((*condition)(arg)){
                unlock(&r->l);
                allow(s);       /* interrupts */
                return;
        }

        /*
         * now we are committed to
         * change state and call scheduler
         */
        if(r->p)
                error("double sleep %d %d", r->p->pid, thisp->pid);
        thisp->state = Wakeme;
        r->p = thisp;
        unlock(&r->l);
        allow(s);       /* interrupts */
        sched();        /* relinquish CPU */
}
```

Figure 1: *sleep*

```
void
wakeup(Rendezvous *r)
{
        Proc *p;
        int s;

        s = inhibit();  /* interrupts; return old state */
        lock(&r->l);
        p = r->p;
        if(p){
                r->p = 0;
                if(p->state != Wakeme)
                        panic("wakeup: not Wakeme");
                ready(p);
        }
        unlock(&r->l);
        if(s)
                allow();
}
```

Figure 2: *wakeup*

the end of each routine, the lock is released and processor priority returned to its previous value. (`Wakeup` needs to inhibit interrupts in case it is being called by a process; it is a no-op if called by an interrupt.)

`Sleep` checks to see if the condition has become true, and returns if so. Otherwise the process posts its name in the rendezvous structure where `wakeup` may find it, marks its state as waiting to be awakened (this is for error checking only) and goes to sleep by calling `sched()`. The manipulation of the rendezvous structure is all done under the lock, and `wakeup` only examines it under lock, so atomicity and mutual exclusion are guaranteed.

`Wakeup` has a simpler job. When it is called, the condition has implicitly become true, so it locks the rendezvous, sees if a process is waiting, and readies it to run.

Discussion

The synchronisation technique used here is similar to known methods, even as far back as Saltzer's thesis [Sal66a]. The code looks trivially correct in retrospect: all access to data structures is done under lock, and there is no place that things may get out of order. Nonetheless, it took us several iterations to arrive at the above implementation, because the things that *can* go wrong are often hard to see. We had four earlier implementations that were examined at great length and only found faulty when a new, different style of device or activity was added to the system.

Here, for example, is an incorrect implementation of wakeup, closely related to one of our versions.

```
void
wakeup(Rendezvous *r)
{
        Proc *p;
        int s;

        p = r->p;
        if(p){
                s = inhibit();
                lock(&r->l);
                r->p = 0;
                if(p->state != Wakeme)
                        panic("wakeup: not Wakeme");
                ready(p);
                unlock(&r->l);
                if(s)
                        allow();
        }
}
```

The mistake is that the reading of `r->p` may occur just as the other process calls `sleep`, so when the interrupt examines the structure it sees no one to wake up, and the sleeping process misses its wakeup. We wrote the code this way because we reasoned that the fetch `p = r->p` was inherently atomic and need not be interlocked. The bug was found by examination when a new, very fast device was added to the system and sleeps and interrupts were closely overlapped. However, it was in the system for a couple of months without causing an error.

How many errors lurk in our supposedly correct implementation above? We would like a way to guarantee correctness; formal proofs are beyond our abilities when the subtleties of interrupts and multiprocessors are involved. With that in mind, the first three authors approached the last to see if his automated tool for checking protocols [Hol91a] could be used to verify our new `sleep` and `wakeup` for correctness. The code was translated into the language for that system (with, unfortunately, no way of proving that the translation is itself correct) and validated by exhaustive simulation.

The validator found a bug. Under our assumption that there is only one interrupt, the bug cannot occur, but in the more general case of multiple interrupts synchronising through the same condition function and ren-

dezvous, the process and interrupt can enter a peculiar state. A process may return from `sleep` with the condition function false if there is a delay between the condition coming true and `wakeup` being called, with the delay occurring just as the receiving process calls `sleep`. The condition is now true, so that process returns immediately, does whatever is appropriate, and then (say) decides to call `sleep` again. This time the condition is false, so it goes to sleep. The wakeup process then finds a sleeping process, and wakes it up, but the condition is now false.

There is an easy (and verified) solution: at the end of `sleep` or after `sleep` returns, if the condition is false, execute `sleep` again. This re-execution cannot repeat; the second synchronisation is guaranteed to function under the external conditions we are supposing.

Even though the original code is completely protected by interlocks and had been examined carefully by all of us and believed correct, it still had problems. It seems to us that some exhaustive automated analysis is required of multiprocessor algorithms to guarantee their safety. Our experience has confirmed that it is almost impossible to guarantee by inspection or simple testing the correctness of a multiprocessor algorithm. Testing can demonstrate the presence of bugs but not their absence [Dij72a].

We close by claiming that the code above with the suggested modification passes all tests we have for correctness under the assumptions used in the validation. We would not, however, go so far as to claim that it is universally correct.

References

[Bac86a] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs (1986).

[Dij72a] Edsger W. Dijkstra, "The Humble Programmer – 1972 Turing Award Lecture," *Comm. ACM* **15**(10), pp. 859-866 (October 1972).

[Hol91a] Gerard J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs (1991).

[Pik90a] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, "Plan 9 from Bell Labs," pp. 1-9 in *Proceedings of the Summer 1990 UKUUG Conference*, London (July, 1990).

[Sal66a] Jerome H. Saltzer, *Traffic Control in a Multiplexed Computer System*, MIT, Cambridge, Mass. (1966).

Capturing the Behaviour of Distributed Systems

Terje Fallmyr

Nordland College, Norway
terje@ioa.hsn.no

David Holden

Harwell Laboratory, England
holden@harwell.uucp

Otto J. Anshus

University of Tromsø, Norway
otto@cs.uit.no

Abstract

The ability to manage distributed computing systems depend on knowledge of its behaviour. Monitoring can be used to obtain information about whole systems or parts of systems. Such pieces of information relate to different *abstraction levels* of the system. A piece of information about a part of a system we term a *behaviour indicator*. This report describes behaviour indicators in relation to the monitoring model developed in the MANDIS† project. In a worked example the mapping from high level behaviour indicators through *object set states* to *basic states and events* is shown. The mapping is illustrated by examples of a language, **SESL**, for specification of events and state changes.

1. Introduction

Monitoring is an important activity in distributed systems, and may be used to get knowledge about the behaviour of a system at various levels; either whole systems, parts of systems, or patterns of interconnection between the parts. Monitoring may be used for management purposes, fault detection and diagnosis, performance measurement, support in experimental distributed systems, long- and short-term planning, etc.

The following aspects of distributed computing systems are emphasised as they form the basic reasoning behind some of the argu-

† This work has been carried out under the COST 11 ter MANDIS project and the TRACE project.

ments in this paper. We regard distributed computing systems to consist of asynchronous parallel processes communicating over non-zero delay links. There is no central point of control, accurate global state or global time. The active entities in a distributed system (e.g. "processes" or "objects") may change their communication patterns according to system changes. These points illustrate the fact that distributed systems are hard to monitor.

In the design of a monitoring system certain goals should be met. Firstly, the monitoring system must fulfill its primary role of getting the information requested by the user from the monitored system to the user. The monitoring information should be presented at the correct abstraction level, that is, a level suited to the actual user or application. The monitoring system should be flexible (or adaptable) and provide the user with the ability to view the monitored system from the preferred point of view. This should give the user the potential to change dynamically the point(s) of interest in the monitored system. The user should be allowed to choose between several temporal aspects of monitoring, that is, whether monitoring information should be presented in real-time or stored for a subsequent (post-mortem) analysis. The monitoring system should not disturb and thereby change the timing aspects of the monitored system in such a way that the monitoring information is only valid when monitoring is done. The monitoring system should preserve the sequence of detected events in the monitored system, and in certain cases also with respect to a hypothetical global real-time clock, specified within some finite delay. Due to the potentially large amounts of data that may be detected, the monitoring system should filter and structure the information flow such that the user only gets the requested information. Finally, good design rules concerning the visualisation of the monitoring information should be used.

The goals listed above place great demands on monitoring systems, and they may be in conflict with one another. In order to try to meet the demands, this paper presents a general approach to monitoring, which may ease the design of monitoring systems for many different purposes.

In the following chapters we will outline our approach to modelling and designing monitoring systems. We will describe a language (SESL) for analysis of monitoring information through the specification of events and state changes. We will use the approach and the language to illustrate a worked example of capturing the behaviour of a distributed service under different load sharing algorithms.

Much work has been done on the subject of monitoring in the past years, especially in the context of debugging distributed systems [Bat83a, Hab90a, Gar84a], but also "pure" monitoring works [Das86a, Ogl88a, Joy87a] have been and are still being carried out. Many of the projects have also succeeded in implementing monitoring systems suited for their applications.

2. Approach

In order to support the somewhat different objectives of the COST 11 ter MANDIS project [Lan88a] and the design of a test bed for distributed operating systems [Ans88a, Fal88a], our work on monitoring aimed to use an approach in which monitoring could initially be described in terms independent of any particular design of monitoring service or system architecture, see also [Hol88a].

Bearing in mind the points listed above on the many goals for the design of monitoring systems as well, there was a need to give general descriptions of the structure of the monitoring information handled by monitoring systems regardless of any actual systems. In order to do this, we chose a top-down approach where we used the concept of the *behaviour* of distributed systems.

Monitoring may be described by two aspects: the structure of the monitoring information, and the monitoring system which handles it. Our approach attempts to keep these two aspects separated.

The approach consists of the identification and outline of four steps. The first step, the **general framework**, outlines an abstract three level model for monitoring information. The model is general, and makes very few assumptions about the monitored system, or how the monitoring information is obtained. The second step defines a **specific framework**, which describes the structure of monitoring information tailored to a monitored system with a specific system architecture. The third step, the **functional model**, is the design of the monitoring system itself. The final step is to describe and build an implementation of the design.

Our motivation for using this top-down approach has been to let the users of the monitoring system and the monitoring information have as much flexibility as possible. The users should have the facility to describe, related to their particular needs, what information is currently of interest. The monitoring system should not merely provide certain fixed standard options; it should provide only the requested monitoring information and be adaptable to changing user needs and system architectures.

Consequently there is a need for filtering "garbage" from a stream of monitoring information, giving the user only the items of information specified, thus ensuring they are "of interest".

2.1. General Framework

The general framework outlines a model for monitoring independent of any particular monitoring system or monitored system. The only assumption about the monitored system is that it is composed of "objects" (in some sense). The general framework regards objects as an elementary and independent unit of structuring. They are independent in the sense that they may be characterised without reference to the context within which they have been placed. Furthermore, objects are the lowest level of granularity, in terms of identifiable (nameable) entities active within the system. Objects have a state, and this state can be observed or communicated in some way (actively (by the object) or passively). Objects are defined recursively, so an object may be composed of other objects, leading to composite objects.

We have divided the general framework into three levels of abstraction. It describes the relations between *monitoring information* on these three levels. The general framework does not describe any monitoring systems, but simply assumes that they support the basic functions of extracting and processing monitoring information.

The highest abstraction level of monitoring information is the **behaviour indicator** (*BI*) [Ans88b]. Behaviour Indicators are specified by the user in the user's context, and collectively give the user's perception of that part of the system in which the user is currently interested. The values of BIs are derived from the underlying monitoring system, and present the *observed behaviour* of (a part of) the moni-

tored system. The values of the *BI*s are presented in a manner specified by the user of the monitoring system.

The second level of abstraction defines **object states** (*OS*) and **object set states** (*OSS*). *OS*s denote the state of a single object, while the *OSS*s denote the combined state of an arbitrary (possibly user-defined) collection of objects. The specifications of the *BI*s should naturally decompose into lower level specifications involving specific single objects or sets of objects. The value of the *OS* and the *OSS* will together give the value of the *BI*s (from which they were determined).

The values of the object (set) states are determined from **basic monitoring data** (*BMD*), which constitute the lowest level of monitoring information. *BMD* are detected directly in the monitored objects. It is only at the lowest level, where the basic monitoring data is captured, that the monitoring system is explicitly intrusive and is in direct contact with the monitored system. As a model, the general framework does not specify how *BMD* are derived.

2.2. The Specific Framework

The specific framework is a development of the general framework and describes the structure of monitoring information tailored to monitored systems with a specific system architecture. The monitored system, and in particular the aspects that have an impact on the monitoring information, is described. In the specific framework concepts such as processes, objects (with certain specific capabilities), clients, servers, interaction paradigm, protection, etc. will be defined, and should be used in the descriptions.

The three levels of abstraction from the general framework are used, and the levels may correspond to particular features of the architecture of the monitored system.

In the example in chapter 4, which will be on the specific framework level, the mapping of *BI*s to *OS*s and *OSS*s are shown.

2.3. The Functional Model

The functional model is a specification of the structure of a particular monitoring system. It is based on the description of the structure of monitoring information in the specific framework, and describes how a monitoring system can be divided into a set of functions which accomplish the tasks required to gather the necessary monitoring information. We propose a generic structure of these specifications using three layers as shown in Figure 1.

2.3.1. Analysis Layer

The analysis layer decomposes specifications of *BI*s into specifications of object (set) states, and tells the underlying combination layer what object (set) states are of current interest. When relevant monitoring information is received from the combination layer, the analysis layer derives the values of the BIs. The computations will range from simple statistics gathering (and calculation of more complex statistical indicators), to the complex task of fault diagnosis.

2.3.2. Combination Layer

The combination layer receives specifications from the analysis layer of which object (set) states (*OS*s and *OSS*s) are of interest, and instructs

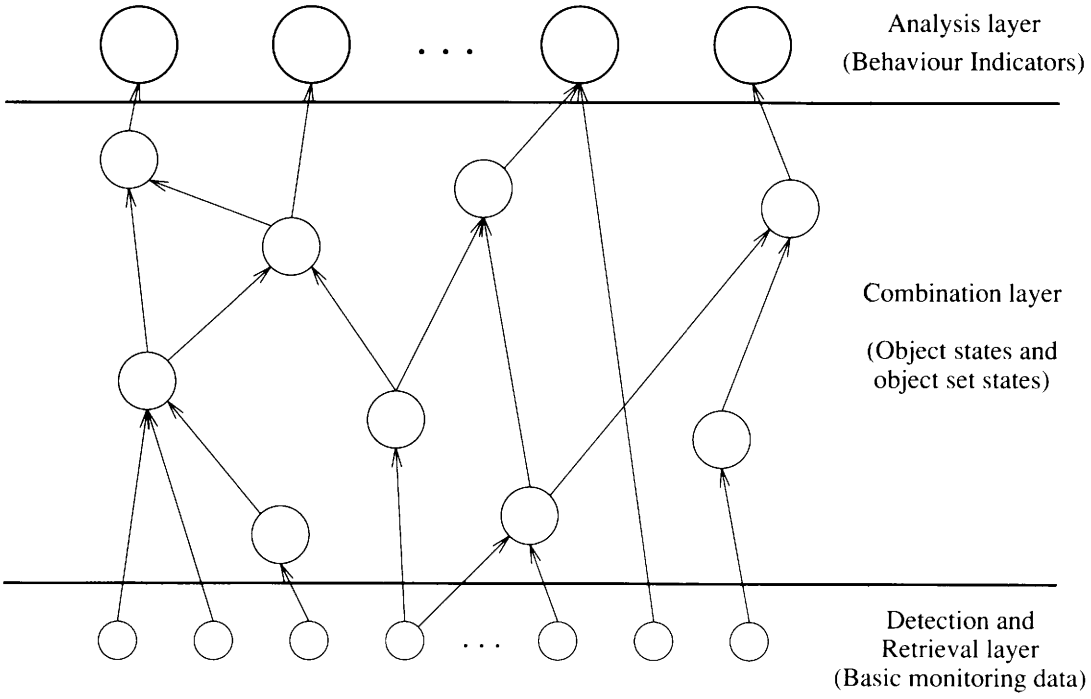


Figure 1: *Functional Model*

the underlying detection and retrieval layer about *BMD* of interest. It sets up the monitoring activity accordingly. The combination layer collects and collates the information generated by the detection and retrieval layer; it performs grouping of events and states guided by the specifications from the analysis layer. Thereby it derives the values of the *OS*s and *OSS*s from *BMD*s, and passes the relevant values up to the analysis layer.

2.3.3. Detection and Retrieval Layer

The detection and retrieval layer receives specifications from the combination layer of what *BMD* (basic events and states), should be detected from the monitored system. In this layer the actual production of monitoring information is done. This can be performed in many ways, for example, by dedicated hardware that captures data on internal busses; or pieces of code inserted into the applications or the kernel, which monitor the activity from the inside; or by any other means that can be devised. The information gathered by this layer consists of event reports, indicating the occurrence of some identifiable event, and state information for various specified states within the monitored system. *BMD* will be passed to the combination layer for further processing.

2.4. The Implementation

Implementations have been carried out from designs based on the functional model e.g. [Dre88a] and [Sae91a]. The implementation built at Harwell [Hol91a] was designed around a central (but possibly distributed) *Data Combiner*. The Combiner gathers data from the processes being monitored, analyses the data, and send the results to the management applications which requested them (see Figure 1). Data can be exchanged between separate combiner processes to enable the analysis to be decomposed into smaller units. This reduces the volume of data

transmitted on a network, and allows different management domains to restrict access to low-level data. The Combiner forms the combination layer of the functional model, and can also perform the functions of the analysis layer (the management application may or may not perform further analysis).

Monitoring access to monitored processes is provided through the *Manager and Detector* processes (the detection and retrieval layer). These are created by the monitored process and handle the control of the monitoring activity, implementation of management commands, extraction of data and external communication with the combiner or management applications. The monitored process generates events and reads and writes monitored and managed states through calls to routines in the *Monitoring Library*. Event indications and state values are held in memory shared by the monitored process and the manager and detector processes, providing a highly efficient form of local communications. The manager process waits for requests from the combiner process or management applications which cause it to report state values, issue user-defined management commands, change state values, enable event reporting, or terminate the monitoring activity. The detector process waits for event indications to be stored by the monitored process and sends notifications to the combiner.

Management applications initiate the monitoring activity, send the analysis rules to the combiner (or perform it themselves), and request state reports or wait for event reports.

Communications between the modules is through efficient LAN and WAN protocols (Amoeba) [Ren87a]. Addressing is handled using global location-independent names stored in the Amoeba Directory Service.

The implementation is described in more detail in [Hol91a].

3. State and Event Specification Language (SESL)

3.1. Rule-Based Data Combination and Analysis

The **State and Event Specification Language** (SESL) was devised in order to provide a simple mechanism for specifying the operations to be performed in order to analyse the events and states existing in a distributed system. It has been designed to perform most or all of the computations needed in the Combination and Analysis layers. A language provides a simple, stable interface with well-defined syntax suitable for the occasional user, but also customisable for the regular user. SESL allows the declaration and definition of states and events in terms of previously defined events and states in an extensible manner, providing a multi-level specification scheme which thus implements the abstraction level technique for decomposing distributed systems described above.

SESL is described in more detail in [Hol91b].

SESL's extensibility derives from its basic statements:

```
HighLevelEvent when EventString [provided State]
HighLevelState := StateString
```

An *EventString* is a specification of a pattern of events, either high-level events (referenced by name), or low-level events (referenced by

EVENT("LowLevelEvent")). These patterns are specified using event operators:

A → B matches when event B occurs (any time) after event A;

A ⇒ B matches when B immediately follows A;

A | B matches when either A or B occur;

A → !B → C matches when C occurs after A without B occurring in between.

A *HighLevelEvent* is defined to occur whenever the pattern defined by EventString is matched. If a `provided` clause is specified, the event only occurs when the EventString is matched *and* the state is true (non-zero).

A *StateString* is an arithmetic expression which is used to calculate a state, defined in terms of other states and state operators. The following states and operators may be used:

`$State` – a high-level state;

`state("LowLevelState")` – a low-level state;

`#Event` – the number of times a particular (high-level) event has occurred;

`@Event` – the time at which a particular event occurred;

`+ - * / % == != < > <= >= and or` – normal arithmetic operators.

A *change of state* is also an event; e.g. `Event WHEN $State == 4` occurs at the instant `State` becomes equal to 4.

Events and states may be defined in a global context, or a local context relating only to the object being monitored. This allows identical local specifications to be used when monitoring several identical objects without the names of events and states interfering with their counterparts in different local contexts. In order for events and states in the global context to use local events and states, the *combination* operators are used. For events:

`GlobalEvent1 WHEN ANY LocalEvent1` triggers the global event when any local event with the specified name occurs.

For states:

`(+)$LocalState` is the sum of all local states of that name;

`(*)$LocalState` is the product of all local states of that name;

`(#)$LocalState` is the number of local states with that name;

`(<)$LocalState` is the smallest local state with that name;

`(>)$LocalState` is the largest local state with that name.

Events can be used to trigger activities in the monitoring system. Using the "AWAIT event" command, the occurrence of an event will cause a notification to be sent to the user's terminal or application. Alternatively the more powerful "ON event procedure" command can be used to define a procedure to be executed when the event occurs. This may be a series of statements analysing states using the

```
set $stateName StateString
```

and

```
if StateString procedure
```

statements, or it may enable or disable event reporting, or simulate the triggering of other events. Procedural statements may be grouped using the `do ... end` construct. For this experiment two new features have been added to the language: state arrays (declared using `declare $arrayName[size]`); and while loops (`while StateString procedure`).

Other actions, such as controlling a piece of hardware or software as part of a distributed systems management activity lie outside the scope of pure monitoring. Such actions are performed by user software, triggered by specific events generated by the analysis layer.

4. Example: Quality of Load Sharing

The problem of load sharing in distributed systems is of concern in several areas, for instance in a global scheduling strategy. A strategy for load sharing should make multiple processing units in a system cooperate to share the system workload. The strategy should share jobs fairly among servers regardless of origin. It should enable concurrency between jobs, avoid bottlenecks in the system, and maintain the robustness and performance in the presence of partial failures in the system.

In this example the system is considered to be a set of nodes connected by a communication channel. All nodes can communicate directly with all the others, and the communication delay is the same between all nodes. Jobs may enter the system through any node, but they will leave the system through the same node. They arrive at the source node from outside the system and depart from the system after processing. Jobs are defined as entities that can be processed. When a job enters the system it will be queued somewhere until it is scheduled for processing at some node (server) according to some global scheduling algorithm, called algorithm *A* throughout the example. Once the job is scheduled, it will be processed at the same node until it exits and departs the system.

In this paper we present a derivation of a measure called the quality of load sharing, or Q-factor [Wan85a]. This high level indicator of system behaviour is defined below as a comparison between the performance of algorithm *A* in servicing jobs, measured by the mean *response time* for those jobs, and the performance of a standard queueing algorithm, global first-come first-served (*FCFS*). We show how this behaviour indicator may be decomposed into object (set) states and then into computations involving only basic monitoring data. The goal is to demonstrate the feasibility of our model for monitoring.

4.1. Behaviour Indicator

The definition of the Q-factor for the load sharing algorithm A, is defined in [Wan85a] as:

$$Q_A(\rho) = \frac{\text{mean response time over all jobs under } FCFS}{\sup_{\frac{1}{K\mu}\sum_{i=1}^{N}\lambda_i = \rho} \max_i \{\text{mean response time at } i\text{-th source under algorithm } A\}} \quad (1)$$

where

N = number of sources,

K = number of servers,

μ^{-1} = mean service time,

λ_i = arrival rate at i-th source,

ρ = aggregated utilization of system.

The response time for a job is defined to be the length of time from when the job arrives at the system until it departs from the system. The performance of algorithm A is measured by the response time it produces, and it will depend on the arrival rates to the system. The value of $Q_A(\rho)$ is derived when the arrival rates are varied such that algorithm A gets its worst performance, i.e. its mean response time is maximised.

The purpose of this definition of the Q-factor is to expose fundamental differences between the various algorithms. In [Wan85a] the performance of different algorithms are analysed under varying conditions, and readers interested in the analytical results are referred to that work. The quality of the *actual* load sharing in a *particular* system is a high level metric that can be computed by observing (monitoring) the system in question. We should thus be able to observe which algorithm A is the best for our system under varying conditions, as for instance arrival rate of jobs, client-server ratios, various mixes of server performances, etc.

In order to compute the *BI* from the observed data, equation (1) is rewritten as:

$$Q_A = \frac{E[T_{FCFS}]}{\max_i E[T_{A,i}]} \tag{2}$$

The components of the expression are explained below.

4.2. Decomposition

In order to compute equation (2) we need the mean response time for jobs from each source under algorithm A and for all jobs under *FCFS*. Since they will be derived from the response time for each job, jobs are the objects of interest, and the decomposition focuses on the information monitoring can give us relating to each job. The Q-factor behaviour indicator is derived from the object states which give the amount of time jobs spend in various parts of the system, which are in turn derived from the basic monitoring data which record the times when a job moves from one place or state to another. Throughout the example values in capital letters denote object states and object set states while values in small letters denote events and constitute the basic monitoring data.

4.2.1. Response Time Under Algorithm A

The mean response time at source i under algorithm A for all jobs k, $1 \le k \le n_i$, is an Object State, and is given by the expression:

$$E[T_{A,i}] = \frac{1}{n_i} \sum_{k=1}^{n_i} T_{A,i}(k)$$

where $T_{A,i}(k)$ is the response time for job k from source i.

The response time of each job from any source is the sum of the time spent servicing the job, the time spent in queues due to algorithm A, and the overhead from communication, which may be dependent on

$t_{i,arr}(k)$	time when job k enters the system at source i
$t_{A,i,queue}(k)$	time when job k is queued under algorithm A
$t_{A,i,dqueue}(k)$	time when job k leaves the queue under algorithm A
$t_{A,i,sched}(k)$	time when job k is scheduled for processing the first time under algorithm A
$t_{A,i,exit}(k)$	time when job k exits processing under algorithm A
$t_{A,i,dep}(k)$	time when job k leaves the system via source i

Table 1: *Basic events*

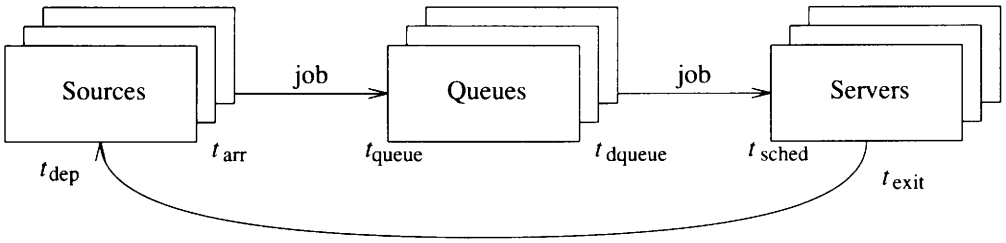


Figure 2: *Generation of basic events*

algorithm A. Thus, the response time for job k from source i under algorithm A (an object state) is given by:

$$T_{A,i}(k) = S_i(k) + Q_{A,i}(k) + O_A \tag{3}$$

where

$S_i(k)$ = service time for job k originating at source i

$Q_{A,i}(k)$ = total time in queue(s) due to algorithm A for job k from source i

O_A = communications overhead due to algorithm A.

These object states can now be decomposed to enable their calculation in terms of the basic monitoring data which are available from this system:

The service time for job k from source i, $S_i(k)$, is an object state defined as the time between the process performing the job first being scheduled for execution and the time when it exits. It is composed of two basic events, given by:

$$S_i(k) = t_{A,i,exit}(k) - t_{A,i,sched}(k)$$

The time spent by job k in a queue under algorithm A, $Q_{A,i}(k)$, is an object state given by:

$$Q_{A,i}(k) = t_{A,i,dqueue}(k) - t_{A,i,queue}(k)$$

The communications overhead due to algorithm A for job k comes from three sources: the time spent by the source searching for a queue and sending it there; the time spent by the server searching for a queue and getting the job from it; and the time spent returning the job to its source. Hence:

$$O_A = t_{A,i,queue}(k) - t_{i,arr}(k) + t_{A,i,sched}(k) - t_{A,i,dqueue}(k) + t_{A,i,dep}(k) - t_{A,i,exit}(k)$$

Not surprisingly, if these decomposed states are put back in equation (3), most of the terms cancel, leaving:

$$T_{A,i}(k) = t_{A,i,dep}(k) - t_{i,arr}(k) \quad (4)$$

However, the point of introducing the object states was to derive the service time $S_i(k)$, which is independent of the queueing algorithm, and is needed to calculate the response time under emulated *FCFS*.

As we can get the mean response time for all jobs at each source, the maximum value of all the values for $E[T_{A,i}]$ over all sources i can now be selected among these values.

4.2.2. Response Times Under Simulated FCFS

The mean response time for *all* sources i, $1 \leq i \leq s$, under *FCFS* is an Object Set State, and is given by the expression:

$$E[T_{FCFS}] = \frac{1}{s}\sum_{i=1}^{s} E[T_{FCFS,i}]$$

where

$$E[T_{FCFS,i}] = \frac{1}{n_i}\sum_{k=1}^{n_i} T_{FCFS,i}(k)$$

As with algorithm A, the response time for job k from source i under *FCFS* is given by:

$$T_{FCFS,i}(k) = S_i(k) + Q_{FCFS,i}(k) + O_{FCFS} \quad (4)$$

Again these break down as follows:

$$\begin{aligned} S_i(k) &= t^*_{FCFS,i,exit}(k) - t^*_{FCFS,i,sched}(k) \\ Q_{FCFS,i}(k) &= t^*_{FCFS,i,dqueue}(k) - t^*_{FCFS,i,queue}(k) \\ O_{FCFS} &= t^*_{FCFS,i,queue}(k) - t_{i,arr}(k) + t^*_{FCFS,i,sched}(k) \\ &\quad - t^*_{FCFS,i,dqueue}(k) + t^*_{FCFS,i,dep}(k) - t^*_{FCFS,i,exit}(k) \end{aligned} \quad (5)$$

The asterisks denote events which do not occur in the real system, since *FCFS* will not be running and therefore cannot be observed. These events can only be found by simulating the events that would have been observed if the same jobs were created at the same times in an identical system running an *FCFS* algorithm. The first thing to note is that the service time $S_i(k)$ is independent of the global load sharing algorithm, i.e.:

$$t^*_{FCFS,i,exit}(k) - t^*_{FCFS,i,sched}(k) = t_{A,i,exit}(k) - t_{A,i,sched}(k)$$

Secondly we make the approximation that the overhead is small enough to be ignored. Although two of the three time intervals in the overhead are independent of the algorithm, the time interval during which a queue is selected (either by the source for queueing a job or the server for scheduling a job) *does* depend upon which algorithm is used. Some algorithms need information about the queues before selecting one, some (*FCFS*, shortest job first) need information about the jobs on the queues. The time taken to get this information for simulated *FCFS* cannot (easily) be obtained by monitoring a system running a different algorithm. We therefore make the approximation that this time *is* independent of the algorithm, and that in fact all overheads are small enough to be ignored. This greatly simplifies the monitoring activity.

Thirdly, we must calculate $Q_{FCFS,i}(k)$. Having made the approximation that $O_{FCFS} = 0$, equation (5) can be rewritten:

$$Q_{FCFS,i}(k) = t^*_{FCFS,i,sched}(k) - t_{i,arr}(k)$$

However *FCFS* may be implemented, it can be modelled as a single central queue where jobs are entered in order of creation, and are removed each time a server becomes available for scheduling. Provided that we begin monitoring when the servers are first started we can continuously calculate the times when servers become available for scheduling and what jobs are on the queue, since we know when the jobs are created ($t_{i,arr}(k)$), how long they take to be processed ($S_i(k)$), and how long previous jobs spent on the queue.

The above analyses result in the following decompositions of the object states needed to calculate the Quality of Load Sharing behaviour indicator:

$$T_{A,i}(k) = S_i(k) + Q_{A,i}(k) + O_A = t_{A,i,dep}(k) - t_{i,arr}(k)$$

$$\begin{aligned} T_{FCFS,i}(k) &= S_i(k) + Q_{FCFS,i}(k) + O_{FCFS} \\ &= t_{A,i,exit}(k) - t_{A,i,sched}(k) \\ &\quad + t^*_{FCFS,i,sched}(k) - t_{i,arr}(k) \end{aligned}$$

where $t^*_{FCFS,i,sched}(k)$ is calculated from $t_{i,arr}(k)$ and $S_i(k)$ by simulation.

4.3. Implementing a Quality of Load Sharing Calculation

The monitoring activity begins with the programmer of the distributed service and client stub routines. For the purposes of these experiments a client-server program pair was written to simulate a service where 2 clients create jobs (of specific durations) and send them to 3 servers to perform them. Monitoring code was inserted into the client to generate "job created" events and to record the number of the job (job numbers are unique, one client using the even numbers starting at 0, the other the odd numbers starting at 1). Job durations are random with an exponential distribution; the time between creation of jobs is random with a poisson distribution. Between a job being created and being scheduled it is placed on a queue according to a queueing algorithm. The following algorithms were implemented: random splitting (client-initiated (*c-i*)), random server (server-initiated (*s-i*)), cyclic splitting (*c-i*), cyclic server (*s-i*), join shortest queue (*c-i*), service longest queue (*s-i*), shortest job first (*s-i*), first come first served (*s-i*). With client-initiated algorithms (where the client chooses which queue to put the job on) queues are maintained by the servers; with server-initiated

```
%%  ** Global context **
declare $queue[64]
set $queueHead 0
set $queueTail 0

declare $jobArrived[64]
declare $jobScheduled[64]
declare $FCFSScheduled[64]
declare $serviceTime[64]
declare $jobSource[64]

set $jobNumber 0

declare $nextFree[3]

declare $AlgATime[2]
declare $AlgANum[2]
set $FCFSTime 0
set $FCFSNum 0
```

Program 1: *Global variables*

```
%1  ** Client 1 local context **
jobArrival when event("job created")
on jobArrival do
  set $jobNumber state("job number")
  set $jobArrived[$jobNumber%64] @jobArrival
  set $jobSource[$jobNumber%64] 0
end

enable jobArrival

%2  ** Client 2 local context **
jobArrival when event("job created")
on jobArrival do
  set $jobNumber state("job number")
  set $jobArrived[$jobNumber%64] @jobArrival
  set $jobSource[$jobNumber%64] 1
end

enable jobArrival
```

Program 2: *Client local context*

algorithms, they are maintained by the clients. Either way, the servers know when a job is removed from a queue and scheduled for execution. Therefore the servers have monitoring code inserted to generate the events "job started" and "job finished" and to record the job number.

The SESL specification to drive the analysis of the incoming monitoring data is now described. It starts with the declaration of global variables (Program 1). The first group controls the queue of jobs for emulating *FCFS*. The head and tail pointers to the queue cycle around the `$queue` array; it is assumed that the queue never exceeds 64 jobs. The second group of arrays maintain information about outstanding jobs, including their arrival (creation) time, the time they are actually scheduled for execution, the time they would have been scheduled under *FCFS*, the length of time the job was being serviced, and the source of the job. `$jobNumber` stores the number of the job referred to by the most recent event notification. `$nextFree` is an array which records the time at which each server will next be available for scheduling under emulated *FCFS*. The remaining variables keep a running total of the response times and job counts for the actual algorithm (A) for each source, and for emulate *FCFS* (averaged over all servers).

The next two sections are specifications in the local context of the two clients (sources) (Program 2).

Internal events `jobArrival` are defined using specifications of the external low-level event "job created" ($t_{i,arr}(k)$)), which occurs in both

```
%3-5  ** Server local contexts **
jobSchedule when event("job started")
jobDeparture when event("job finished")

on jobSchedule do
  set $jobNumber state("job number")
  set $jobScheduled[$jobNumber%64] @jobSchedule
end
on jobDeparture do
  set $jobNumber state("job number")
  set $serviceTime[$jobNumber%64] @jobDeparture
        - $jobScheduled[$jobNumber%64]
end

enable jobSchedule, jobDeparture
```

Program 3: *Server local contexts*

clients. An action is defined for this event which sets the global `$jobNumber` variable and fills in the tables for that job. Because of the local context, the state `@jobArrival` (time at which the event occurred) refers only to the event defined in the same local context. Note that the only difference between these specifications is the number stored in the `$jobSource` table.

The next section is for the local context of the server, and is repeated three times, once for each server (Program 3). As with the client specifications, the low-level events are defined ("job started" ($t_{A,i,sched}(k)$) and "job finished" ($t_{A,i,dep}(k)$)), and job tables filled in with the actual times of the events under algorithm A.

Returning to the global context, an action is defined for `any jobArrival`, which is executed in addition to (and after) the action specified in the local context:

```
%%  ** Global context **
on any jobArrival do
  set $queue[$queueTail] $jobNumber
  set $queueTail $queueTail+1
end
```

The action puts the newly created job on the *FCFS* emulation queue. Here it will stay until the actual job has finished in order to calculate its service time ($S_i(k)$). Emulation of the *FCFS* queue is performed by actions occurring on the receipt of a `jobDeparture` event.

A state calculation is defined for convenience, which acts like a macro in a conventional programming language:

```
$nextServer := ($nextFree[2]<$nextFree[1] and
                        $nextFree[2]<$nextFree[3]) +
               2 * ($nextFree[3]<$nextFree[1] and
                        $nextFree[3]<$nextFree[2])
```

`$nextServer` returns the number of the server (0 to 2) which will next be available (in *FCFS* emulation) to service a job.

The next action is the meat of the specification, calculating the ordering of jobs under emulated *FCFS* (Program 4). Firstly the cumulative response time for algorithm A from the respective source is calculated from the times of the actual events. Then in a loop while the *FCFS* emulation queue is not empty *and* the service time for the job is known, the jobs on the queue are removed in order of arrival to be assigned to the next server that becomes available. If the job arrived before the server was (conceptually) available then it is scheduled at the time the server is next free; otherwise the job is scheduled at the same time as it was created. The time at which the server becomes free after this job is calculated (from the job's scheduling time and its service time) and stored in the `$nextFree` array. Note that the times the servers are conceptually active may be far in advance or behind the real time; the calculations can be performed any time after the service time of the job is known. Finally, the cumulative response time ($\Sigma T_A(k)$) and job count for *FCFS* emulation are calculated and the program returns to the top of the loop for the next job.

The last part of the specification defines the high-level states which measure the maximum response time for a server in algorithm A (max $E[T_{A,i}]$), the mean response time under (emulated) *FCFS* ($E[T_{FCFS}]$), and the Q-factor behaviour indicator (Q_A):

```
$S1Resp := $AlgATime[0]/$AlgANum[0]
$S2Resp := $AlgATime[1]/$AlgANum[1]
$AlgAResponse := $S1Resp*($S1Resp>=$S2Resp) +
                        $S2Resp*($S1Resp<$S2Resp)
$FCFSResponse := $FCFSTime/$FCFSNum
$QFactor := $FCFSResponse/$AlgAResponse
```

5. Conclusions

While the SESL implementation has been a useful demonstrator of monitoring techniques [Hol89a, Hol90a, Hol91b] the language itself has been found to be restrictive. For all its power in matching patterns of events, this ability has seldom been used; whether this is because of the nature of most monitoring activities, or just the scenarios chosen, or perhaps the style of the programmer, is not clear. Of much greater use were the procedural aspects of SESL when used to perform analysis of monitored data. The ability to run a procedure in response to a system event, using values measured in the system at that time, has been valuable. The fact that SESL has only a very small range of procedural constructs is a major limitation, and is explained by the fact that when SESL was originally conceived it had *no* procedures, as it was intended to be only a language for event and state combination. When the advantages were seen in extending it to perform analysis, the procedural constructs were added as they were needed. As a result, SESL has become a rather complex mix of declarative, imperative and procedural commands, difficult to learn both in terms of the range of commands, their syntax and effects (some are rather subtle), and how to put them together to form programs. If SESL was to be designed again from scratch it would surely be not only a useful, but a *usable* tool for monitoring distributed systems.

In this paper many problems regarding monitoring of distributed systems have not been discussed. These problems include the difficulties

```
on any jobDeparture do

  set $source $jobSource[$jobnumber%64]
  set $AlgATime[$source] $AlgATime[$source] +
        $serviceTime[$jobNumber%64] +
        $jobScheduled[$jobNumber%64] -
        $jobArrived[$jobNumber%64]
  set $AlgANum[$source] $AlgANum[$source]+1

while $queueHead!=$queueTail and
                $serviceTime[$queue[$queueHead%64]%64] do
  set $nextJob $queue[$queueHead%64]
  set $queueHead $queueHead+1
  set $server $nextServer
  ie $nextFree[$server] > $jobArrived[$nextJob%64]
    set $FCFSSheduled[$nextJob%64] $nextFree[$server]
  else
    set $FCFSSheduled[$nextJob%64] $jobArrived[$nextJob%64]
    set $nextFree[$server] $FCFSSheduled[$nextJob%64] +
                $serviceTime[$nextJob%64]

    set $FCFSTime $FCFSTime + $serviceTime[$nextJob%64] +
                $FCFSScheduled[$nextJob%64] -
                $jobArrived[$nextJob%64]
    set $FCFSNum $FCFSNum+1
  end
end
```

Program 4: *Job ordering*

concerning establishing a meaningful time, sharing of basic monitoring data and behavioral indicators, and the validity and usefulness of different viewpoints and abstraction levels of the "same" subsystem. Other problems not discussed are to what extent and in which way our approach to monitoring will influence the systems being monitored. This includes the potential problems that can occur when scaling the monitoring, for instance by having complex behavior indicators for a large subsystem.

Our experience to date suggests that the approach and techniques described in this paper seems to be useful for establishing higher-level insights in the function of certain aspects of distributed systems. We feel that our approach supports reasoning about monitoring of distributed systems, and supports a controlled envirnoment for development of tools for monitoring.

References

[Ans88b] Otto J. Anshus and T. Fallmyr, "A Framework for Monitoring," COST 11 ter/DSM (Tromsø):10 (1988).

[Ans88a] Otto J. Anshus, Terje Fallmyr, Tore Larsen, and Dag Johansen, "A Testbed for Distributed Operating Systems Using VMEbus Based Computers," pp. 141-146 in *Proc. VMEbus in Research*, NHPC, Amsterdam (October 1988).

[Bat83a] P. C. Bates and J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," *The Journal of Systems and Software*(3), pp. 255-264 (1983).

[Das86a] P. Dasgupta, "A Probe-Based Monitoring Scheme for an Object-Oriented Distributed Operating System," *ACM SIGPLAN Not.* **21**(11), pp. 57-66, Also in proc. of OOPSLA'86 (November 1986).

[Dre88a] G. D. Drenth, M. Wilcke, R. van Renesse, and H. van Staveren, "Monitoring in Distributed Client-Server Environments," Free University of Amsterdam report no. IR-155 (June 1988).

[Fal88a] Terje Fallmyr, Otto J. Anshus, Tore Larsen, and Dag Johansen, "The TRACE project; An Approach to Basic Research on Distributed Systems," in *Norsk Informatikk Konferanse (NIK '88)* (November 1988).

[Gar84a] H. Garcia-Molina, F. Germano jr., and W. H. Kohler, "Debugging a Distributed Computing System," *IEEE Trans. on Software Engineering* **SE-10**(2), pp. 210-219 (March 1984).

[Hab90a] Dieter Haban and Dieter Wybranietz, "A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems," *IEEE Trans. on Software Engineering* **16**(2) (Feb 1990).

[Hol88a] D. Holden, Otto J. Anshus, Terje Fallmyr, Jane Hall, Robbert van Renesse, Hans van Staveren, and Gunn Skogseth, "An Approach to Monitoring in Distributed Systems," pp. 811-823 in *Proc. Eur. Teleinformatics Conf.*, North-Holland, Vienna (1988).

[Hol89a] D. Holden, "Predictive Languages for Management," pp. 585-596 in *Proc. IFIP Sym. on Integrated Netrowk Management*, North-Holland, Boston (1989).

[Hol90a] D. Holden, "A Study on Control in the Distributed Systems Environment," in *Proc. IFIP Conf. on Local Comms. Sys. Management*, Canterbury, UK (1990).

[Hol91a] D. B. Holden, "An Implementation of Distributed Monitoring with SESL," Internal report DMP/45, Sys. & S/w Eng. Grp., AEA Technology, Harwell, UK (1991).

[Hol91b] D. B. Holden, "A Tutorial to Writing Programs in SESL," Internal report DMP/55, Sys. & S/w Eng. Grp., AEA Technology, Harwell (1991).

[Joy87a] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger, "Monitoring Distributed Systems," *ACM Transactions on Computer Systems* **5**(2), pp. 121-150 (May 1987).

[Lan88a] A. Langsford, "MANDIS – An Experiment in Distributed Processing," pp. 787-794 in *Proc. Eur. Teleinformatics Conf.*, North-Holland, Vienna (1988).

[Ogl88a] David M. Ogle, "Real-Time Monitoring of Parallel and Distributed Systems," Ph.D thesis, Ohio State University (1988).

[Ren87a] Robbert van Renesse, Andrew S. Tanenbaum, Hans van Staveren, and Jane Hall, "Connecting RPC-Based Distributed Systems Using Wide-Area Networks," in *Proc. 8th IEEE ICDCS*, Berlin (1987).

[Sae91a] Asbjoern Saetran, "Monitoring Network Activity; An Application of the COST11 ter MANDIS Approach to Monitoring," Master thesis, Univ. of Tromsø (April 1991).

[Wan85a] Y-T Wang and R. J. T. Morris, "Load Sharing in Distributed Systems," *IEEE Transactions on Computers* **c-34**(3) (March 1985).

Tools for Monitoring and Controlling Distributed Applications

Keith Marzullo Mark D. Wood

Cornell University, Ithaca, New York, USA

meta@cs.cornell.edu

Abstract

The Meta system is a UNIX-based toolkit that assists in the construction of *reliable reactive systems*, such as distributed monitoring and debugging systems, tool integration systems and reliable distributed applications. Meta provides mechanisms for instrumenting a distributed application and the environment in which it executes, and Meta supplies a service that can be used to monitor and control such an instrumented application. The Meta toolkit is built on top of the ISIS toolkit; they can be used together in order to build fault-tolerant and adaptive distributed applications.

1. Constructing Reactive Systems

In a *reactive system* architecture, the system is partitioned into two pieces: an environment that follows a basic course of action, and a control program that monitors the state of the environment in order to influence the environment's progress. This architecture is very general. For example, process control systems, system monitors and debuggers, and tool integration services all have a reactive system structure.

Another application of the reactive system architecture is the structuring of distributed applications. For example, many distributed applications are constructed by taking off-the-shelf programs and connecting them with some communication subsystem. Such an application can be thought of as an "environment" with a state including the properties of machines running the application, current performance of the component programs, and the state of the communication subsystem. The job of the control program is to monitor the state of the application in order to guarantee that the system operates efficiently in spite of changing load and failures. The control program can also be used to interconnect the application's components in a more loosely bound manner than conventional RPC mechanisms.

This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision. This work was also partially supported by a grant from Xerox.

Mark Wood was also partially supported by a G.T.E. Graduate Student Fellowship.

The Meta system, described in this paper, is a UNIX-based toolkit that provides the basic primitives needed to build a non-real-time reactive system. Using the toolkit, a distributed program can be instrumented with sensors and actuators in order to expose its state for purposes of control. Meta provides mechanisms that allow a control program to query the state of the instrumented application and to respond by invoking actuators when some condition of interest occurs. The toolkit includes facilities for structuring individual components into collections of components for fault-tolerance. In addition, Meta guarantees that the monitoring and reaction is done atomically.

Meta itself is built on top of another toolkit, the ISIS system. The application designer can use ISIS for fault-tolerant communication and Meta for distributed control. In fact, the Meta project was started when four of us in the ISIS project worked on integrating a distributed application constructed from off-the-shelf components [Mar90a]. The facility we found lacking in ISIS was support for distributed control.

The next section introduces the architecture of an application managed by Meta. Section 3 presents how applications are instrumented, and Section 4 discusses how the resulting application is controlled. Finally, Section 5 presents the current status of Meta and discusses our future plans.

2. The Meta Architecture

The architecture of Meta can be illustrated through an example of managing a distributed application. Consider an application that includes services and clients making use of the services. A given service consists of a set of identical servers replicated both for fault-tolerance and for coarse-grained parallelism. Meta will be used to manage the services; in particular, if the load on a service is too large or the number of servers becomes too small due to crashes, then a new server is to be started and added to the service. Additionally, if a server's queue becomes too long, then waiting requests are to be migrated to less-loaded servers in the service. There are other conditions that would probably need to be maintained as well, such as reducing the number of servers when appropriate, but for sake of brevity we will keep our example limited.

Meta structures a distributed application using a data model based on the entity-relation data model [Che76a], with each instrumented component (i.e., a program equipped with sensors and actuators) being viewed as an *entity* and its sensors and actuators being the *attributes* of that entity. For example, a server in the above example could be instrumented with sensors that give the server's load and the queue of waiting requests. Entities of the same type, that is, having the same set of sensor and actuator attributes, form an entity set.

Subsets of an entity set may be grouped together to form *aggregates*. Aggregate structures provide control programs with a way of grouping related entities together and limiting actions to members of that group. For example, the servers comprising a service can be grouped into an aggregate representing the service. Aggregates are themselves entities, and the system architect can define sensors and actuators on aggregates. An aggregate sensor is a function over the state of all the members of the aggregate. For example, a service aggregate could have a sensor that gives the median queue length of the servers in the

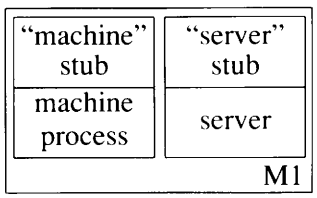


Figure 1: *An Instrumented Component*

service. An aggregate actuator causes an action to be performed on some subset (from one to all) of the current members.

A distributed application is managed through the use of guarded commands; that is, through a set of (*condition, action*) pairs that reference the sensors and actuators of the instrumented application. These commands are executed by interpreters that reside in *stubs* (somewhat like RPC stubs) coresident with the instrumented programs, thus allowing for fast notification and reaction. Each condition is a proposition on the state of system; references to both local sensorswithin the entity to which the stub is attachedand nonlocal sensors are allowed. The action portion is a sequence of actuator invocations that are executed atomically. Actions may enable guarded commands on another Meta stub; this facility allows one to write control programs that span multiple components.

Since guarded commands are evaluated in the same address space as an instrumented program, their impact on the performance of the application is a concern. The syntax of the guarded command language (a postfix language called NPL) is tailored for fast and efficient evaluation, and so we do not expect programs to be written directly in this language. We are designing an object-oriented control language called *Lomita* [Mar90a] that can be used to describe the structure of the application and to specify its control behavior. A Lomita program contains a schema specifying the entity and aggregate structure along with their sensors and actuators. The control behavior of the application is specified in Lomita through the use of rules, where the conditions for the rule may include real-time interval logic expressions [Sch83a]. Such temporal expressions are compiled into finite state automata, where the state transitions are implemented using Meta guarded commands.

Figure 1 illustrates the use of stubs. The machine `M1` is running a server that has been instrumented, so there is a stub running in the same address space as this server that can directly access the sensors and actuators of the server. The machine is also running a separate Meta-supplied program accessing the various properties of the machine and its operating system, such as the amount of available memory and the processor load. This program is instrumented, and so has a stub that supports a set of sensors and actuators over the machine and operating system state.

3. Application Instrumentation

An application first must be instrumented before it can be controlled. This is accomplished by inserting into the application a small amount of code, and then linking the application with a Meta library. This section describes the instrumentation process in more detail.

3.1. Access to Base Values

A sensor provides access to the value of some underlying system variable. An application defines a sensor with a Meta library routine:

```
meta_new_sensor(svr_q_length, "load",
                        TYPE_INTEGER, min_period);
```

This routine creates an integer-valued sensor named "load". When this sensor is referenced, the function `svr_q_length` in the instrumented program is called, which presumably returns the number of entries on the server's work queue.

In a reactive system, the fact that a sensor's value has changed is as important to know as the current value of the sensor. There are two methods by which an application can alert its stub that a sensor's value has changed. In some cases, a sensor's value changes either slowly or regularly, in which case a lower bound on the time between changes in its value can be determined. The application tells the stub this lower bound as the fourth parameter of the `meta_new_sensor` call. This value states how long that sensor's value can be cached before repolling is needed. In other cases, it would be very hard to determine such a lower bound. In this case, the fourth parameter of the `meta_new_sensor` call is zero, and the stub will obtain a fresh value only when the application makes an upcall to the stub. Such upcalls never block and can be made even when a nonzero polling period has been specified.

Actuators provide the means through which Meta acts upon the system. Like sensors, actuators are implemented by function calls in the application program. Actuators can be parameterized and can return either *success* or *failure*.

3.2. Functional Composition

A control program may wish to monitor a sensor whose value is a function of an existing sensor or sensors. For example, the control program may wish to monitor the maximum load of a server or the difference between two queue lengths supported by a server. Such sensors can be easily defined using Meta. A stub can construct functions of the sensors it supports and can define additional sensors in terms of these functions. The stub ensures that the sensors comprising such a sensor are sampled atomically. A extensive collection of pre-defined functions are available, and this collection can be augmented with user-defined functions.

3.3. Aggregates

An aggregate has, as predefined sensors, set-valued versions of the sensors on the components comprising the aggregate. For example, if a component has an integer sensor named `load`, then an aggregate of this component has a group sensor named `load` whose type is "set of integers" and whose value is the set of loads of the components. Other aggregate sensors can then be defined as functions of group sensors.

Just as an aggregate inherits the sensors of its components, an aggregate also inherits the actuators of its components. For example, if a component has an actuator named `run`, then an aggregate of this component has a group actuator named `run`. An invocation of the group actuator `run` invokes all of the component `run` actuators.

3.4. Fault-Tolerance

When necessary, sensor fault-tolerance is achieved through replication. The process containing the sensor to be made fault-tolerant is replicated, and the replicas are grouped into an aggregate; the value of the fault-tolerant, aggregate sensor is then a function of the members' sensor values [Sch90a].

The severity of sensor failures that can be tolerated depends on the choice of aggregate function. For example, to provide tolerance to crash failures, the aggregate function need only pick one of the member's values to return as the sensor value. In this case, the availability of the sensor is the same as the availability of any member of the aggregate. In process control systems, however, a real-world sensor such as the temperature of a reaction vessel can be represented as an interval bounding the actual value of the quantity being measured. In this case, a fault-tolerant intersection function can be used to mask arbitrary failures of sensors [Mar90b, Mar90c].

Group actuators are useful for achieving fault-tolerance in that they can be used to implement a *coordinator-cohort* style of actuation [ISI90a]. When invoking a group actuator, the command can include two additional parameters: an integer specifying the number of individual actuations to perform, and a preference list of aggregate members which indicates which aggregate members to try first. If the chosen actuator fails, then another member will be picked according to the preference list until the number of requested actuations is achieved or can not be achieved, in which case the group actuation fails.

4. Control

Once an application is instrumented, a control program can be written. The basis for controlling applications in Meta is a language of guarded commands that reference the state of the instrumented application.

4.1. Interpreting Guarded Commands

Each Meta stub implements a guarded command interpreter that has direct access to the sensors and actuators of the component to which the stub is attached. A stub can reference sensors and actuators not local to the component by communicating with the interpreter that does have direct access. The name of a sensor or actuator is sufficient for the Meta system to resolve which interpreter has direct access. So, a guarded command can be executed by any stub, although some stubs would provide better performance than others.

Since aggregates are not represented by a single component in the application, some stub must be selected to maintain the definitions of a given aggregate's sensors (and actuators). Exactly which stub computes the aggregate values is up to the application designer; either an existing stub or a "Meta server" (a stub instrumenting a dummy process) can be designated to do so, and other stubs can be designated as *cohorts*[†] that will take over in case the stub instrumenting the aggregate fails. This approach centralizes the computation of aggregate values, which in turn facilitates providing consistent views of the aggregate's state.

† These cohorts should not be confused with the cohorts in the ISIS *coordinator-cohort* facility, although the concept is the same. We are currently investigating how to best implement this structure.

The interpreters for Meta guarded commands may also be made fault-tolerant through replication. In this case, one interpreter is responsible for executing a given guarded command while the others remain as standbys. Sufficient state is exchanged among the replicas so that one of the standbys can take over in case the primary interpreter fails.

In our client-server example, the servers of a service are grouped into an aggregate. Each member of the aggregate (a server) has been instrumented, as described previously in Section 3, with a sensor that gives the load of the server. An aggregate sensor can then be defined that provides some measure of the service load, such as the median load of all the servers. If each server is equipped with an actuator that accepts a request for migration, then reliable migration can be implemented by invoking the set-valued aggregate actuator with the number of actuations specified as one and the preference list selected, for example, from the servers' loads. The stub that implements the aggregate sensors and actuators could be one of the servers in the service (presumably in the server stub) or a separate Meta server.

4.2. Atomic Guarded Commands

Recall that a guarded command consists of a set of (*condition, action*) pairs. A condition is a propositional expression over the sensor values, and an action is a sequence of parameterized actuator invocations. Ideally, Meta would ensure that the action is executed as an *atomic* command, that is, atomically and consistently with respect to its triggering condition [Lam84a].

When a predicate becomes true, the action should be executed in the same state in which it was triggered, but due to the asynchrony in the environment this can not be done without introducing blocking. Instead, Meta guarantees that any reference to sensor values during the action sequence obtains the same value as when the condition was triggered. Another property of atomic actions is that either all of the action is executed or none of it is executed. Providing this property requires a transactional facility with the ability either to undo the effects of partial actions or to invoke a forward recovery mechanism. Additionally, to provide consistent execution, the intermediate states of the action should not be visible to other guarded commands.

Meta currently provides only a limited amount of atomicity. For example, if a guarded command references only the sensors and actuators of a single component (either simple or aggregate), then its execution will be atomic. This amount of consistency is all that is needed for our client-server problem. For example, Meta will guarantee that if a machine is selected and removed from a *free-machine* aggregate when starting a new server, then the selection and removal will be done atomically. Other applications will require stronger guarantees of atomicity, however, so we are currently examining mechanisms that will enforce stronger guarantees of atomicity when necessary.

4.3. Example

Figure 2 shows part of a Lomita description of our client-server application. The description first defines the schema for server entities. In this simplified presentation, a server contains separate actuators for starting and stopping a job, with jobs being named by a string. For the sake of discussion, we assume that a job may be started and stopped repeatedly. The service aggregate has the sensor `sload` which is defined to be the median load of the individual sensors. The `run`

```
server: entityset
  attributes
    key name : string;
    sensor load: integer;
    sensor jobs: {string};
    actuator stop(string);
    actuator start(string);
  end
end


service: server aggregate
  attributes
    key port : string = "JobService";
    sensor sload : integer = median(load);
    actuator run(job : string) = start(job)[load,1,"<="];
    actuator create = ...;
    ...
  end
end


when server(Name).load > 5 do
  job = First(server(Name).jobs);
  server(Name).suspend(job);
  service("JobService").run(job);
end


when SIZE(service("JobService")) < 3 or
  during service("JobService").sload > 5 for 60
    always service("JobService").sload) > 5
do
    create(...);
end
```

Figure 2: *Job Service*

actuator starts a job on some member of the aggregate, and the preference list specifies that the member should be selected on the basis of its load.

The two rules shown in this figure are compiled into NPL programs. The first rule states that a job should be migrated from a server whose load is too high. This rule can be translated into a single guarded command that can run in the server's stub. The following C call distributes the NPL command to all server entities:

```
meta_npl("server",
         "load 5 > GUARD jobs First 'job' BIND job suspend
          job service('JobService').run");
```

This guarded command contains the conditional predicate *load* > 5 and then the action sequence of binding the variable `job` to the first job on the job list, suspending that job, and then resubmitting it for execution by invoking the `service` aggregate operator `run`.

The second rule is more complex; it states that if the size of a service is too small or the load remains high for too long, then a new server should be started. The Lomita compiler would translate this rule into a finite state automaton, which in turn would be implemented by a set of Meta guarded commands.

5. Discussion

The Meta project has explored the feasibility of toolkit-based architecture for building reactive systems and has applied this approach to distributed application management. Meta provides a uniform way of interconnecting disparate components, facilitating both the design of new systems and the construction of systems glued together from existing applications. Our approach has the benefit of separating management policies from their implementationthat is, how those policies are carried out.

5.1. Related Work

Although much work has been done on system monitoring, our work differs in that it combines control with monitoring to provide the general architectural support needed to construct a class of reactive systems. A prominent example of a system designed strictly for monitoring is the work of Snodgrass [Sno88a]; in his work, the system state is cast as a temporal database. Systems for debugging (especially those for debugging distributed systems), are a specialization of general monitoring systems. These systems provide a way to access the system state and to watch for certain predicates to be satisfied through the use of breakpoints [Bat88a, McD89a]. Of particular interest is the system IDD [Har85a] that permits interval logic expressions in specifying breakpoints.

Lomita is a rule-based language built on a real-time extension of interval logic. The rule-based language we have found most similar to Lomita is L.0 [Cam90a]. However, this executable language does not deal with the problem of instrumenting existing applications nor does it use a sensor-actuator data model. Configuration systems such as Conic [Kra89a] overlap with the use of Meta for distributed application management in that they facilitate interconnecting components, but they lack the means for specifying reactive behavior.

5.2. The ISIS System

Much of Meta depends upon facilities provided by the ISIS toolkit. One such facility is the notion of a *group*. An ISIS group is a named dynamic set of processes. Each member of the group has the same view of which processes are currently in the group despite other processes asynchronously joining the group, leaving the group and crashing. Among other uses, Meta uses ISIS process groups to implement atomicity of aggregate invocation and to organize the members of an aggregate.

Providing consistent behavior in Meta relies heavily upon the notion of *virtual synchrony* provided by the ISIS system [Bir87a]. The ISIS system make asynchronous events such as message receipts and group membership changes appear to happen synchronously. This property greatly facilitates reasoning about system behavior and constructing a system that behaves in a consistent manner. Fundamental to this property is the notion of an ordered broadcast. ISIS provides two important broadcast primitives [Jos89a]; *abcast*, which totally orders the broadcasts to a group, and *cbcast* which partially orders the broadcasts to a group dependent on the causal order of the broadcasts. For example, if two apparently concurrent events occur in the instrumented

application, Meta can impose a global total order on these events by using *abcast*.

5.3. Status

Several iterations of prototypes have been built with the latest being available from Cornell as part of the ISIS toolkit. Work is currently underway on a major release supporting the complete functionality described here. Preliminary performance figures from this work show the system to impose a low amount of overhead. The following benchmarks were obtained by running Meta on Sun 4/60's with interprocess communication handled by ISIS over a 10 Mbps Ethernet.

The time to execute a simple guarded command of the form `A GUARD B` with trivial local sensor `A` and trivial local actuator `B` is 84.1 microseconds, with uncertainty less than .1 microsecond. This implies approximately 12,000 guarded commands can be executed a second.

The bulk of the time for remote actions is of course in the message delivery. The ISIS causal broadcast (cbcast) takes 14.4 milliseconds;† the ISIS atomic broadcast abcast takes up to twice as long. Running the previous simple guarded command at a remote interpreter takes 32.6 milliseconds. This figure includes one cbcast to the interpreter to report the value and an abcast from the interpreter to effect the actuation.

The act of referencing a remote sensor has some initial start-up cost, which we call the *subscription* cost. Upon receiving a subscription request from some remote interpreter, a Meta stub will report all changes in the sensor's value to the subscriber. To get a feel for the subscription cost, we measured the time need to do the following: send to the local interpreter a guard that immediately triggers and causes the interpreter to subscribe to a remote sensor, get the first value, and cancel the subscription. This time was measured to be 66.2 milliseconds. This figure includes the time to parse the guard, but the cost of this should be negligible, less than one percent. Note that the guard is sent locally via cbcast rather than via a (faster) direct procedure call because we wish to support replication of interpreters. The cbcast therefore results in communication with the ISIS protocol server for that machine.

Note that all communication in Meta goes through the ISIS protocol server, a separate process running on each machine. Newer versions of ISIS now under development allow for restricted types of broadcasts to be sent directly to the intended recipients, bypassing the ISIS protocol servers. This results in considerable savings; a cbcast of this form only costs 5.6 milliseconds. The bypass mode of communication requires the sender and receiver to be in the same group, which is not typically the case in Meta. However, the current implementation of Meta does put aggregates in the same group, opening the way to use the bypass mode of communication, and we are currently exploring ways of exploiting it even further.

Previous versions of Meta have been released, but these did not support the complete NPL language but instead had the notion of a *watch*, in which a Meta stub could be instructed to wait for the value of some sensor to satisfy some relation. This earlier work has emphasized the

† Performance figures of the order of milliseconds are accurate to within 0.2 milliseconds with a confidence of 95%, except for the time to subscribe, which is accurate to within 1.1 milliseconds.

benefit of detecting conditions as close as possible to the site at which they become satisfied.

We are currently building a network manager as a test application for Meta, and are designing a debugging and monitoring tool and a system configuration system.

5.4. Directions

The current Meta toolkit is adequate for use in systems in which timing is not crucial. Although guarded commands can make temporal assertions, given the potentially unbounded latencies in the underlying UNIX and ISIS platforms, such assertions can only be viewed as approximate upper bounds. However, the structure that Meta provides is general enough that we should be able to extend it to real-time reactive systems as well.

There are two main obstacles we see to extending Meta to real-time systems. The first has to do with the underlying ISIS toolkit; to guarantee bounded reaction time, the underlying causal broadcast and group membership protocols must provide some real-time guarantees. A companion project in the ISIS group is currently looking into structuring ISIS under Mach to provide these two protocols. The second obstacle has to do with the semantics of guarded commands. Guarded commands currently have the semantics of atomic actions; if a guarded command is continuously enabled, then it will eventually execute. We need to add an upper bound on how long the command can be enabled without executing, and then build a scheduler that either guarantees the command will be executed within its deadline or aborts the command if it cannot be executed within its deadline.

5.5. Acknowledgements

Several people have contributed to the Meta project. Nancy Thoman designed and wrote the first version of the guarded command language, and Wanda Chiu designed a reactive relational database that provided a testbed for earlier versions of Meta. Kenneth Birman and Robert Cooper have contributed much to the design of the overall system. We would also like to thank Robert Cooper and Laura Sabel for their helpful comments on earlier drafts of this paper.

References

[Bat88a] Peter Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," in *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (1988).

[Bir87a] Ken Birman and Thomas Joseph, "Exploiting Virtual Synchrony in Distributed Systems," pp. 123-138 in *Proceedings of the Eleventh Symposium on Operating System Principles*, ACM SIGOPS (1987).

[Cam90a] E. J. Cameron, D. M. Cohen, L. A. Ness, and H. N. Srinidhi, "L.0: A Language for Modeling and Prototyping Communications Software," ARH--015547, Bellcore (April 1990).

[Che76a] P. P.-S. Chen, "The Entity-Relationship Model – Toward a Unified View of Data," *Transactions on Database Systems* **1**(1), pp. 9-36 (March 1976).

[Har85a] Paul K. Harter, Dennis M. Heimbigner, and Roger King, "IDD: An Interactive Distributed Debugger," pp. 498-506 in *Proceedings of the Fifth International Conference on Distributed Computing Systems* (1985).

[ISI90a] ISIS, *ISIS – A Distributed Programming Environment – User's Guide and Reference Manual*, Cornell University, Department of Computer Science, Upson Hall, Ithaca, New York 14853 (March 1990).

[Jos89a] Thomas Joseph and Kenneth Birman, "Reliable Broadcast Protocols," pp. 294-318 in *Distributed Systems*, ACM Press, New York (1989).

[Kra89a] Jeff Kramer, Jeff Magee, and Morris Sloman, "Constructing Distributed Systems in Conic," *Transactions on Software Engineering* **SE-15**(6), pp. 663-675 (June 1989).

[Lam84a] Leslie Lamport and Fred B. Schneider, "The "Hoare Logic" of CSP, and All That," *ACM Transactions on Programming Languages and Systems* **6**(2), pp. 281-296 (April 1984).

[Mar90a] Keith Marzullo, Robert Cooper, Mark Wood, and Ken Birman, *Tools for Distributed Application Management*, Cornell University (June 1990). Submitted for publication.

[Mar90b] Keith Marzullo and Mark Wood, "Making Real-Time Reactive Systems Reliable," pp. 1-6 in *Proceedings of the Fourth ACM SIGOPS European Workshop*, ACM SIGPLAN/SIGOPS (September 1990).

[Mar90c] Keith Marzullo, "Tolerating Failures of Continuous-Valued Sensors," TR 90-1156, Cornell University (September 1990).

[McD89a] Charles E. McDowell and David P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys* **21** (December 1989).

[Sch90a] Fred B. Schneider, "The State Machine Approach: A Tutorial," *Computing Surveys* **22**(3) (September 1990).

[Sch83a] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt, "An Interval Logic for Higher-Level Temporal Reasoning," *Proceedings of the Second Symposium on Principles of Distributed Computing*, pp. 173-186, ACM SIGPLAN/SIGOPS (1983).

[Sno88a] Richard Snodgrass, "A Relational Approach to Monitoring Complex Systems," *Transactions on Computer Systems* **6**(2), pp. 157-196 (1988).

Distributing Objects

Andrew Herbert

Architecture Projects Mangement Limited
Cambridge, England
ajh@ansa.co.uk

Abstract

Similar concepts, called "objects" have appeared in several areas of computing, from object-oriented databases, object-oriented programming languages, application environments and graphical user interfaces. These concepts have been reviewed by Alan Snyder of HP in a technical report called "The Essence of Objects". This paper builds upon Snyder's analysis and presents the requirements for adding distribution to the object concept. It is written for an audience who understand object orientation, accept Snyder's principles, and want to know how distribution might modify them.

What are "Objects"?

Similar concepts, called "objects" have appeared in several areas of computing, from object-oriented databases (such as Iris [Fis87a], object-oriented programming languages (such as C++ [Str86a], and Smalltalk [Roba], application environments (such as MacApp [Sch86a], ET++ [Wei88a], HP New Wave Environment [Hew89a] and graphical user interfaces (such as the HP New Wave Desktop).

These concepts have been reviewed by Alan Synder of HP in a technical report called "The Essence of Objects" [Sny89a] (a revision of which is scheduled for publication by IEEE Software during 1991). This paper builds upon Synder's analysis and presents the requirements for adding distribution to the object concept. It is written for an audience who understand object orientation, accept Synder's principles and want to know how distribution might modify them.

Definitions and commentary taken from Snyder's report are shown in italics.

The Essentials of Objects

Snyder characterizes the essential principles of objects as follows:

1. *An object is not just bits*
   - 1a. An object embodies an ***abstraction***
   - 1b. An object provides ***services***

2. Clients ***request*** services from objects
   - 2a. Objects are ***encapsulated***
   - 2b. Clients issue requests
   - 2c. Requests are named
   - 2d. Requests identify objects
   - 2e. Requests may take arguments and produce results
   - 2f. Services can be described
3. Requests can be ***generic***
4. Objects may be organized ***hierarchically*** in terms of the services they provide
5. Objects may be organized ***hierarchically*** in terms of the degree to which they share a common implementation
   - 5a. Objects may share a common implementation (multiple ***instances***)
   - 5b. Objects may partially share a common implementation (***implementation inheritance***).

The following sections explore the effect of distribution upon these principles.

1. An Object is not just Bits

An object is not just a data structure. It embodies an abstraction that is meaningful to its clients (users or programs). The purpose of the data is to represent abstract information. An object is more than just information: it provides a set of services to its clients. These services correspond to the embodied abstraction – they do more than just read and write data. The services are carried out by executing code that access or manipulates the actual data. The set of services provided by an object is called the ***behaviour*** *of the object.*

This principle shows the benefit of object-orientation in **heterogeneous** systems since it separates the service provided by an object from the implementation of that service.

Benefit: Objects provide application independence – different implementations of a service can be provided for different environments based on different programming languages, operating systems or hardware, provided that a uniform way of interworking with services is provided.

Benefit: Objects enable controlled, incremental evolution of a system – the implementation of a service can be changed **transparently** to all the clients of that service.

2. Clients Request Services from Objects

Clients respect the intent to use data to represent abstractions. Instead of directly accessing the data, clients issue requests for service that are carried out by objects.

2.a. Objects are Encapsulated

This principle is the basis for using objects in distributed systems: since objects are encapsulated they need not be in the same place as their clients provided that some means is provided for clients to identify and remotely access an object's services. Approaching the principle from the opposite direction, distribution – in the sense of separate location – enforces the encapsulation of objects and prevents direct access to data.

In non-distributed systems, the benefit of encapsulation is to guarantee that an object satisfies application-defined integrity constraints since there is no direct client contact with the data. The encapsulation property of objects equates two notions – objects as:

1. A unit of *service*, or unit of representation. This is an object in the sense of a design, or a representation of a part of a system. Within the object the design or representation can change without affecting the rest of the system.

   Benefit: Objects provide strong modularity in design and permit incremental development and evolution of designs.

2. A unit of *programming*: an object in an object oriented programming language, for instance. A unit of service would be made up of a one, or more, units of programming.

   Benefit: Objects in programs can be subject to scope-checking to ensure a program maintains the modularity of the design it implements and separately compiled and linked into multiple programs.

In distributed systems there are further integrity constraints which have to be met by encapsulation: the different types of encapsulation needed are:

3. A unit of *distribution*– encapsulation for objects that may migrate around a distributed system independently from other objects, but must remain integral within themselves.

   Benefit: Objects can migrate from one computer to offload functionality from a processor being taken out of service, to balance load between processors, to bring data to the processor where it is being used to reduce latency, to move data out to storage services when it is not in current use without involving the object's clients

4. A unit of *failure*– encapsulation for objects such that all of an object fails, or none of it does. This is usually achieved by keeping the whole object on the same processor.

   Benefits: Replication techniques can be used to make fault tolerant implementations of critical objects; distributed applications can be written to provide "graceful degradation" in the presence of object failures

5. A unit of *security*– encapsulation such that all of the components of an object are subject to the same access control rules. This usually means the components belong to the same principal and that interaction within the components of the object is not subject to access control

   Benefits: Hardware protection mechanisms can be used to physically enforce object encapsulation; encipherment can be used to physically protect request and responses between objects.

In a distributed system, the boundaries of these units are not identical; but in modelling and discussing object oriented distributed systems

each of these different units is collectively known as an "object". All of these units exist in an object oriented programming system, but they are implicit and implicitly bound to the program unit. Consequently, the object oriented programming concept of an object can be seen as a simplification of the more general concept of object which arises in object oriented distributed systems.

The challenge in a distributed system is to manage the different units (objects) at run-time with a mixture of programming language and configuration tools; but without the simplifications of a single programming environment and recompilation assumed by object oriented programming systems.

2.b. Clients Issue Requests

Clients issue requests for services that are carried out by objects. A request causes code to be executed to perform the requested service. The details of where and how this code runs is intentionally not of concern to the client.

In a distributed system there may be many clients on separate processors simultaneously requesting the same service and therefore a server object must be able to exercise concurrency control over requests. Concurrency control takes two forms:

1. *Ordering and synchronization*: the server may require that requests be processed one at a time, or that only certain sequences of overlapped execution are possible (for example, producers and consumers can both access a finite buffer while there are free slots in it), or that only a certain number of requests be executed at once to limit the consumption of resources.

2. *Separation*: the server may require that sequences of requests from separate clients be scheduled in an order that avoids conflicting updates to the data contained in the object (for example conflicts between reads and write to the records of a data base object). A client may wish to abandon a sequence of requests if an intermediate request produces a particular result (e.g. a debit on an empty account), or if an object to be invoked as part of the sequence fails. This requires that the execution model support the notion of *commiting* and *aborting* sequences of requests and the correct interplay of *commit* and *abort* with the scheduling mechanisms for separation (i.e. transaction processing capabilities).

In a distributed system objects may be remote from their clients, introducing a latency due to the overheads of communication. A client may be able to reduce latency by making concurrent requests to different objects (or even the same object if its concurrency controls permit). It must be possible to indicate that a concurrent request is either part of a transaction or starting a new independent transaction.

2.c. Requests are Named

Requests are typically named. To make a request, the client identifies the request by name.

2.d. Requests Identify Objects

To make a request a client must identify one or more objects to perform the requested service. An object can identified directly or reliably. Object reference is direct in the sense that one is naming the object not describing it. Object reference is reliable in that, within certain limits of time and space, repeating reference to an object will reliably access the same object.

Requests are directed towards objects which are units of service; it may be that an object which is a unit of distribution encapsulates several units of service. The latter can be conveniently termed the *interfaces* of the distributed object (and thus objects in object oriented programming languages can be equated with distributed objects containing just one interface). For example, there may be some data which is encapsulated in a single object for reasons of security, but for which there is the notion of both "user" services and "manager" services. The two forms of service can be readily distinguished by putting each in a separate interface. In a distributed system requests identify **interfaces**.

In many distributed systems there is no notion of "system restart" and so an object (i.e. interface) reference has to retain its meaning for all time. Nor can it be assumed that separate distributed systems will never become joined (for example when the organizations merge or do business with one another) and so an interface reference has to retain its meaning throughout space.

It may not be possible in a distributed system to distinguish between an object which cannot be accessed because of disruption of communications and an object which has become lost from the system because it did not take steps to ensure its reliability. Clients must be prepared to cope with the failure of communication, and objects which use replication to increase their stability must take steps to ensure the replicas present a consistent service to their clients even if replicas are unable to communicate with one another.

Since objects may migrate in a distributed system, interface references must be **location independent** names. A distributed system must include a **location service** for discovering the current location – i.e. address – of objects which have migrated so that requests can be delivered to the correct place. (For objects which will never migrate the interface reference can include an address for the interface to save on the time overheads of name to address resolution and the potentially vast overhead of storing name to address translations for all interfaces).

Client to server binding can be **early** or **late.** In early binding client and server are made together and an interface reference for the servers embedded in the client. Late binding is a dynamic process, called **trading**. A server object providing a service registers (or has registered on its behalf) a description of the service provided and its interface identifier with a **trading service**. A client object wanting to use a service queries the trading service, and if a matching service offer is found its interface reference is returned. The client can then use the interface reference to make requests.

Both location and trading services may be built upon **name services** which provide name-to-name translations.

Service descriptions given by servers and clients must necessarily describe the range of services available at the interface so that the client gets access to an object providing at least the service required. There may be many objects providing a suitable service and therefore service descriptions may involve names and attributes to permit disam-

biguation between service offers. For example a service may be further qualified by its location, who owns it, how secure it is, how fast it is, how robust it is against failures, and so on.

If a client makes purely functional use of a service (i.e. does not require that the service keep state on its behalf), the client may elect to rebind on every request rather than retain the interface reference found on the first attempt at trading. Alternatively, a functional client may only rebind if the service it is using becomes inaccessible because of failure or communications problems.

> *Benefit*: Providing a trading service makes a system configuration open-ended – new objects can be added to the system and made accessible to existing clients without requiring that the clients be rebuilt in any way. Interface references (i.e addresses) need not be built into programs.

2.e. Requests may take Arguments and Produce Results

Particularly in a computational context (as opposed to a user interface), it is commonly the case that a request may have associated argument values (which may be object references) and the service may return one or more results (which may also be object references) when it completes.

In a distributed system all arguments and results have to be either interface references or immutable data types (i.e. integers, booleans, characters etc). It is not meaningful to pass pointers since client and server may not be on the same computer. (Some systems give the illusion of passing pointers by wrapping them up as an interface reference to a service for accessing memory locations, or by copying the data referenced by the pointer).

Passing an interface reference gives the recipient the right to share in the use of a service (hence the need for the concurrency controls mentioned in Section 2b). Passing an immutable data type requires that a copy of the data type be made at the recipient. In a computational model in which all data types are objects, both these schemes can be viewed as providing sharing semantics, as can other schemes such as migrating the object to the recipient, or replicating the object so that both sender and recipient have local copies kept in step by some sort of consistency protocol.

> *Benefit*: Treating all arguments and results as interface references (i.e. a pure object model of data) provides a clean computational abstraction of a wide range of argument and result passing schemes. Alternative schemes can be substituted without requiring changes to the programs involved.

Many programming languages only permit a request to return a single data type as result. Often what is returned is a memory address for a data structure made by the called service. Since memory addresses cannot be permitted as results in distributed systems, an interface reference to a result object constructed by the service would have to be used instead. But this then imposes a significant latency overhead when the recipient of the reference tries to access the object. Therefore in a distributed system it should always be possible to pass multiple arguments and obtain multiple results.

In a distributed system a request may fail because of some communications problem or resource limitation. This fact has to be conveyed back to the caller as an abnormal outcome of the request. It may also be that the service has several possible outcomes. These could be encoded as

a datatype – a discriminated union for example; alternatively a general mechanism permitting a request to generate different outcomes could be provided, with facilities for the requestor to take different actions depending upon the particular outcome obtained for any given request.

The synchronous request-response style of interaction is well suited to distributed computing. It matches well with the concept of **remote procedure call** found in many distributed systems architectures. It also fits well with the concept of a**nested transactions**as guarantees given in Section 2b. Note that a request which returns no results is strictly a request that returns an "empty" response – the response contains no results, but there is an explicit indication of termination. Request – response interactions create chains of dependent nested calls. In 2b it was noted that there is a need to establish new independent activities: this can be modelled by a service which can be requested but which produces no response at all – a "fire and forget" style of interaction.

It is useful to compare conventional object-orientation and remote procedure call in terms of the object with the "multiple interfaces, each interface supporting several services" model outlined above. Conventional object-oriented systems merge the concepts of object and interface into the single concept "object" and thereby lose the ability to determine which services are available to which clients and the ability to distinguish clients by giving each one a separate interface and associating client state with the interface used. Remote procedure systems merge the concept of interface and request together into the concept "procedure" and thereby lose the benefits of encapsulation and abstraction that come from grouping services together. Some remote procedure call systems do provide the notion of interface so that services can be grouped to form an abstraction, but they do not provide means to pass such interfaces as arguments and results, and therefore lack the flexibility of object-oriented systems. The general object / interfaces / services model supports both conventional object-orientation and remote procedure call as a special case.

2.f. Services can be Described

The set of services provided by an object to its clients is often made explicit to clients in the form of an ***interface description*** *that identifies a set of requests that can be made to an object. This interface description is sometimes called a protocol. Often this specification will include information about the expected arguments and results associated with each request. Such a specification is sometimes called a signature.*

(In distributed systems the term protocol usually refers to the means provided by networks to copy data from one computer to another.)

Services must be described by signatures in distributed systems, since clients and servers are often written by different programmers in different locations and at different times. The signature provides a contract between the two programmers, telling them what service is to be provided at an interface to an object. It may be that the two programmers use different languages to write their programs and the programs run on computers with different data representations. The signature provides the information need to automatically generate the data type conversions needed to permit interworking between client and server.

> *Benefit*: Signatures permit decoupling of client and server programs and the use of multiple implementation languages.

As discussed in Section 2d, there may be many objects providing the same service – i.e. with the same signature. Therefore in a trading system additional attributes beyond signatures must be used to distinguish between different offers of the same service.

3. Requests can be Generic

A client can issue the same request to different kinds of object that provide "similar" (at least homonymous) services. Depending upon which objects are identified by the client, different code may be run to perform the requested service. The selection of code to execute is based on the object identified by the clients in the request. In the general case, the identification of objects is not determined until the request is actually issued, so the selection of code would happen at that time. (In some cases information exists at compile time or link time to statically bind a request to the code that will implement it). Also in the general case there is no limit on the number of different kinds of object that may support a given request.

The benefit of generic request in distributed systems is that services can be more general, which implies more re-usable, and that users benefit by being able to apply a standard model in many cases. For example all objects can be made to support a common management model by requiring they support a common management interface.

An open system is one in which new objects can be introduced dynamically, such that the new objects can be operated upon by existing clients, without changing the existing clients. The existing clients are able to use the new objects because the new objects support the requests for generic services made by the existing clients.

> *Benefit*: Openness is mandatory requirement for practical distributed systems.

4. Objects may be Organized Hierarchically in Terms of the Services they Provide

Objects can be classified in terms of the services they provide to clients or equivalently, in terms of the requests that can be made of them. Objects that provide the same set of services would be classified together. This classification may be based on explicit descriptions of services (or requests) called interfaces.

(Note: since the term "interface" has a particular meaning in distributed systems – see 2d above – classifications of services will be called **types** in the following discussion.)

An object could provide a subset of the services provided by another object, leading to a hierarchical classification. This interface hierarchy can be used as a type hierarchy in describing permissible values for arguments to procedures in a program etc.

> *Benefit*: A classification of objects based on their services is a way of organizing objects to make their behaviours easier to understand. A classification of object services can also be used to describe the services expected of an object by a client.

The need for type descriptions – signatures – in distributed systems was discussed in Section 2f. Organizing types hierarchically eases the burden of writing such descriptions since a complex service can be defined

as being an extension of a set of simpler services. This is particularly useful when there are large numbers of generic requests that can be made of an object.

The existence of a type hierarchy means that the model of type-checking in a distributed system should be one of **type conformance** rather than type equality: a client request is acceptable to an object if the object is capable of responding to the request, if the client offers arguments which conform to those expected by the server and if the server returns results which conform to those expected by the client. The use of type conformance increases the genericity of objects and hence the openness to service evolution in a system since it permits an object to incrementally "widen" the type of an interface without disrupting the client.

5. Objects may be Organized Hierarchically in Terms of the Degree by which they Share a Common Implementation

Mechanisms are generally provided (in object systems) to allow different objects to share the same implementation. Mechanisms are often provided by which the implementation of one object can not only share the implementation of another object, but also extend or refine it.

5.a. Objects may Share a Common Implementation (Instances)

*The implementation of an object generally specifies both the format of the data used to represent the information associated with an object and the code used to implement the services it provides. Mechanisms are generally provided to allow different objects to share the same implementation. Objects that share a common implementation have identical data formats and share executable code; however each object has its own copy of the actual data. Objects that share the same implementation would be classified together. Each object can be thought of as an **instance** of the common implementation.*

It is the sharing of implementations that is of primary interest, not the classification resulting from it. In general clients should be concerned with the services provided by an object, not how the services are implemented, and thus should not be interested in an implementation-oriented classification.

> *Benefit*: Sharing one implementation among many objects has the obvious benefit of reduced source code duplication (which eases maintenance by avoiding the need for manual propagation of changes) and reduced executable code size (where sharing of executable code is possible).

In a distributed system there may be instances of a common implementation on many different computers with different data representations and instruction formats, so sharing of a single execution image and data format is not possible (except in the special case where several instances reside on the same computer).

Some object systems are **reactive** in that a change to the source code for a set of instances is immediately reflected in a change in behaviour of the instances. This is a difficult effect to achieve in a distributed system; it raises many questions about consistency and atomicity since there is not a single copy of the executable code to be updated.

Programmers may wish to exercise control over where an instance is created. This is readily accomplished by providing **factory** services which create new instances upon demand. A factory service embodies a template for the class of which the object to be made is an instance. All the objects made by the same factory may share the same data format and executable code.

If an object is to be able to migrate from one computer to another means must be provided for the object to externalize itself into a representation which can be moved between machines and re-instantiated at the destination. In the general case, the external representation must include information about all the activities that were taking place inside the object at the moment it began migrating. If the source and destination computers are identical, the external representation can be close to the internal representation. If the potential destinations for a migrating object are known in advance the size of the external representation can be reduced by pre-arranging for the code and data formats to exist at the destinations.

5.b. Objects may Partially Share a Common Implementation

Mechanisms are often provided by which the implementation of one object can not only share the implementation of another object, but also extend or refine it. (Such mechanisms are generally called ***inheritance*** *mechanisms.) In this case of partial sharing implementations, the classification of object implementations becomes hierarchical.*

> *Benefits*: In addition to the maintenance and size benefits listed above, partial sharing of implementations extends the benefits of software re-use to cases where implementations are similar but not identical. Partial sharing is a useful technique for encouraging consistent behaviour among related objects.

Inheritance has a number of properties that make it unsuitable for general use in distributed systems [Raj89a]:

1. *Encapsulation is violated* – Inheritance may violate encapsulation in at least three ways: a subclass may (a) refer to data defined in the superclass, (b) request an internal service of the superclass, and (c), refer to the superclasses of its superclasses. The consequence of this in a distributed system is that the locality of objects is lost – inheritance introduces object dependence on unknown, potentially remote, inherited information defeating the major benefits of objects as independent units of migration, failure propagation and security.

3. *Classes are not automatically reusable* – For successful reuse, inheritance requires the use of a set of coding rules and a set of design rules to ensure consistent interpretations. In a distributed context it is not viable to expect all code to be written to the same conventions except in as much as there must be a commitment to the same means of interaction between objects. The internal structure of objects is a local concern guided by the implementor's local rules. (Indeed it cannot be assumed in a distributed system that all implementors are using object-oriented languages, let alone the same language and the same inheritance structure!)

4. *Class organization is not scaleable* – Inheritance is successful where software is written by a few people working together with an agreed hierarchy and where the number of classes is hundreds at most; inheritance falls down when there are large numbers of

classes involved or where there are large numbers of people involved.

5. *Reactive inheritance is difficult to achieve* – Reactive inheritance requires a consistent, atomic update be made to all members of a class and its sub-classes wherever they are located.

6. *There should be no linkage between typing and implementation* – The desirability of types as a means to permit multiple implementations of the same service has already been discussed. Many object-oriented systems use inheritance as a substitute for type-checking – two objects are deemed to be of the same type if they are made from the same components. This is too restrictive a view for a distributed system. Implementations have no part to play in the classification of services.

Thus implementation inheritance has little part to play in distributed systems. The objectives of maintenance and re-use must be met by techniques for the identification, sharing and composition of source code components alone. (Only where a same component is referenced several times by objects which are going to be co-located on the same computer is there scope for sharing object code as an optimization). If maintenance and re-use are conducted at this level, many other tools beyond inheritance become available for classifying and linking together software components. Distributed systems are facilitated by objects whose definition and implementation are fully self-contained.

References

[Fis87a] D. H. Fishman, "Iris: An Object-Oriented Data Base System," pp. 48-69 in *ACM Transaction on Office Automation Systems, 5 (1)* (1987).

[Hew89a] Hewlett-Packard, "HP New Wave User Guide ," in *Part number 5958-9678* (August 1989).

[Raj89a] R. K. Raj and H. M. Levy, "A Compositional Model for Software Reuse," in *Technical Report TR 89-01-04, Department of Computer Science, University of Washington, Washington* (1989).

[Roba] A. Goldberg and D. Robson, "SmallTalk 80: The Language and its Implementation," in *Addison-Wesley, Reading, Massachusetts* (1983).

[Sch86a] K. J. Schmucker, "MacApp: An Application Framework," pp. 189-193 in *Byte 11 (8)* (August 1986).

[Sny89a] A. Snyder, "The Essence of Objects," in *Report STL-89-25, Hewlett-Packard Laboratories, Palo Alto, California* (1989).

[Str86a] B. J. Stroustrup, "The C++ Programming Language," in *Addison-Wesley, Reading, Massachusetts* (1986).

[Wei88a] A. Weinand, E. Gamma, and R. Marty, "ET++ – An Object-Oriented Application Framework in C++," pp. 46-57 in *Proceedings OOPSLA 1989* (1988).

A Comparative Study of Five Parallel Programming Languages

Henri E. Bal

Vrije Universiteit, Amsterdam

bal@cs.vu.nl

Abstract

Many different paradigms for parallel programming exist, nearly each of which is employed in dozens of languages. Several researchers have tried to compare these languages and paradigms by examining the expressivity and flexibility of their constructs. Few attempts have been made, however, at *practical* studies based on actual programming experience with multiple languages. Such a study is the topic of this paper.

We will look at five parallel languages, all based on different paradigms. The languages are: SR (based on message passing), Emerald (concurrent objects), Parlog (parallel Horn clause logic), Linda (Tuple Space), and Orca (logically shared data). We have implemented the same parallel programs in each language, using real parallel machines. The paper reports on our experiences in implementing three frequently occurring communication patterns: message passing through a mailbox, one-to-many communication, and access to replicated shared data.

1. Introduction

During the previous decade, a staggering number of languages for programming parallel and distributed systems has emerged [And83a, Bal89a]. These languages are based on widely different programming paradigms, such as message passing, concurrent objects, logic, and functional programming. Both within each paradigm and between paradigms, heated discussions are held about which approach is best [Car89a, Kah89a, Sha89a].

The intent of this paper is to cast new light on these discussions, using a practical approach. We have implemented a number of parallel applications in each of several parallel languages. Based on this experience, we will draw some conclusions about the relative advantages and disadvantages of each language. So, unlike most of the discussions in the literature, this paper is based on actual programming experience in several parallel languages on real parallel systems.

This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.).

A preliminary version of the paper appeared in the Proceedings of the PRISMA Workshop on Parallel Database Systems, Noordwijk, The Netherlands, September 1990.

Language	Paradigm	Origin
SR	Message passing	University of Arizona
Emerald	Concurrent object-based language	University of Washington
Parlog	Concurrent logic language	Imperial College
Linda	Tuple space	Yale University
Orca	Distributed shared memory	Vrije Universiteit

Table 1: *Overview of the languages discussed in the paper*

The languages studied in this paper obviously do not cover the whole spectrum of design choices. Still, they represent a significant subset of what we feel are the most important paradigms for parallel programming. We discuss only a single language for each paradigm, although other languages may exist within each paradigm that are significantly different.

The languages that have been selected for this study are: SR, Emerald, Parlog, Linda, and Orca (see Table 1). SR represents message passing languages. It provides a range of message sending and receiving constructs, rather than a single model. Emerald is an object-based language. Parlog is a concurrent logic language. Linda is a set of language primitives based on the Tuple Space model. Orca is representative of the Distributed Shared Memory model.

We focus on languages for parallel applications, where the aim is to achieve a speedup on a single application. These applications can be run on either *multiprocessors* with shared memory or *distributed* systems without shared memory. We have selected only languages that are suitable for *both* architectures. So, we do not discuss shared-variable or monitor-based languages, since their usage is restricted to shared-memory multiprocessors. Functional languages are not discussed either. Most functional languages are intended for different parallel architectures, (e.g., dataflow or graph reduction machines) and often try to hide parallelism from the programmer. This makes an objective comparison with the other languages hard. We also do not deal with distributed languages based on atomic transactions (e.g., Argus [Lis88a]), since these are primarily intended for fault-tolerant applications. The issue of fault-tolerant parallel programming is discussed in a separate paper [Bal90a].

The outline of the rest of this paper is as follows. In Section 2, we will briefly describe the applications we have used, focusing on their communication patterns. Next, in Sections 3 to 7, we will discuss each of the five languages, one language per section. Each section has the following structure:

- Background information on the language. (All languages have been described in a recent survey paper [Bal89a], so we will be very brief here.)
- A description of our programming experience. We will comment on the ease of learning the language and on the effort needed to implement the communication patterns discussed in Section 2.
- Comments on the language implementation and its performance. Unfortunately, there is no single platform on which all the languages run, so we had to use many different platforms. The systems we used differ in the number of processors, processor type and speed, as well as in the way processors are intercon-

nected. A fair comparison between the languages is therefore not possible, but the measurements do give some rough indication of the relative speedups that can be obtained.

- Conclusions on the language.

Finally, in Section 8, we will compare the approaches used for the different languages.

2. The Applications and their Communication Patterns

There are many ways to compare parallel languages. One way is a theoretical study of the expressiveness of their primitives. This works well for languages using the same paradigm (e.g., message passing), but is more problematic for comparison between different paradigms. Comparing, say, remote procedure calls and shared logical variables is not a trivial task.

The approach taken in this paper is to implement a set of small, yet realistic, problems in each language, and compare the resulting programs. The example problems we have used include matrix multiplication, the All Pairs Shortest Paths problem, the Traveling Salesman Problem, alpha-beta search, and successive overrelaxation.

The applications and the algorithms used for them are described in detail in [Bal90b]. For this paper, we will restrict ourselves to only two applications: the All Pairs Shortest Paths problem and the Traveling Salesman Problem. These applications will be described below. We will focus on the communication aspects of the applications, since, from a parallel programming point of view, these are most interesting.

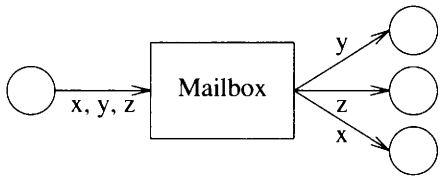
The Traveling Salesman Problem (TSP)

The Traveling Salesman Problem computes the shortest route for a salesman among a given set of cities. The program uses a simple branch-and-bound algorithm and is based on replicated workers style parallelism [And89a, Car86a]. The TSP program uses two interesting communication patterns: mailboxes and replicated shared data.

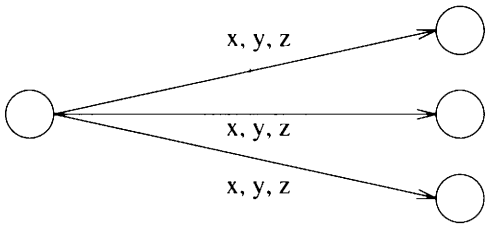
A *mailbox* (see Figure 1a) is a communication port with **`send`** (non-blocking) and **`receive`** operations [Bal89a]. Mailboxes can be contrasted with direct message passing, in which the sender always specifies the destination process (receiver) of the message. With mailboxes, any process that can access the mailbox can receive a message sent to it. So, each message sent to a mailbox is handled by one process, but it is not determined in advance which process will accept the message.

The TSP program uses a mailbox for distributing work. A process that has computed a new job (to be executed in parallel) sends it to a mailbox, where it will eventually be picked up by an idle worker process. Since it is not known in advance which worker process will accept the job, mailbox communication is required here, rather than direct message passing.

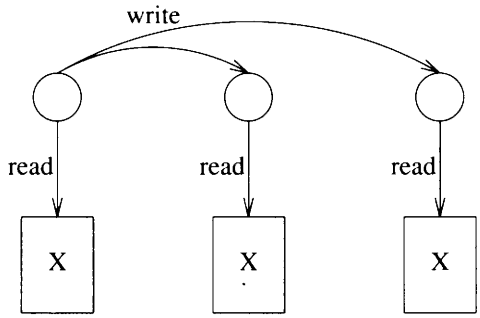
The second communication pattern used in the TSP program is *replicated shared data* (see Figure 1c). The branch-and-bound algorithm requires a global variable containing the length of the current best solution. This variable is used for pruning partial solutions whose initial paths are already longer than the current best full route.



(a) *Message passing through a mailbox (used by TSP)*



(b) *One-to-many communication (used by ASP)*



(c) *Communication through replicated shared data (used by TSP)*

Figure 1: *Communication patterns used by the two applications discussed in the paper*

In a distributed system, this global variable cannot be put in shared memory, since such systems lack shared memory. One solution is to store the variable on one processor and let other processors access it through remote operations. For TSP (and many other applications), however, a much more efficient solution is possible. The bound is usually changed (improved) only a few times, but may be used millions of times by each processor, so its read/write ratio is very high. Therefore, the variable can be implemented efficiently by *replicating* it in the local memories of the processors. Each processor can directly read the variable. Physical communication only occurs when the variable is written, which happens infrequently.

The All Pairs Shortest Paths Problem (ASP)

The second application is the All Pairs Shortest Paths problem, which computes the lengths of the shortest paths between each pair of nodes in a given graph. ASP uses a parallel iterative algorithm. Each processor is assigned a fixed portion of the rows of the distances matrix. At the beginning of each iteration, one process sends a *pivot row* of the matrix to all the other processes. Each process then uses this pivot row to update its portion of the matrix.

The most important communication pattern of the ASP program thus is *one-to-many* communication (see Figure 1b). This pattern transmits

data from one process to many others, *all* of which use these data. (In contrast, a message sent to a mailbox is used by only *one* process.)

Of course, this pattern can be simulated through multiple point-to-point messages, but frequently much better solutions are possible. Many networks have a *multicast* or *broadcast* capability, which can be used to speed up one-to-many communication significantly. So, there are two issues involved here: how one-to-many communication is *expressed* in a given language and how it is actually *implemented*. For ASP, it is very important that the implementation uses a real multicast. Otherwise, the communication costs may easily become a dominating factor.

3. Synchronizing Resources (SR)

SR [And86a, And88a] is a language for writing distributed programs, developed by Greg Andrews, Ron Olsson, and their colleagues at the University of Arizona and the University of California at Davis. The language supports a wide variety of (reliable) message passing constructs, including shared variables (for processes on the same node), asynchronous message passing, rendezvous, remote procedure call, and multicast.

Programming Experience

Given its ambitious goal of supporting many communication models, it is not surprising that SR is a fairly large language. Yet, we found it reasonably easy to learn. With regard to the sequential parts, the syntax, type system, and module constructs are different from most other languages. Nevertheless, these were fairly easy to learn, although the type system is far from perfect [Bal90c].

SR tries to reduce the number of concepts for distributed and parallel programming by using an *orthogonal* design. There are two ways for sending messages (blocking and nonblocking) and two ways for accepting messages (explicit and implicit). These can be combined in all four ways, yielding four different communication mechanisms. We agree with the designers that this orthogonality principle simplifies SR's design. Unfortunately, there also are some less elegant design features. The concurrent-send (`co`) command, for example, is a rather ad-hoc extension of the basic model, with specialized syntax rules.

Our programming experience indicates that, even within the restricted domain of parallel programming, nearly all facilities provided by SR are useful. We found uses for synchronous and asynchronous message invocation, explicit, implicit, conditional, and ordered message receipt, and multicast [Bal90c]. Below we will report on our experiences in implementing the three communication patterns of Figure 1 in SR.†

Mailbox communication. Despite its large number of features, SR does not directly support message passing through a mailbox. The receiver of a message is fully determined when the message is *sent*. With a mailbox, the destination process is not determined until the message is *accepted* (serviced).

In contrast with the sender of a message, the receiver need *not* specify the other party, so in this sense message passing in SR is asymmetric. This observation also implies a solution to the mailbox problem. We

† The language used for this paper is referred to as SR Version 1.1. The SR designers are currently working on Version 2, in which many of the problems described here will be solved.

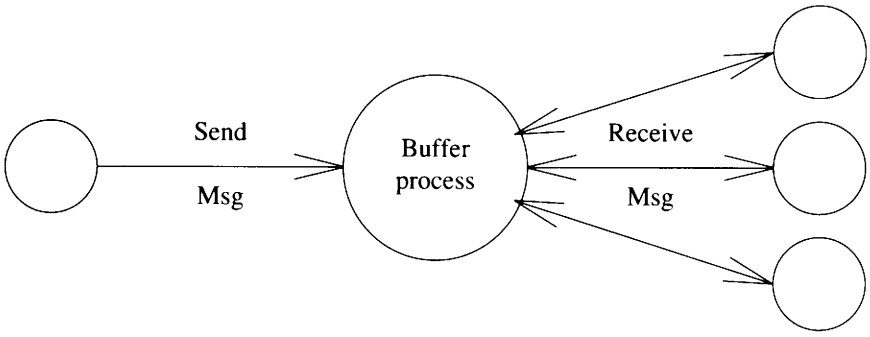


Figure 2: *Simulating message passing through a mailbox in SR*

can simply add an intermediate *buffer* process between the sender and receivers, as shown in Figure 2. The sender sends its message to this buffer process, so the (initial) destination is fixed. The receivers ask the buffer process for a message, whenever they need one.

The buffer process accepts *SendMsg* and *ReceiveMsg* requests one at a time; if the buffer is empty, only *SendMsg* will be accepted. With this scheme, the destination of each message is fixed: it is sent to the buffer process. In this way, the asymmetry of message passing is worked around, at the overhead of implementing an extra process.

In applications where only one process is sending messages, a simpler solution can be used. When the sender wants to send a message, it blocks until a receiver asks for a message. In this case, the receiver can directly fetch a message from the sender, thus eliminating the need for a buffer process.

One-to-many communication. The second communication pattern, one-to-many communication, is supported in SR through a special language construct:

```
co (i := 1 to P)
    send receiver[i].SendMsg(msg)
oc
```

The **co** statement sends a message concurrently to several processes, as specified in the array *receiver*. This approach to multicasting has an important disadvantage, however. If two SR processes concurrently multicast two messages, these messages need not arrive in the same order everywhere. In other words, multicast in SR is not *indivisible*. Applications for which the ordering matters must do it themselves.

Shared data. The third communication pattern, replicated shared data, is not supported by SR. Only processes running on the same node can share variables, other processes must communicate through message passing.

Of course, shared data can be simulated by storing them on one server process and having other processes send messages to this server. As stated before, however, we assume that the read/write ratio of the shared data is very high, in which case it is far more efficient to *replicate* the shared data in the local memories. Each processor keeps its own local copy, which is used for *reading*. Whenever the variable is *written*, all these copies are updated.

Concurrent updates of the shared data will have to be synchronized. For example, if two processes P and Q simultaneously write the shared data, all copies should be updated in a consistent way. It should never be the case that part of the copies are set to P's value while another part

is set to Q's value. With message passing this requirement is difficult to realize, because messages are not globally ordered. In other words, the update messages sent by P and Q may arrive in different orders at different receivers.

The solution we have taken is to send update messages through a central manager process. This process orders the update messages and forwards them in a consistent order to all other processes. These update messages are accepted implicitly by each receiver, which means that the run time system will automatically create a new process for servicing such a message. This is important, since it is not known in advance when the update messages may arrive.

Implementation and Performance

SR has been implemented on a range of multiprocessors (Encore, Sequent Balance, Sequent Symmetry) and distributed systems (homogeneous networks of VAXes, Sun-3s, Sun-4s, and others). The compiler and run time system are available from the University of Arizona.

We have done some initial performance measurements on a Sequent Symmetry with 6 CPUs. Although this machine has a shared memory, SR uses it only for implementing message passing, so the machine is not really used as a multiprocessor.

For the All Pairs Shortest Paths problem, we have measured a speedup of 4.08 (on 6 CPUs). The reason why this speedup is less than linear is the fact that the **co** statement currently is not implemented as a true (physical) multicast. The message is copied once for every receiver. The communication overhead of the ASP program therefore is high, which prevents a linear speedup.

For the Traveling Salesman Problem, we measured a maximum speedup of 5.87. The latter program uses the simplified solution for mailbox communication (i.e., the manager generates only one job at a time and blocks until this job has been accepted).

Conclusions on SR

Since SR provides so many communication primitives, it is a flexible language. SR is also more expressive than most other message passing languages. It can be argued, however, that message passing is a low level of abstraction. As we will see, for several applications other mechanisms than message passing are simpler to use. These higher-level mechanisms are frequently more expressive yet less flexible. In conclusion, SR is reasonably suited for virtually all applications. It is seldom spectacularly good or bad for any application.

4. Emerald

Emerald [Bla87a, Jul88a] is an object-based language, designed at the University of Washington by Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. An object in Emerald encapsulates both static data and an active process. Objects communicate by invoking each other's operations. There can be multiple active invocations within one object, which synchronize through a *monitor*. The remote invocation mechanism is *location transparent*.

Central to Emerald's design is the concept of *object mobility*. An object may migrate from one processor to another, as initiated either by

the programmer or the system. Emerald uses a novel parameter mode, *call-by-move*. This mode has similar semantics as call-by-reference, but additionally moves the object parameter to the node of the invoked object.

Programming Experience

Emerald is reasonably easy to learn. Since it is an object-based language, it treats all entities as objects. Unlike object-*oriented* languages, it does not support inheritance. Nonwithstanding its object-based nature, Emerald contains many constructs also found in procedural languages (e.g., nested scopes, functions, expressions, assignment and control statements). The type system is one of the more important contributions of the language. Although it is not easy to get used to, it is flexible and features static type checking and polymorphism.

Below, we will discuss how mailboxes, one-to-many communication, and shared data can be implemented in Emerald.

Mailbox communication. A message (or operation) in Emerald is always sent to a specific object, so mailbox-style communication is not provided. It is possible, however, to construct a mailbox *object*, which can be accessed by the senders and receivers. Such an object has the following user-defined polymorphic type:

```
type Mailbox
    operation AddMsg[eType]              % Add a message to the mailbox
    operation GetMsg -> [job: eType]     % Fetch a message from the mailbox
end Mailbox
```

The object type is implemented using a queue of messages. To synchronize access to the queue, it is encapsulated in a monitor. The *AddMsg* and *GetMsg* operations are thus executed in a mutually exclusive way. Also, the *GetMsg* operation will block on a condition variable if the queue is empty; this condition variable will be signalled by an invocation of *AddMsg*.

This implementation of mailboxes is roughly similar to the SR version, except that a passive object rather than an active process is used for storing the message queue. Also, the synchronization of the queue operations is entirely different. In the SR version, the buffer process synchronizes the operations by accepting them one at a time and by delaying requests for messages when the buffer is empty. The Emerald version uses a monitor and a condition variable for synchronizing the operations.

One-to-many communication. Emerald does not support any form of one-to-many communication. To send data to multiple objects, a sequential **for** loop has to be used. A subtle problem arises here that does not occur in the other languages. Emerald provides a uniform parameter mechanism: all objects are passed by reference, no matter where the sender and receiver are located. With multicasting, however, each receiver should be given a *copy* of the data, not a remote reference to it. What is needed here is call-by-value semantics, which is not supported in Emerald.

Thus, the sender must copy the data explicitly and pass this copy as call-by-move parameter. A distinct copy must be made for every receiver. So a multicast is simulated as follows in Emerald:

```
for all receivers r do
  r.send[move copy[msg]]
```

Here, *copy* is a user-defined procedure that copies a message.

The Emerald implementation of one-to-many communication is fairly complex. In addition, the solution is far from efficient. Not only does it refrain from using physical multicast, but it also forces the sender to copy the message once for every receiver, which may become a sequential bottleneck.

Shared data. Although Emerald supports a shared namespace for objects, this is not sufficient for implementing replicated shared data. If a shared variable were stored in a single object, nearly all accesses to the variable would require physical communication, including read-only operations. What is needed is a replicated object, which is not provided in Emerald.

The programmer therefore has to replicate data explicitly. A copy of the shared data is kept by each process needing the data. To update these copies, a similar scheme as for SR is used, based on implicitly received messages. The main difference with the SR solution is the usage of a monitor for synchronizing access to the local copy of the shared variable.

Implementation and Performance

A prototype implementation of Emerald exists on networks of VAXes or Sun-3 workstations, connected by an Ethernet. The Emerald system is not yet available to other users. We have not been able to do any meaningful performance measurements on the prototype system.

Conclusions on Emerald

Support for parallel and distributed programming in Emerald is best understood using two levels of abstraction. At the highest level, we have concurrent objects that invoke each other's operations in a synchronous (blocking) way, certainly a nice and simple abstraction. To see what is really going on, we need to look at how invocations are implemented and synchronized. Here, we are at the level of *monitors*. Monitors are well understood, but are harder to program than most other mechanisms discussed in this paper. This clearly shows of in the implementation code: most of our Emerald programs are significantly longer than their counterparts in the other languages.

For parallel programming, Emerald is less flexible than SR. It provides only one form of interprocess communication: synchronous remote procedure calls that are accepted implicitly. The parameter mechanism is consistent (call-by-reference is used throughout), but copying parameters is a problem. In principle, call-by-value parameters could have been allowed for passive objects (not containing a process). This extension would have made the parameter mechanism less uniform, however, and would have created a distinction between active and passive objects.

Emerald probably is more suitable for distributed applications (e.g., electronic mail, name servers) than for parallel applications. For such distributed applications, features like object migration and location independent invocations are more beneficial and the need for copying objects (e.g., electronic mailboxes) will be less.

5. Parlog

We have chosen Parlog [Cla88a, Cla86a, Con89a, Gre87a] as representative for the large class of concurrent logic languages. Parlog has been developed at Imperial College, London, by Keith Clark, Steve Gregory, and their colleagues.

The language is based on AND/OR parallelism and committed-choice nondeterminism. The user can specify the order (parallel or sequential) in which clauses are to be evaluated. For this purpose, sequential and parallel conjunction and disjunction operators can be used.

Programming Experience

The time needed for learning Parlog depends on ones background education in concurrent logic programming. The language itself is quite simple. In addition, there are certain programming idioms one should master, such as streams and objects built with shared logical variables.

Mailbox communication. As in most concurrent logic languages, processes in Parlog can communicate through message streams. Such streams can easily be built out of shared logical variables. Streams, however, have one disadvantage: the receiving end can *scan* over the stream, but it cannot *remove* items from it [Car89a]. Thus, mailbox-type communication cannot be expressed easily with streams.

Instead, we can use similar solutions as for SR, which means either adding a buffer process between the sender and receivers (see Figure 2), or blocking the sender of the message. For our TSP program [Bal90d], we have chosen the latter option. There is only a single sender, which blocks when it wants to send a message. The sender takes a stream of incomplete messages of the form *getmsg(Msg)* as input. These messages are generated by the receivers. After receiving such a message, the sender instantiates the logical variable *Msg* to the next message it wants to send.

One-to-many communication. One-to-many communication is easy to express using shared logical variables. All that is needed is a stream of messages shared among the sender and the receivers. All receivers can scan this stream, thus receiving all the messages.

It depends on the language implementation whether physical multicast is used for this type of one-to-many communication. For example, multicast is used to some extent in the hypercube implementation of Flat Concurrent Prolog [Tay87a]. The Parlog system we have used uses shared memory, which takes away the need for physical multicast.

Our Parlog ASP program uses an even simpler approach to one-to-many communication. Rather than creating a fixed number of long-living processes, it creates a new set of parallel processes for each iteration of the algorithm. The pivot row for the next iteration is passed as parameter to each of these processes. In other words, the program does not send a message to existing processes, but it creates new processes and passes the message as a parameter. This approach only works well because the Parlog system efficiently supports fine-grained parallelism. With the other languages discussed in this paper, the overhead of creating new processes for each iteration would be far too high.

Shared data. Parlog supports shared logical variables, but these variables can be assigned only once. Implementing mutable shared variables in Parlog is much more complicated. We represent such a vari-

able as a stream of values, the last one of which is the current value of the variable. The predicate *current_value* scans the stream until the tail is an unbound variable, and returns the current last element of the stream as output value:

```
mode current_value(Stream?, Value^).  % Stream is input, Value is output

current_value([V|Vs], Value) <- var(Vs): Value = V;        % tail is unbound
current_value([_|Vs], Value) <- current_value(Vs, Value).  % try next element
```

To update the variable, a new value is appended to the end of the stream. A process using the variable must periodically check for new values, by scanning the stream until the end. (This technique is also used by Huntbach [Hun89a]).

An important issue is how often to check the stream. Since scanning streams is expensive, it cannot be done too often. On the other hand, if it is done infrequently, the process will usually have an old value of the shared variable. For branch-and-bound applications like TSP, this means pruning will become less efficient, so more nodes will be searched (the so-called *search overhead*).

This solution is somewhat similar to the SR and Emerald implementations described above. The stream representing the shared variable can be regarded as a stream of *update* messages. An important difference is the way these messages are accepted. In SR and Emerald, a new process is created when a message arrives, which will service the message immediately (i.e., the message is received implicitly). Parlog does not have implicit message receipt, so the receiver must explicitly look for new messages. Since it is not know in advance when update messages may arrive, there is a problem in deciding *when* to look for them.

Implementation and Performance

An interpreter for Parlog has been implemented on several shared-memory multiprocessors (Sequent Balance and Symmetry, Butterfly). A commercially available subset of Parlog, called Strand, has also been implemented on distributed systems (hypercubes, networks). The Parlog system is available from Imperial College.

We have used a 6-CPU Sequent Balance for running some initial performance measurements. This implementation of Parlog relies on the presence of shared memory. Also, the implementation is based on an interpreter and runs on slow processors, so its absolute performance currently is one to two orders of magnitude less than that of the other languages described in this paper. These two issues taken together result in a relative communication overhead that is far less than what would be expected in a production-quality, distributed implementation.

We have measured a speedup of 5.33 for ASP and 4.98 for TSP, using 6 CPUs. The speedup for ASP is fairly high, due to the low communication overhead. For TSP, the speedup is not optimal, because the global bound is not kept up-to-date everywhere. The TSP program therefore suffers from a search overhead.

Conclusions on Parlog

The shared logical variable is at a higher level of abstraction than message passing. For some applications, it is spectacularly expressive. Our Parlog program for ASP, for example, is just as simple as the original *sequential* algorithm. The synchronization of the parallel tasks is done implicitly, using suspension on unbound logical variables. On the negative side, it is not clear whether the program will run efficiently on a realistic large-scale parallel system.

For other applications, shared logical variables are less suitable, but one can then fall back on message passing through streams. This form of message passing has some drawbacks, however, as discussed in [Car89a].

6. Linda

Linda is a set of language primitives developed by David Gelernter and colleagues at Yale University [Ahu86a, Car89b, Car89a]. Linda is based on the Tuple Space model of communication. The Tuple Space is a global memory consisting of tuples (records) that are addressed associatively. Three atomic operations are defined on Tuple Space: **out** adds a tuple to TS; **read** reads a tuple contained in TS; **in** reads a tuple and also deletes it from TS, in one atomic action.

Programming Experience

Of all five languages discussed in this paper, Linda undoubtedly is the simplest one to learn. It adds only a few primitives to an existing base language. Below, we will discuss how these primitives can be used to implement the three communication patterns.

Mailbox communication. The simulation of a mailbox in Linda is simple. A mailbox is represented as a distributed data structure [Car86a] in Tuple Space. To send a message to the mailbox, a new tuple containing the message is added to this data structure. To receive a message, a tuple is retrieved from Tuple Space and its contents are read.

To preserve the ordering of the messages, a sequence number field is added to each message tuple. The tuples are generated and retrieved in the same order. The next sequence number to generate and the sequence number of the next message to accept are also stored in tuples. They are initialized to zero, by the statements:

```
out("head", 0);    # initialize tuple containing index of
                   #   head of queue
out("tail", 0);    # initialize tuple containing index of
                   #   tail of queue
```

To send a message *msg* to a mailbox, the following code is executed:

```
in("tail", ? &tail);      # obtain next sequence number
out("tail", tail + 1);    # put back next sequence number
out("MB", msg, tail);     # put message with sequence
                          #      number in TS
```

The **in** operation blocks until a matching tuple is found. Next, it assigns the formal parameters of the **in** (denoted by a "?") the corresponding values of the tuple. Finally, it deletes the tuple from Tuple Space. All of this is done atomically.

Receiving a message from a mailbox is implemented through the following code:

```
in("head", ? &head);     # first obtain sequence number
out("head", head+1);     # put sequence-number tuple
                         #    back in TS
in("MB", ? &msg, head); # now fetch message with right
                         #    sequence number
```

The tuples can be thought of as forming a distributed *queue* data structure, with pointers (indices) to the head and tail of the queue.

This example clearly illustrates the advantages and disadvantages of Linda. The mailbox implementation is very simple: it requires only a few lines of code. On the other hand, the operations used for accessing the mailbox are fairly low-level. For example, three Tuple Space operations are needed for sending or receiving a single message. It is far from trivial that this code is correct. Also, the implementation must do extensive optimization to make the send/receive operations efficient.

One-to-many communication. In Linda, data can be transferred from one process to all the others by putting the data in Tuple Space, where it can be read by everyone. So, expressing one-to-many communication in Linda is trivial; it just requires a single **out** statement:

```
out(msg);
```

A key question that remains, however, is what *really* happens. For efficiency, it makes considerable difference whether the data are transferred through a real multicast protocol or not.

There are many different implementations of Tuple Space to consider. The S/Net system replicates all tuples everywhere, using the S/Net broadcast capability [Car86b]. The hypercube and Transputer implementations of Linda, on the other hand, hash each tuple onto one specific processor and do not replicate tuples [Bjo89a, Zen90a]. In this case, the data in the message will not be multicast. Each receiver will have to fetch the data itself, using a **read** statement. The communication overhead will thus be linear to the number of receivers. In conclusion, expressing one-to-many communication is Linda is trivial, but the performance will be hard to predict.

Shared data. In theory, a shared variable can be simulated in Linda by storing it in Tuple Space. This solution makes heavy demands on the implementation of Tuple Space, however. If the variable is read very frequently (as is true in TSP), the overhead of reading it must be very low. So, for efficiency each processor should have a local copy of the tuple. Not all Tuple Space implementations have this property. The hypercube and Transputer implementations mentioned above, for example, store each tuple on only a single processor. An additional performance problem is the associative addressing of Tuple Space. Part of this overhead can be optimized away [Car87a], but it is not clear whether it can be eliminated entirely. So, whether or not the above solution is practical, depends on the implementation.

Implementation and Performance

Linda has been implemented on many parallel machines, both with and without shared memory, and has been used for numerous applications [Car89a]. The system is distributed as a commercial product. (The Linda system we have used for our performance measurements is not the most recent one; newer versions of the Linda software may obtain better performance.)

We have used a VME-bus based multiprocessor for some initial performance measurements. For the All Pairs Shortest Paths problem, we have measured a speedup of 7.4 on 8 CPUs. Since the implementation

uses shared memory, the distribution of the pivot rows is efficient. Each new pivot row is put in a tuple in shared memory, where it can be read by all processors.

The Traveling Salesman Problem program obtains a speedup of 7.06 on 8 CPUs. The program stores the global bound in a tuple. In our Linda system, using this tuple for every read access is too expensive. Therefore, each processor also keeps a local copy of the variable. These copies are updated occasionally. So, this implementation is similar to the Parlog implementation, except that the bound is stored in a tuple rather than in a stream. Updating the local copies is relatively cheaper in the Linda version, so it can be done more frequently. As a result, the relative search overhead in the Linda program is less than that of the Parlog version.

Conclusions on Linda

Most of the criticism on Linda in the literature is related to efficiency. The associative addressing and global visibility of the Tuple Space have led many people to believe that Linda cannot be implemented efficiently. However, its implementors have made considerable progress during the past few years in optimizing the performance on several machines. The **in** operation, for example, hardly ever scans the entire Tuple Space, but typically uses hashing or something even more efficient. Just as with virtual memory, however, there will probably always remain cases where the easy-to-program approach will not be optimal. So, the performance of Linda programs may sometimes be hard to predict.

An important decision in Linda is to hide the physical distribution of data from the user. In contrast, Emerald gives the programmer control over the placement of data, by supporting user-initiated object migration. The Linda approach is simpler, but it makes heavier demands on the implementation. Again, the transparent approach will sometimes be less efficient, but it remains to be seen how big the differences in performance are for actual programs.

The concept of distributed data structures is probably one of the most important contributions of Linda. However, the way Linda implements distributed data structures – through a fixed number of operations on Tuple Space – is rather low-level, in our view [Kaa89a].

7. Orca

Orca is a language for implementing parallel applications on distributed systems. Orca was designed at the Vrije Universiteit in Amsterdam [Bal90b, Bal89b, Bal90e, Bal88a].

The programming model of Orca is based on logically shared data. The language hides the physical distribution of the memory and allows processes to share data even if they run on different nodes. In this way, Orca combines the advantages of distributed systems (good price/performance ratio and scalability) and shared-memory multiprocessors (ease of programming).

The entities shared among processes are data objects, which are variables of user-defined abstract data types. These data objects are replicated in the local memories, so each process can directly read its own copy, without doing any communication. The language run time system atomically updates all copies when an object is modified.

This model is similar to that of Distributed Shared Memory (DSM) systems [Li89a]. In Orca, however, the unit of sharing is a logical (user-defined) object rather than a physical (system-defined) page, which has many advantages [Bal90b].

Programming Experience

Orca is a new language rather than an extension to an existing sequential language. An important disadvantage of extending a base language is the difficulty of implementing pointers and global variables on systems lacking shared memory. These problems can more easily be avoided if the language is designed from scratch. Orca, for example, supports first-class *graph* variables rather than pointers. Unlike pointers, graphs can freely be moved or copied from one machine to another. Of course, this approach also implies that programmers have to learn a new language. The design of Orca has been kept as simple as possible, so this disadvantage should not be overestimated.

Mailbox communication. A mailbox can be implemented in Orca in a similar way as in Emerald, by using a shared mailbox object. The specification of a generic abstract data type *Mailbox* in Orca is shown below:

```
generic (type T)
object specification GenericMailbox;
    operation AddMsg(Msg: T);
    operation GetMsg(): T;
end generic;
```

The implementation of the mailbox is simpler than the one in Emerald, because operations in Orca are indivisible. In other words, mutual exclusion synchronization is done automatically in Orca, whereas Emerald requires the usage of a monitor construct for this purpose. Also, Orca provides a powerful mechanism for condition synchronization (based on guarded commands), so blocking the receivers when the mailbox is empty is easy to express.

One-to-many communication. Orca's shared data-objects can be used for expressing one-to-many communication. If one process applies a write-operation to an object, all other processes sharing the object can observe the effects. Our ASP program in Orca, for example, uses an object-type *RowCollection*, with the following operations:

```
object specification RowCollection;
    type RowType = array[integer] of integer;

    operation AddRow(iter: integer; R: RowType);
        # Add the row for the given iteration number
    operation AwaitRow(iter: integer): RowType;
        # Wait until the row for the given iteration is
        # available, then return it.
end;
```

The process that wants to send the pivot row applies the operation *AddRow* to the object. The run time system will then update all copies of this object by multicasting the operation [Bal89b]. A process requiring the pivot row invokes the operation *AwaitRow*, which blocks until the requested row has been added to the object and then returns this row. The latter operation is done locally, without needing any communication. So, the Orca solution is efficient, since it uses physical multicasting, if available.

Shared data. Orca has the support for logically shared data as a design goal, so it is no surprise that communication through shared data is

easy to express in this language. The shared variable is put in a data object shared among all processes. The run time system automatically replicates the object in the local memories, so processes can directly read the value. Whenever the object is changed, all copies are updated immediately, by broadcasting the new value. Moreover, atomicity of the operations is already guaranteed by the language. This solution is both simple and efficient. The only overhead in reading the value is that of a local operation invocation. When the variable is changed, its new value is broadcast to all processors containing a copy.

Implementation and Performance

Orca has been implemented on top of Amoeba [Tan90a] as well as on a collection of MC68030s connected through an Ethernet. The latter implementation uses the physical multicast capability of the Ethernet. The Orca implementation is being distributed as part of the Amoeba system.

We have done many performance measurements on these systems, as described in detail elsewhere [Bal90b]. Here, we will present some recent results for the multicast system, using 16 CPUs.

The measured speedup for the All Pairs Shortest Paths problem on 16 CPUs is 15.9. This high speedup is mainly due to the efficient broadcast protocol, which is used for transmitting the pivot rows. For the Traveling Salesman Problem, the speedup on 16 CPUs is 14.44. Since all copies of the global bound are updated immediately, the search overhead is low.

Conclusions on Orca

Orca is *not* an object-based language; it merely provides abstract data types. It supports both active processes and passive data-objects. Since objects in Orca are purely passive, they can be replicated, which is a very important goal in the implementation.

An important difference with Linda is the support for user-defined, high-level operations on shared data [Kaa89a]. Linda only provides a fixed number of built-in operations on tuples, but Orca allows programmers to construct their own atomic operations. Unlike Linda, Orca uses direct rather than associative addressing of shared data, and thus avoids any problems with associative addressing.

For some applications, Orca has important advantages over other languages. Programs that need logically shared data are easy to implement in Orca and are efficient. Orca also is one of the few languages that uses physical broadcasting in its implementation. As we have seen, for ASP this is of critical importance. On the other hand, there also are cases where the model is less efficient, for example when plain point-to-point message passing is required.

8. Discussion

In the previous three sections we have looked at how the five languages deal with three example communication patterns. The results of this study are summarized in Table 2. Below, we will compare the approaches taken for the different languages.

For communication through mailboxes, there are three different solutions. For Linda, we store a mailbox as a distributed data structure in

	Mailboxes	One-to-many communication	Replicated shared data
SR	Buffer process	Concurrent send	Messages with implicit receive
Emerald	Shared-object message queue	Point-to-point messages	Messages with implicit receive
Parlog	Buffer process	Shared stream (or solution with fine-grained parallelism)	Messages with explicit receive
Linda	Distr. data structure message queue	Shared data	Shared tuple (or m.p. with explicit receive)
Orca	Shared-object message queue	Shared data	Distributed shared memory

Table 2: *Summary of the solutions taken for all 5 languages to the 3 communication patterns*

Tuple Space. This solution requires only a few lines of code. For Emerald and Orca, a mailbox is represented as an abstract object, with operations to send and receive messages. This approach requires more code, especially for synchronizing access to the mailbox. On the other hand, the abstract operations on a mailbox object are higher level than the Linda operations on tuples. The third solution, used for SR and Parlog, is to add an extra buffer process between the sender and receivers.

For one-to-many communication, Parlog, Linda, and Orca provide the simplest solutions, all based on shared data. SR has a concurrent-send primitive built in, but it does not make any guarantees about the order in which messages are delivered. Emerald has no provision for one-to-many communication, so it must be simulated with multiple point-to-point messages, which are sent sequentially. An important issue is how one-to-many communication is implemented: as a physical multicast or not. Most language implementations do not use multicast, Orca and Linda being two notable exceptions.

The third communication pattern, replicated shared data, is simple to express in Orca and Linda, since these languages provide logically shared data. For Linda, the performance of the resulting programs is hard to predict, because many different strategies are used for distributing tuples. Orca, on the other hand, always tries to replicate shared objects wherever they are needed. For the other languages, we simulate shared data through message passing. Here, the ability to accept messages implicitly (i.e., by a newly created process) is very important. SR and Emerald both provide this facility. Parlog uses only explicit message receipt, which makes efficient updating of the copies of shared data harder.

Acknowledgements

The work on SR and Emerald was done while the author was visiting the University of Arizona, Department of Computer Science, Tucson, AZ. The work on Parlog was done while he was at Imperial College, Department of Computing, London. The author is grateful to both departments for receiving him as an academic visitor. Also, he would like to thank Nick Carriero, Greg Andrews, Dave Bakken, Gregg

Townsend, Mike Coffin, Norman Hutchinson, Keith Clark, Jim Crammond, and Andrew Davison for the discussions on their languages. The work on Linda has been done in cooperation with Frans Kaashoek. Erik Baalbergen, Arnold Geels, Frans Kaashoek, and Andy Tanenbaum provided useful comments on an earlier version of the paper.

References

[Ahu86a] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer* **19**(8), pp. 26-34 (Aug. 1986).

[And83a] G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys* **15**(1), pp. 3-43 (March 1983).

[And86a] G. R. Andrews and R. A. Olsson, "The Evolution of the SR Programming Language," *Distributed Computing* **1**, pp. 133-149 (July 1986).

[And88a] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, "An Overview of the SR Language and Implementation," *ACM Trans. Program. Lang. Syst.* **10**(1), pp. 51-86 (January 1988).

[And89a] G. R. Andrews, "Paradigms for Process Interaction in Distributed Programs," TR 89-24, University of Arizona, Tucson, AZ (October 1989). (accepted for publication in ACM Computing Surveys)

[Bal88a] H. E. Bal and A. S. Tanenbaum, "Distributed Programming with Shared Data," *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, Miami, FL, pp. 82-91 (October 1988).

[Bal89a] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys* **21**(3), pp. 261-322 (September 1989).

[Bal89b] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "A Distributed Implementation of the Shared Data-object Model," *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL., pp. 1-19 (October 1989).

[Bal90c] H. E. Bal, "An Evaluation of the SR Language Design," IR-219, Vrije Universiteit, Amsterdam, The Netherlands (August 1990).

[Bal90d] H. E. Bal, "Heuristic Search in PARLOG using Replicated Worker Style Parallelism," IR-229, Vrije Universiteit, Amsterdam, The Netherlands (November 1990).

[Bal90e] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Experience with Distributed Programming in Orca," *Proc. IEEE CS 1990 Int. Conf. on Computer Languages*, New Orleans, LA, pp. 79-89 (March 1990).

[Bal90a] H. E. Bal, "Fault-tolerant Parallel Programming in Argus," IR-214, Vrije Universiteit, Amsterdam, The Netherlands (May 1990).

[Bal90b] H.E. Bal, *Programming Distributed Systems*, Silicon Press, Summit, NJ (1990).

[Bjo89a] R. Bjornson, N. Carriero, and D. Gelernter, "The Implementation and Performance of Hypercube Linda," Report RR-690, Yale University, New Haven, CT (March 1989).

[Bla87a] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Trans. Softw. Eng.* **SE-13**(1), pp. 65-76 (January 1987).

[Car86a] N. Carriero, D. Gelernter, and J. Leichter, "Distributed Data Structures in Linda," *Proc. 13th ACM Symp. Princ. Progr. Lang.*, St. Petersburg, FL, pp. 236-242 (January 1986).

[Car86b] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM Trans. Comp. Syst.* **4**(2), pp. 110-129 (May 1986).

[Car87a] N. Carriero, "The Implementation of Tuple Space Machines," Research Report 567 (Ph.D. dissertation), Yale University, New Haven, CT (December 1987).

[Car89b] N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Comp. Surveys* **21**(3), pp. 323-357 (September 1989).

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

[Cla86a] K. L. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," *ACM Trans. Program. Lang. Syst.* **8**(1), pp. 1-49 (January 1986).

[Cla88a] K. L. Clark, "PARLOG and Its Applications," *IEEE Trans. Softw. Eng.* **SE-14**(12), pp. 1792-1804 (December 1988).

[Con89a] T. Conlon, *Programming in PARLOG*, Addison-Wesley, Wokingham, England (1989).

[Gre87a] S. Gregory, *Parallel Logic Programming in PARLOG*, Addison-Wesley, Wokingham, England (1987).

[Hun89a] M. Huntbach, *Combinatorial Search in PARLOG Using Speculative Computation*, Imperial College, London (May 1989).

[Jul88a] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Trans. Comp. Syst.* **6**(1), pp. 109-133 (February 1988).

[Kaa89a] M. F. Kaashoek, H. E. Bal, and A. S. Tanenbaum, "Experience with the Distributed Data Structure Paradigm in Linda," *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL., pp. 175-191 (October 1989).

[Kah89a] K. M. Kahn and M. S. Miller, "Technical Correspondence on ''Linda in Context'','' *Comm. ACM* **32**(10), pp. 1253-1255 (October 1989).

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

[Lis88a] B. Liskov, "Distributed Programming in Argus," *Commun. ACM* **31**(3), pp. 300-312 (March 1988).

[Sha89a] E. Shapiro, "Technical Correspondence on ''Linda in Context'','' *Comm. ACM* **32**(10), pp. 1244-1249 (October 1989).

[Tan90a] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, A. J. Jansen, and G. van Rossum, "Experiences with the Amoeba Distributed Operating System," *Comm. ACM* **33**(12), pp. 46-63 (December 1990).

[Tay87a] S. Taylor, S. Safra, and E. Shapiro, "A Parallel Implementation of Flat Concurrent Prolog," *Int. J. of Parallel Programming* **15**(3), pp. 245-275 (1987).

[Zen90a] S.E. Zenith, "Linda Coordination Language; Subsystem Kernel Architecture (on Transputers)," RR-794, Yale University, New Haven, CT (May 1990).

Linking a Stub Generator (AIL) to a Prototyping Language (Python)

Guido van Rossum

CWI, The Netherlands
guido@cwi.nl

Jelke de Boer

HIO Enschede, The Netherlands

Abstract

This paper describes how two tools that were developed quite independently gained in power by a well-designed connection between them. The tools are Python, an interpreted prototyping language, and AIL, a Remote Procedure Call stub generator. The context is Amoeba, a well-known distributed operating system developed jointly by the Free University and CWI in Amsterdam.

As a consequence of their integration, both tools have profited: Python gained usability when used with Amoeba – for which it was not specifically developed – and AIL users now have a powerful interactive tool to test and experiment with new client/server interfaces.

1. Introduction

Remote Procedure Call (RPC) interfaces, used in distributed systems like Amoeba [Mul90a, Tan90a], have a much more concrete character than local procedure call interfaces in traditional systems. Because clients and servers may run on different machines, with possibly different word size, byte order, etc., much care is needed to describe interfaces exactly and to implement them in such a way that they continue to work when a client or server is moved to a different machine. Since machines may fail independently, error handling must also be treated more carefully.

A common approach to such problems is to use a *stub generator*. This is a program that takes an interface description and transforms it into functions that must be compiled and linked with client and server applications. These functions are called by the application code to take care of details of interfacing to the system's RPC layer, to implement transformations between data representations of different machines, to check for errors, etc. They are called "stubs" because they don't actually perform the action that they are called for but only relay the parameters to the server [Bir84a].

Amoeba's stub generator is called AIL, which stands for Amoeba Interface Language [Ros90a]. The first version of AIL generated only C functions, but an explicit goal of AIL's design was *retargetability*: it should be possible to add back-ends that generate stubs for different languages from the same interface descriptions. Moreover, the stubs generated by different back-ends must be *interoperable*: a client written in Modula-3, say, should be able to use a server written in C, and vice versa.

This interoperability is the key to the success of the marriage between AIL and Python. Python is a versatile interpreted language developed by the first author [Ros91a]. Originally intended as an alternative for the kind of odd jobs that are traditionally solved by a mixture of shell scripts, manually given shell commands, and an occasional ad hoc C program, Python has evolved into a general interactive prototyping language. It has been applied to a wide range of problems, from replacements for large shell scripts to fancy graphics demos and complete window-based applications.

One of Python's strengths is the ability for the user to write some code and immediately test it: no compilation or linking is necessary. Interactive performance is further enhanced by Python's concise, clear syntax, its very-high-level data types, and its lack of declarations (which is compensated by extensive run-time type checking). All this makes programming in Python feel like a leisure trip compared to the hard work involved in writing and debugging all but the smallest C programs.

It should be clear by now that Python will be the ideal tool to test servers and their interfaces. Especially during the development of a complex server, one often needs to generate test requests on an ad hoc basis, to answer questions like "what happens if request X arrives when the server is in state Y", to test the behavior of the server with requests that touch its limitations, to check server responses to all sorts of wrong requests, etc. Python's ability to immediately execute "improvised" code makes it a much better tool for this situation than C.

The link to AIL extends Python with the necessary functionality to connect to arbitrary servers, making the server testbed sketched above a reality. Python's high-level data types, general programming features, and system interface ensure that it has all the power and flexibility needed for the job.

1.1. Overview of this Paper

The rest of this paper contains three major sections and a conclusion. First an overview of the Python programming language is given. Next comes a short description of AIL, together with some relevant details about Amoeba. Finally, the design and construction of the link between Python and AIL is described in much detail. The conclusion looks back at the work and points out weaknesses and strengths of Python and AIL that were discovered in the process.

2. An Overview of Python

Pythons [†] owes much to ABC [Geu90a], a language developed at CWI as a programming language for non-expert computer users. Python

† Named after the funny TV show, not the nasty reptile.

borrows freely from ABC's syntax and data types, but adds modules, exceptions and classes, extensibility, and the ability to call system functions. The concepts of modules, exceptions and (to some extent) classes are influenced strongly by their occurrence in Modula-3 [Car89a].

Perhaps the best introduction to Python is a short example. The following is a complete Python program to list the contents of a UNIX directory.

```
import sys, posix

def ls(dirname):    # Print sorted directory contents
    names = posix.listdir(dirname)
    names.sort()
    for name in names:
        if name[0] <> '.': print name

ls(sys.argv[1])
```

The largest part of this program, in the middle starting with `def`, is a function definition. It defines a function named `ls` with a single parameter called `dirname`. (Comments in Python start with '#' and extend to the end of the line.) The function body is indented; Python uses indentation for statement grouping instead of braces or begin/end keywords. This is shorter to type and avoids frustrating mismatches between the perception of grouping by the user and the parser. Python accepts one statement per line; long lines may be broken in pieces using the standard backslash convention. If the body of a compound statement is a single, simple statement, it may be placed on the same line as the head.

The first statement of the function body calls the function `listdir` defined in the module `posix`. This function returns a list of strings representing the contents of the directory name passed as a string argument, here the argument `dirname`. If `dirname` were not a valid directory name, or perhaps not even a string, `listdir` would raise an exception and the next statement would never be reached. (Exceptions can be caught in Python; see below.) Assuming `listdir` returns normally, its result is assigned to the local variable `names`.

The second statement calls the method `sort` of the variable `names`. This method is defined for all lists in Python and does the obvious thing: the elements of the list are reordered according to their natural ordering relationship. Since in our example the list contains strings, they are sorted in ascending ASCII order.

The last two lines of the function contain a loop that prints all elements of the list whose first character isn't a period. In each iteration, the `for` statement assigns an element of the list to the local variable `name`. The `print` statement is intended for simple-minded output; more elaborate formatting is possible with Python's string handling functions.

The other two parts of the program are easily explained. The first line is an `import` statement that tells the interpreter to import the modules `sys` and `posix`. As it happens these are both built into the interpreter. Importing a module (built-in or otherwise) only makes the module name available in the current scope; functions and data defined in the module are accessed through the dot notation as in `posix.listdir`. The scope rules of Python are such that the imported module name `posix` is also available in the function `ls` (this will be discussed in more detail below).

Finally, the last line of the program calls the `ls` function with a definite argument. It must be last since Python objects must be defined before they can be used; in particular, the function `ls` must be defined before it can be called. The argument to `ls` is `sys.argv[1]`, which happens to be the Python equivalent of `$1` in a shell script or `argv[1]` in a C program.

2.1. Python Data Types†

Python's syntax may not have big surprises (which is exactly as it should be), but its data types are quite different from what is found in languages like C, Ada or Modula-3. All data types in Python, even integers, are "objects". All objects participate in a common garbage collection scheme, currently implemented using reference counting. Assignment is cheap, independent of object size and type: only a pointer to the assigned object is stored in the assigned-to variable. No type check is performed on assignment; only specific operations like addition test for particular operand types.

The basic object types in Python are numbers, strings, tuples, lists and dictionaries. Some other object types are open files, functions, modules, classes, and class instances; even types themselves are represented as objects. Extension modules written in C can define additional object types; examples are objects representing windows and Amoeba capabilities. Finally, the implementation itself makes heavy use of objects, and defines some private object types that aren't normally visible to the user.

There is no explicit pointer type in Python.

Numbers, both integers and floating point, are pretty straightforward. The notation for numeric constants is the same as in C, including octal and hexadecimal integers; precision is the same as `long` or `double` in C. All standard operations are supported, except that integers and floating point numbers cannot be mixed (yet).

Strings are "primitive" objects just like numbers. String constants are written between single quotes, using similar escape sequences as in C. Operations are built into the language to concatenate and to replicate strings, to extract substrings, etc. There is no limit to the length of the strings created by a program. There is no separate character data type; strings of length one do nicely.

Tuples are a way to "pack" small amounts of heterogeneous data together and carry them around as a unit. Unlike structure members in C, tuple items are nameless. Packing and unpacking assignments allow access to the items, for example:

```
x = 'Hi', (1, 2), 'World'    # x is a 3-item tuple,
                             # its middle item is (1, 2)
p, q, r = x                  # unpack x into p, q and r
a, b = q                     # unpack q into a and b
```

A combination of packing and unpacking assignment can be used as parallel assignment, and is idiom for permutations, e.g.:

```
p, q = q, p                  # swap without temporary
a, b, c = b, c, a            # cyclic permutation
```

Tuples are also used for function argument lists if there is more than one argument. A tuple object, once created, cannot be modified; but it

† This and the following subsections describe Python in quite a lot of detail. If you are more interested in AIL, Amoeba and how they are linked with Python, you can skip to section 3 now.

is easy enough to unpack it and create a new, modified tuple from the unpacked items and assign this to the variable that held the original tuple object (which will then be garbage-collected).

Lists are array-like objects. List items may be arbitrary objects and can be accessed and changed using standard subscription notation. Lists support item insertion and deletion, and can therefore be used as queues, stacks etc.; there is no limit to their size.

Strings, tuples and lists together are *sequence* types; these share a common notation for generic operations on sequences like subscription, concatenation, slicing (taking subsequences) and membership tests. As in C, subscripts start at 0.

Dictionaries are "mappings" from one domain to another. The basic operations on dictionaries are item insertion, extraction and deletion, using subscript notation with the key as subscript. The current implementation allows only strings in the key domain, but this is a relict of the extensive use of dictionary objects in the Python interpreter for all kinds of symbol tables.

2.2. Statements

Python knows various kinds of simple statements, such as assignments and `print` statements, and several kinds of compound statements, like `if` and `for` statements. Formally, function definitions and `import` statements are also statements, and there are no restrictions on the ordering of statements or their nesting: `import` may be used inside a function, functions may be defined conditionally using an `if` statement, etc. The effect of a declaration-like statement takes place only when it is executed.

All statements except assignments and expression statements begin with a keyword; this makes the language easy to parse. Here follows an overview of the most common statement forms in Python.

An *assignment* has the general form

variable = variable = ... = variable = expression

It assigns the value of the expression to all listed variables. (As shown in the section on tuples, variables and expressions can in fact be comma-separated lists.) The assignment operator is not an expression operator; there are no horrible things in Python like

```
while (p = p->next) { ... }
```

Expression syntax is mostly straightforward and will not be explained in detail here.

An *expression statement* is just an expression on a line by itself. This writes the value of the expression to standard output, in a suitably unambiguous way, unless it is a procedure call that returns no value. Writing the value is useful when Python is used in "calculator mode", and forces the programmer to do something with function results.

The `if` statement allows conditional execution. It has optional `elif` and `else` parts; a construct like `if ... elif ... elif ... elif ... else` can be used to compensate for the absence of a *switch* or *case* statement.

Looping is done with `while` and `for` statements. The latter iterates over the elements of a "sequence" (see the discussion of data types below). It is possible to terminate a loop with a `break` statement. There is no *continue* statement (it smells too much of *goto*). Both looping statements have an optional `else` clause which is executed

after the loop is terminated normally, but skipped when it is terminated by `break`. This can be handy for searches, to handle the case that the item is not found.

Python's *exception* mechanism is modelled after that of Modula-3. Exceptions are raised by the interpreter when an illegal operation is tried. It is also possible to explicitly raise an exception with the `raise` statement:

`raise` *expression, expression*

The first expression identifies which exception should be raised; there are several built-in exceptions and the user may define additional ones. The second, optional expression is passed to the handler, e.g., as a detailed error message.

Exceptions may be handled (caught) with the `try` statement, which has the following general form:

```
try: block
except expression, variable: block
except expression, variable: block
...
except: block
```

When an exception is raised during execution of the first block, a search for an exception handler starts. The first `except` clause whose *expression* matches the exception is executed. A handler without expression serves as a "catch-all". If there is no match, the search for a handler continues with outer `try` statements; if no match is found on the entire invocation stack, an error message and stack trace are printed, and the program is terminated (interactively, the interpreter returns to its main loop).

Other common statement forms, which we have already encountered, are function definitions, `import` statements and `print` statements. There is also a `del` statement to delete one or more variables, and a `return` statement to return from a function or procedure. Finally, the `pass` statement serves as a no-op.

2.3. Execution Model

A Python program is executed in a straightforward manner. At any point during execution, two "symbol tables" are used to hold variables: the local and the global symbol table. When a variable is assigned to, an entry for it is always made in the local symbol table. When a variable's value is needed, it is searched first in the local symbol table, then in the global one. Pointers to the current local and global symbol table are kept on a stack, which is pushed or popped whenever a scope is entered or left.

The term "variable" in this context refers to any name: functions and imported modules are also searched in this way (and usually found in the global symbol table). Names of built-in functions and exceptions (though not keywords like `if`) are found in a built-in symbol table, which is searched after the global symbol table.

The local symbol table is bound to a particular function invocation, giving standard semantics for local variables. It is initialized with the values of the call's arguments.

The global symbol table used by a particular function call is that of the module where the function was defined. Names in a module's symbol table survive until the end of the program. This approximates the

semantics of file-static global variables in C or module variables in Modula-3.

The statements in a module (written in Python) are executed when the module is first imported; this ensures modules are always properly initialized before used. For statements executed directly in the module, the global symbol table is also the local symbol table, so assignments here have a global effect. The program (script) itself is a nameless module which is executed first. Since `import` statements have to be executed like all other statements, the initialization order of the modules used in a program is well-defined.

The "attribute" notation *m.name*, where *m* is a module, accesses the symbol *name* in that module's symbol table. It can be assigned to as well. This is in fact a special case of the construct *x.name* where *x* denotes an arbitrary object; the type of *x* determines how this is to be interpreted, and what assignment to it means.

For instance, when `a` is a list object, `a.append` yields a built-in method object which, when called, appends an item to `a`. (If `a` and `b` are distinct list objects, `a.append` and `b.append` are distinguishable method objects.) Normally, in statements like `a.append(x)`, the method object `a.append` is called and immediately discarded, but this is a matter of convention.

List objects refuse assignment to attributes. Some objects, like numbers and strings, have no attributes at all. Like all type checking in Python, the meaning of an attribute is determined at run-time – when the parser sees *x.name*, it has no idea of the type of *x*. Note that *x* here does not have to be a variable – it can be an arbitrary (perhaps parenthesized) expression.

Given the attribute notation, one is tempted to use it to replace all standard operations. Yet, Python has kept a small repertoire of built-in functions like `len()` and `abs()`. The reason is that in some cases the function notation is more familiar than the method notation; just like programs would become less readable if all infix operators were replaced by function calls, they would become less readable if all function calls had to be replaced by method calls (and vice versa!).

The choice whether to make something a built-in function or a method is a matter of taste. For arithmetic and string operations, function notation is preferred, since frequently the argument to such an operation is an expression using infix notation, as in `abs(a+b)`; this definitely looks better than `(a+b).abs()`. The choice between make something a built-in function or a function defined in a built-in method (requiring `import`) is similarly guided by intuition; all in all, only functions needed by "general" programming techniques were made built-in functions.

2.4. Classes

Python has a class mechanism distinct from the object-orientation already explained. A class in Python is not much more than a collection of methods and a way to create class instances. Class methods are ordinary functions whose first parameter is the class instance; they are called using the method notation.

For instance, a class can be defined as follows:

```
class Foo():
   def meth1(self, arg): ...
   def meth2(self): ...
```

A class instance is created by `x = Foo()` and its methods can be called thus:

```
x.meth1('Hi There!')
x.meth2()
```

The functions used as methods are also available as (read-only) attributes of the class object, and the above method calls could also have been written as follows:

```
Foo.meth1(x, 'Hi There!')
Foo.meth2(x)
```

Class methods can store instance data by assigning to instance data attributes, e.g.:

```
self.size = 100
self.title = 'Dear John'
```

Data attributes do not have to be declared; as with local variables, they spring into existence when assigned to. It is a matter of discretion to avoid name conflicts with method names. This facility is also available to class users; instances of a method-less class can be used as records with named fields.

There is no built-in mechanism for instance initialization; classes conventionally provide an `init()` method which initializes the instance and returns the instance, so the user can write

```
x = Foo().init(...)
```

Any user-defined class can be used as a base class to derive other classes. However, built-in types like lists cannot be used as base classes. (Incidentally, the same is true for C++ or Modula-3.) Multiple inheritance is supported: a derived class can have multiple base classes. A class may override any method of its base classes. Instance methods are first searched in the method list of their class, and then, recursively, in the method lists of their base classes (depth-first). Initialization methods of derived classes should explicitly call the initialization methods of their base classes.

2.5. The Python Library

Python comes with an extensive library, structured as a number of modules. A few modules are built into the interpreter; these generally provide access to system libraries implemented in C such as mathematical functions or operating system calls. Two built-in modules provide access to internals of the interpreter and its environment. Even abusing these internals will at most cause an exception in the Python program; the interpreter cannot dump core because of errors in Python code.†

Most modules however are written in Python and distributed with the interpreter; they provide general programming tools like string operations and random number generators, provide more convenient interfaces to some built-in modules, or provide specialized services like a *getopt*-style command line option processor for stand-alone scripts.

There are also some modules written in Python that dig deep in the internals of the interpreter; one module can browse the stack backtrace when an unhandled exception has occurred, and one module can disassemble the internal representation of Python code.

† At least in theory...

2.6. Extensibility

It is easy to add new built-in modules written in C to the Python interpreter. Extensions appear to the Python user as built-in modules. Using a built-in module is no different than using a module written in Python, but obviously the author of a built-in module can do things that cannot be implemented purely in Python.

In particular, built-in modules can contain Python-callable functions that call functions from particular system libraries ("wrapper functions"), and they can define new object types. In general, if a built-in module defines a new object type, it should also provide at least one function that creates such objects. Attributes of such object types are also implemented in C; they can return data associated with the object or methods, implemented as C functions.

For instance, an extension was created for Amoeba: it provides wrapper functions for the basic Amoeba name server functions, and defines a "capability" object type, whose methods are file server operations. Another extension is a built-in module is called `posix`; it provides mostly wrappers around UNIX system calls. Extension modules also provide access to two different windowing/graphics interfaces: STDWIN [Ros88a] (which connects to X11 on UNIX and to the Mac Toolbox on the Macintosh), and the Graphics Library (GL) for Silicon Graphics machines.

Any function in an extension module is supposed to type-check its arguments; the interpreter contains convenience functions to facilitate extracting C values from arguments and type-checking them at the same time. Returning values is also painless, using standard functions to create Python objects from C values.

2.7. Statistics and Performance

The source code of the Python interpreter currently consists of about 100 files, totalling about 390 kbytes or 19000 lines, not counting the LL parser generator that comes with it. Included in these numbers are several large optional built-in modules, but no generated files. The files used for the Stubcode interpreter (the link to AIL) comprise less than five percent of the total.

On a Silicon Graphics 4D/25 (a MIPS R3000 based system) the stripped binary is 709 kbytes in size, of which 624 kbytes are text. This includes almost all optional built-in modules. It also includes the STDWIN standard window interface library and the GNU Readline library; SGI's Graphics Library (GL) and X11 are shared libraries and thus not counted. The stripped binary size of the minimal portable version of the interpreter (which excludes all optional modules) is much smaller; this size is given for various architectures in Table 1. This table also shows the range of architectures to which Python has been ported to date.

A small performance test was done: how long does it take to compute `fac(5)`, defined by the following function:

```
def fac(n):
    if n = 1: return 1
    else: return n * fac(n-1)
```

This particular test was chosen since timing figures for it are also given by Ousterhout for Tcl [Ous90a]. It is not representative for Python's performance when its high level data types are used, but gives some insight in the interpreter's overhead for basic operations like function

Machine	Time to compute `fac(5)` (ms.)	Time for `fac(5)` in Tcl (ms.)	Stripped binary size (kbytes)
SGI 4D/25	1.4	-	176
SparcStation 1+	2.7	-	208
Sun 3/260	7.2	-	176
Sun 3/75	-	11.25	176
DECstation 3100	1.3	3.63	168
VAXstation 2000	11.8	-	120
Harris HCX-7 (tahoe)	3.9	-	112
Macintosh Plus	53	-	162

Table 1: *Python Statistics*

calls, arithmetic, and flow control. The test was performed using Python's built-in `time` module; results are shown in Table 1, with Ousterhout's figures for comparison.

3. A Short Description of AIL and Amoeba

An RPC stub generator takes an interface description as input. The designer of a stub generator has at least two choices for the input language: use a suitably restricted version of the target language, or design a new language. The first solution was chosen, for instance, by the designers of Flume, the stub generator for the Topaz distributed operating system designed at DEC SRC [Bir87a, McJ87a].

Flume's one and only target language is Modula-2+ (the predecessor of Modula-3). Modula-2+, like Modula-N for any N, has an interface syntax that is well suited as a stub generator input language: an interface module declares the functions that are "exported" by a module implementation, with their parameter and return types, plus the types and constants used for the parameters. Therefore, the input to Flume is simply a Modula-2+ interface module. But even in this ideal situation, an RPC stub generator needs to know things about functions that are not stated explicitly in the interface module: for instance, the transfer direction of VAR parameters (IN, OUT or both) is not given. Flume solves this and other problems by a mixture of directives hidden in comments and a convention for the names of objects. Thus, one could say that the designers of Flume really designed a new language, even though it looks remarkably like their target language.

Amoeba uses C as its primary programming language. C function declarations (at least in pre-Standard C) don't specify the types of the parameters, let alone their transfer direction. Using this as input for a stub generator would require almost all information for the stub generator to be hidden inside comments, which would require a rather contorted scanner. Therefore we decided to design the input syntax for Amoeba's stub generator "from scratch". This gave us the liberty to invent proper syntax not only for the transfer direction of parameters, but also for variable-length arrays.

On the other hand we decided not to abuse our freedom, and borrowed as much from C as we could. For instance, AIL runs its input through the C preprocessor, which gets us macros, include files and conditional compilation for free. AIL's type declaration syntax is a superset of C's, so the user can include C header files to use the types declared there as function parameter types – which are declared using function proto-

types as in C++ or Standard C. It should be clear by now that AIL's lexical conventions are also identical to C's. The same is true for its expression syntax.

Where does AIL differ from C, then? Function declarations in AIL are grouped in *classes*. Classes in AIL are mostly intended as a grouping mechanism: all functions implemented by a server are grouped together in a class. Inheritance is used to form new groups by adding elements to existing groups; multiple inheritance is supported to join groups together. Classes can also contain constant and type definitions, and one form of output that AIL can generate is a header file for use by C programmers who wish to use functions from a particular AIL class. Here are a few (unrealistically) simple class definitions:

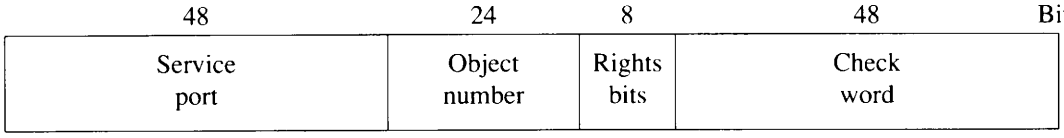
```
#include <amoeba.h>     /* Defines capability, etc. */

class standard_ops [1000 .. 1999] {
    /* Standard operations supported by most interfaces */
    std_info(*, out char buf[size:100], out int size);
    std_destroy(*);
};
class tty [2000 .. 2099] {
    inherit standard_ops;
    const TTY_MAXBUF = 1000;
    tty_write(*, char buf[size:TTY_MAXBUF], int size);
    tty_read(*, out char buf[size:TTY_MAXBUF], out int size);
};
class window [2100 .. 2199] {
    inherit standard_ops;
    win_create(*, int x, int y, int width, int height,
                  out capability win_cap);
    win_reconfigure(*, int x, int y, int width, int height);
};

class tty_emulator [2200 .. 2299] {
    /* Demonstrate multiple inheritance */
    inherit tty, window;
};
```

(The bracketed number ranges after the class names are used to assign request codes; for various technical reasons AIL cannot be trusted to choose request codes itself.)

Note that the power of AIL classes doesn't go as far as C++. AIL classes cannot have data members, and there is no mechanism for a server that implements a derived class to inherit the implementation of the base class – other than copying the source code. The syntax for class definitions and inheritance is also different.

3.1. Amoeba

The smell of "object-orientedness" that the use of classes in AIL creates matches nicely with Amoeba's object-oriented approach to RPC. In Amoeba, almost all operating system entities (files, directories, processes, devices etc.) are implemented as *objects*. Objects are managed by *services* and represented by *capabilities*. A capability gives its holder access to the object it represents. Capabilities are protected cryptographically against forgery and can thus be kept in user space. A capability is a 128-bit binary string, subdivided as follows:



The service port is used by the RPC implementation in the Amoeba kernel to locate a server implementing the service that manages the object. In many cases there is a one-to-one correspondence between servers and services (each service is implemented by exactly one server process), but some services are replicated. For instance, Amoeba's directory service, which is crucial to gain access to almost all other services, is implemented by two servers that listen on the same port and know about exactly the same objects.

The object number in the capability is used by the server receiving the request to identify the object to which the operation applies. The rights bits specify which operations the holder of the capability may apply. The last part of a capability is a 48-bit long "check word", which is used to prevent forgery. The check word is computed by the server based upon the rights bits and a random key per object that it keeps secret. If you change the rights bits you must compute the proper check word or else the server will refuse the capability. Due to the size of the check word and the nature of the cryptographic "one-way function" used to compute it, inverting this function is impractical, so forging capabilities is impossible.†

A working Amoeba system is a collection of diverse servers, managing files, directories, processes, devices etc. While most servers have their own interface, there are some requests that make sense for some or all object types. For instance, the *std_info()* request, which returns a short descriptive string, applies to all object types. Likewise, *std_destroy()* applies to files, directories and processes, but not to devices.

Similarly, different file server implementations may want to offer the same interface for operations like *read()* and *write()* to their clients. AIL's grouping of requests into classes is ideally suited to describe this kind of interface sharing, and a class hierarchy results which clearly shows the similarities between server interfaces (not necessarily their implementations!).

The base class of all classes defines the *std_info()* request. Most server interfaces actually inherit a derived class that also defines *std_destroy()*. File servers inherit a class that defines the common operations on files, etc.

3.2. How AIL Works

The AIL stub generator functions in three phases:

- Parsing,
- Strategy Determination,
- Code Generation.

Phase one parses the input and builds a symbol table containing everything it knows about the classes and other definitions found in the input.

Phase two determines the strategy to use by taking each function declaration in turn and decides upon the request and reply message formats. This is not a simple matter, because of various optimization attempts. Amoeba's kernel interface for RPC requests takes a fixed-size header and one arbitrary-size buffer. A large part of the header holds the capability of the object to which the request is directed, but there is some space left for a few integer parameters whose interpreta-

† As computers become faster, inverting the one-way function becomes less impractical. Therefore, the next version of Amoeba will have 64-bit check words.

tion is up to the server. AIL tries to use these slots for simple integer parameters, for two reasons.

First, unlike the buffer, header fields are byte-swapped by the RPC layer in the kernel if necessary, so it saves a few byte swapping instructions in the user code. Second, and more important, a common form of request transfers a few integers and one large buffer to or from a server. The *read()* and *write()* requests of most file servers have this form, for instance. If it is possible to place all integer parameters in the header, the address of the buffer parameter can be passed directly to the kernel RPC layer. While AIL is perfectly capable of handling requests that do not fit this format, the resulting code involves allocating a new buffer and copying all parameters into it. It is a top priority to avoid this copying ("marshalling") if at all possible, in order to maintain Amoeba's famous RPC performance.

When AIL resorts to copying parameters into a buffer, it reorders them so that integers indicating the lengths of variable-size arrays are placed in the buffer before the arrays they describe, since otherwise decoding the request would be impossible. It also adds occasional padding bytes to ensure integers are aligned properly in the buffer – this can speed up (un)marshalling.

Phase three is the code generator, or back-end. There are in fact many different back-ends that may be called in a single run to generate different types of output. The most important output types are header files (for inclusion by the clients of an interface), client stubs, and "server main loop" code. The latter decodes incoming requests in the server. The generated code depends on the programming language requested, and there are separate back-ends for each supported language.

It is important that the strategy chosen by phase two is independent of the language requested for phase three – otherwise the interoperability of servers and clients written in different languages would be compromised.

4. Linking AIL to Python

From the previous section it can be concluded that linking AIL to Python is a matter of writing a back-end for Python. This is indeed what we did.

Considerable time went into the design of the back-end in order to make the resulting RPC interface for Python fit as smoothly as possible in Python's programming style. For instance, the issues of parameter transfer, variable-size arrays, error handling, and call syntax were all solved in a manner that favors ease of use in Python rather than strict correspondence with the stubs generated for C.

4.1. Mapping AIL Entities to Python

For each programming language that AIL is to support, a mapping must be designed between the data types in AIL and those in that language. Other aspects of the programming languages, such as differences in function call semantics, must also be taken care of.

While the mapping for C is mostly straightforward, the mapping for Python requires a little thinking to get the best results for Python programmers.

4.1.1. Parameter Transfer Direction

Perhaps the simplest issue is that of parameter transfer direction. Parameters of functions declared in AIL are categorized as being of type `in`, `out` or `in out` (the same distinction as made in Ada). Python only has call-by-value parameter semantics; functions can return multiple values as a tuple. This means that, unlike the C back-end, the Python back-end cannot always generate Python functions with exactly the same parameter list as the AIL functions.

Instead, the Python parameter list consists of all `in` and `in out` parameters, in the order in which they occur in the AIL parameter list; similarly, the Python function returns a tuple containing all `in out` and `out` parameters. (In fact Python packs function parameters into a tuple as well, stressing the symmetry between parameters and return value.) For example, a stub with this AIL parameter list:

```
in int p1, in out int p2, in int p3, out int p4
```

will have the following parameter list and return values in Python:

```
(p1, p2, p3)  →  (p2, p4)
```

4.1.2. Variable-size Entities

The support for variable-size objects in AIL is strongly guided by the limitations of C in this matter. Basically, AIL allows what is feasible in C: functions may have variable-size arrays as parameters (both input or output), provided their length is passed separately. In practice this is narrowed to the following rule: for each variable-size array parameter, there must be an integer parameter giving its length. (An exception for null-terminated strings is planned but not yet implemented.)

Variable-size arrays in AIL or C correspond to *sequences* in Python: lists, tuples or strings. These are much easier to use than their C counterparts. Given a sequence object in Python, it is always possible to determine its size: the built-in function `len()` returns it. It would be annoying to require the caller of an RPC stub with a variable-size parameter to also pass a parameter that explicitly gives its size. Therefore we eliminate all parameters from the Python parameter list whose value is used as the size of a variable-size array. Such parameters are easily found: the array bound expression contains the name of the parameter giving its size. This requires the stub code to work harder (it has to recover the value for size parameters from the corresponding sequence parameter), but at least part of this work would otherwise be needed as well, to check that the given and actual sizes match.

Because of the symmetry in Python between the parameter list and the return value of a function, the same elimination is performed on return values containing variable-size arrays: integers returned solely to tell the client the size of a returned array are not returned explicitly to the caller in Python.

4.1.3. Error Handling

Another point where Python is really better than C is the issue of error handling. It is a fact of life that everything involving RPC may fail, for a variety of reasons outside the user's control: the network may be disconnected, the server may be down, etc. Clients must be prepared to handle such failures and recover from them, or at least print an error message and die. In C this means that every function returns an error status that must be checked by the caller, causing programs to be cluttered with error checks – or worse, programs that ignore errors.

In Python, errors are generally indicated by exceptions, which can be handled out of line from the main control flow if necessary, and cause immediate program termination (with a stack trace) if ignored. To profit from this feature, all RPC errors that may be encountered by AIL-generated stubs in Python are turned into exceptions. An extra value passed together with the exception is used to relay the error code returned by the server to the handler. Since in general RPC failures are rare, Python test programs can usually ignore exceptions – making the program simpler – without the risk of occasional errors going undetected. (I still remember a hundredfold speed improvement reported, long, long, ago, about a new version of a certain program, which later had to be attributed to a benchmark failing silently...)

4.1.4. Function Call Syntax

Amoeba RPC operations always need a capability parameter; the service is identified by the port field of the capability. In C, the capability must always be the first parameter of the stub function, but in Python we can do better.

A Python capability is an opaque object type in its own right, which is used, for instance, as parameter to and return value from Amoeba's name server functions. Python objects can have methods, so it is convenient to make all AIL-generated stubs methods of capabilities instead of just functions. Therefore, instead of writing

```
some_stub(cap, other_parameters)
```

(as in C), Python programmers can write

```
cap.some_stub(other_parameters)
```

This is better because it reduces name conflicts: in Python, no confusion is possible between a stub and a local or global variable or user-defined function with the same name.

4.1.5. Example

All the preceding principles can be seen at work in the following example. Suppose a function is declared in AIL as follows:

```
some_stub(*, in char buf[size:1000], in int size,
             out int n_done, out int status);
```

In C it might be called by the following code (including declarations, for clarity, but not initializations):

```
int err, n_done, status;
capability cap;
char buf[500];
...
err = some_stub(&cap, buf, sizeof buf, &n_done, &status);
if (err != 0) return err;
printf("%d done; status = %d\n", n_done, status);
```

Equivalent code in Python might look as follows:

```
cap = ...
buf = ...
n_done, status = cap.some_stub(buf)
print n_done, 'done;', 'status =', status
```

No explicit error check is required in Python; if the RPC fails, an exception is raised so the `print` statement is never reached.

4.2. The Implementation

More or less orthogonal to the issue of how to map AIL operations to the Python language is the question of how they should be implemented.

In principle it would be possible to use the same strategy that is used for C: add an interface to Amoeba's RPC primitives to Python and generate Python code to marshal parameters into and out of a buffer. However, Python's high-level data types are not well suited for marshalling: byte-level operations are clumsy and expensive, with the result that marshalling a single byte of data can take several Python statements. This would mean that a large amount of code would be needed to implement a stub, which would cost a lot of time to parse and take up a lot of space in "compiled" form (as parse tree or pseudo code). Execution of the marshalling code would be sluggish as well.

We therefore chose an alternate approach, writing the marshalling in C, which is efficient at such byte-level operations. While it is easy enough to generate C code that can be linked with the Python interpreter, it would obviously not stimulate the use of Python for server testing if each change to an interface required relinking the interpreter. This is circumvented by the following, rather elegant solution: the marshalling is handled by a simple *virtual machine*, and AIL generates instructions for this machine. An interpreter for the machine is linked into the Python interpreter and reads its instructions from a file written by AIL.

The machine language for our virtual machine is dubbed *Stubcode*. Stubcode is a super-specialized language. There are two sets of about a dozen instructions each: one set marshals Python objects representing parameters into a buffer, the other set (similar but not quite symmetric) unmarshals results from a buffer into Python objects. The Stubcode interpreter uses a stack to hold Python intermediate results. Other state elements are an Amoeba header and buffer, a pointer indicating the current position in the buffer, and of course a program counter. Besides (un)marshalling, the virtual machine must also implement type checking, and raise a Python exception when a parameter does not have the expected type.

The Stubcode interpreter marshals Python data types very efficiently, since each instruction can marshal a large amount of data. For instance, a whole Python string is marshalled by a single Stubcode instruction, which (after some checking) executes the most efficient byte-copying loop possible – i.e., it calls `memcpy()`.

Construction details of the Stubcode interpreter are straightforward. Most complications are caused by the peculiarities of AIL's strategy module and Python's type system. By far the most complex single instruction is the "loop" instruction, which is used to marshal arrays.

As an example, here is the complete Stubcode program (with spaces and comments added for clarity) generated for the function `some_stub()` of the example above. The stack contains pointers to Python objects, and its initial contents is the parameter to the function, the string `buf`. The final stack contents will be the function return value, the tuple `(n_done, status)`. The name `header` refers to the fixed size Amoeba RPC header structure.

```
BufSize     1000            Allocate RPC buffer of 1000 bytes
Dup         1               Duplicate stack top
StringS                     Replace stack top by its string size
PutI        h_extra int32   Store top element in header.h_extra
TStringSlt  1000            Assert string size less than 1000
PutVS                       Marshal variable-size string
Trans       1234            Execute the RPC (request code 1234)
GetI        h_extra int32   Push integer from header.h_extra
GetI        h_size int32    Push integer from header.h_size
Pack        2               Pack top 2 elements into a tuple
```

As much work as possible is done by the Python back-end in AIL, rather than in the Stubcode interpreter, to make the latter both simple and fast. For instance, the decision to eliminate an array size parameter from the Python parameter list is taken by AIL, and Stubcode instructions are generated to recover the size from the actual parameter and to marshal it properly. Similarly, there is a special alignment instruction (not used in the example) to meet alignment requirements.

Communication between AIL and the Stubcode generator is via the file system. For each stub function, AIL creates a file in its output directory, named after the stub with a specific suffix. This file contains a machine-readable version of the Stubcode program for the stub. The Python user can specify a search path containing directories, which the interpreter searches for a Stubcode file the first time a particular stub is used.

The transformations on the parameter list and data types needed to map AIL data types to Python data types make it necessary to help the Python programmer a bit in figuring out the parameters to a call. Although in most cases the rules are simple enough, it is sometimes is hard to figure out exactly what the parameter and return values of a particular stub are. There are two sources of help in this case: first, the exception contains enough information so that the user can figure what type was expected where; second, AIL's Python back-end optionally generates a human-readable "interface specification" file.

5. Conclusion

We have succeeded in creating a useful extension to Python that enables Amoeba server writers to test and experiment with their server in a much more interactive manner. We hope that this facility will add to the popularity of AIL amongst Amoeba programmers.

Python's extensibility was proven convincingly by the exercise (performed by the second author) of adding the Stubcode interpreter to Python. Standard data abstraction techniques are used to insulate extension modules from details of the rest of the Python interpreter; in the case of the Stubcode interpreter this worked well enough that it survived a major overhaul of the main Python interpreter virtually unchanged.

On the other hand, adding a new back-end to AIL turned out to be quite a bit of work. One problem, specific to Python, was to be expected: Python's variable-size data types differ considerably from the C-derived data model that AIL favours. Two additional problems we encountered were the complexity of the interface between AIL's second and third phases, and a number of remaining bugs in the second phase that surfaced when the implementation of the Python back-end was tested. The bugs have been tracked down and fixed, but nothing permanent has been done about the complexity of the interface.

5.1. Future Plans

AIL's C back-end generates server main loop code as well as client stubs. The Python back-end currently only generates client stubs, so it is not yet possible to write servers in Python. While it is clearly more important to be able to use Python as a client than as a server, the ability to write server prototypes in Python would be a valuable addition: it allows server designers to experiment with interfaces in a much earlier stage of the design, with a much smaller programming effort. This makes it possible to concentrate on concepts first, before worrying about efficient implementation.

The unmarshalling done in the server is almost symmetric with the marshalling in the client, and vice versa, so relative small extensions to the Stubcode virtual machine will allow its use in a server main loop. We hope to find the time to add this feature to a future version of Python.

References

[Bir84a] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* **2**(1), pp. 39-59 (February 1984).

[Bir87a] A. D. Birrell, E. D. Lazowska, and E. Wobber, "Flume – Remote Procedure Call Stub Generator for Modula-2+," Topaz manual page, DEC SRC, Palo Alto, CA (1987).

[Car89a] Luca Cardelli, "Modula-3 Report (revised)," Research Report 52, DEC SRC, Palo Alto, CA (November 1, 1989).

[Geu90a] Leo Geurts, Lambert Meertens, and Steven Pemberton, *ABC Programmer's Handbook,* Prentice-Hall, London (1990). ISBN 0-13-000027-2

[McJ87a] P. R. McJones and G. F. Swart, "Evolving the UNIX System Interface to Support Multithreaded Programs," Research Report 21, DEC SRC, Palo Alto, CA (September 28, 1987).

[Mul90a] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and J. M. van Staveren, "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer Magazine* **23**(5), pp. 44-53 (May 1990).

[Ous90a] John K. Ousterhout, "Tcl: an Embeddable Command Language," pp. 133-146 in *Proceedings of the Winter 1990 USENIX Conference*, USENIX Association, Washington, DC (Januari 1990).

[Ros91a] G. van Rossum, *Python – an Extensible Prototyping Language*, (In preparation), 1991.

[Ros88a] G. van Rossum, "STDWIN – A Standard Window System Interface," CWI Report CS-R8817, CWI, Amsterdam (April 1988).

[Ros90a] G. van Rossum, "AIL – A Class-Oriented Stub Generator for Amoeba," pp. 13-21 in *Workshop on Progress in Distributed Operating Systems and Distributed Systems Management*, ed. E. W. Zimmer, Springer Verlag (1990).

[Tan90a] A. S. Tanenbaum, R. van Renesse, J. M. van Staveren, G. J. Sharp, S. J. Mullender, A. J. Jansen, and G. van Rossum, "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM* **33**(12), pp. 46-63 (December 1990).

Providing Application Interoperability using Functional Programming Concepts

Frank Eliassen Randi Karlsen

University of Tromsø, Norway

{frank | randi}@cs.uit.no

Abstract

Applications that are interoperable (like federated databases) may be manipulated together by the user without global integration. This can be achieved by providing a uniform language format for the definition and manipulation of multiple autonomous applications.

In this paper is given an overview of a proposed uniform language format FRIL based on a computing model and interface paradigm combining functional and object-oriented approaches. The framework for this approach is given as a distributed generic infrastructure (or platform) supporting interoperability of separate and autonomous applications.

FRIL provides remote programmable services interfaces to heterogeneous autonomous applications and includes constructs for formulating global actions combining operations from these interfaces. Global actions are supported by a distributed execution system executing FRIL programs as distributed transactions. Using a functional computing model for the area of distributed transaction processing is an original approach which is both interesting and challenging.

We also briefly present the transaction model for the generic infrastructure and the exception handling facilities of FRIL. The transaction model allows for applications with highly different requirements to local autonomy and local transaction management to participate in the same distributed transaction, e.g. they may exhibit different degrees of control over subtransactions they execute. Fault tolerance is supported by allowing for alternative transactions that are executed instead of transactions that fail.

1. Introduction

It is widely recognized that future information systems will be heterogeneous and distributed containing a wide variety of computer and networking technologies, supplied by a number of information technology vendors. This has generated a strong need for principles and tools for integration of separate functions of independent systems in an organization and for linking together various functions of information systems of *autonomous* organizations as e.g. in electronic trading and international banking [Eli87a].

Within this general framework, the goal of our research activity is the development of a distributed generic infrastructure (or platform) providing supporting services and tools for making (possibly) preexisting, heterogeneous applications interoperable. Applications that are interoperable may be manipulated together without global integration [Lit86a]. The approach taken is to provide a uniform language format combining functional and object-oriented paradigms, for the definition and manipulation of multiple autonomous applications [Eli88a, Eli89a].

The suitability of a function-based approach as the basis for a canonical data model or uniform language for heterogeneous databases and applications integration has been advocated by many authors. The application of functional programming concepts to the data representation and querying aspects of databases has been discussed in, for example, [Shi81a, Atk84a, Bun84a] while functional approaches to additional aspects of database systems like updating and transactions are advocated in [Kel85a]. In recent years object-oriented approaches have also been applied for (generalized) databases [Kim89a]. Also a growing interest in combining functional and object-oriented approaches have been observed over the last years [Gog87a, Fis87a, Day89a]. For example, queries against databases are applicative in nature and hence can be expressed as functional programs. A declarative style for query languages is generally considered an advantage. From these developments we have come to the conclusion that the combination of the functional and object-oriented approaches seems highly suitable for modelling and integrating a wide variety of different application systems and their (programmatic) service interfaces, including (generalized) databases, multimedia and multipurpose systems.

Our approach to a uniform language format encompasses a notation (SESSL) for specifying abstract service interfaces. An abstract service interface defines the view remote systems will have of the offered service. Application interoperation is supported through the language FIOL that allows for the formulation of expressions (i.e. global programs) combining individual operations from multiple service interfaces. SESSL and FIOL together constitute the two language component of the language FRIL [Eli88a, Eli89a].

The interface notation must be capable of expressing semantics preserving abstractions of highly different (existing) service interfaces. We have therefore based the notation for specifying abstract programmable service interfaces on the idea of algebraic specifications of abstract data types (ADTs) [Gut78a]. This technique facilitates object-oriented and fully implementation independent interface specifications which is of particular value for the purpose of overcoming heterogeneity.

An additional requirement to the infrastructure is to support reliable execution of global actions, i.e. actions with transactional properties like atomicity, consistency preservation and concurrency [Eli88a]. To support this requirement a flexible transaction model preserving the autonomy of the component systems of a federation and tolerating failures of individual subtransactions, have been developed [Kar91a].

In most (database) systems, updates are provided through different "non-functional" mechanisms relying on side-effects. These mechanisms are clearly in conflict with a functional approach. However, to serve as a vehicle for application interoperability, the uniform language format must also be able to capture this kind of semantics. The outstanding challange is to represent update in such a way that it can be handled within a functional programming framework and without com-

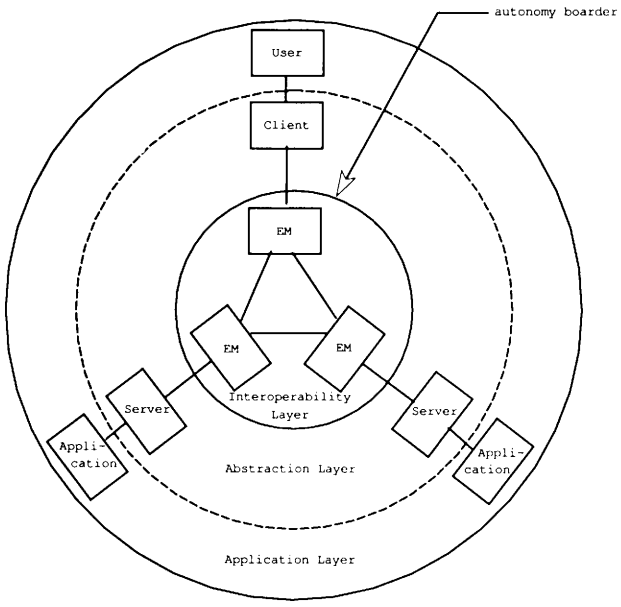


Figure 1: *Layered architecture for interoperable applications*

pletely destroying the elegance of the language [Bun90a]. Our contribution to this challenge is a way of declaring side-effects of functions and constraining permissable threads of applications of functions with side-effects. This mechanism can be used to model different forms of updates of objects within a functional framework, including update semantics relying on side-effects.

In the remainder of the paper we present our approach in greater detail. In Section 2 we give a overview of an architecture model for application interoperability. Section 3 gives a brief overview of the flexible transaction model supporting local system autonomy and alternative transactions. It is shown how this model can be integrated into the infrastructure by mapping different levels of transaction management functions onto different entities of the above architecture. In Section 4 we present FIOL and the notation for interface specifications (SESSL), including our approach for modelling update. Section 5 presents exception handling in SESSL and FIOL for the support of alternative transactions.

2. Overview of System Architecture Model

The conceptual framework for the functional approach to interoperable applications is modeled as a three layered architecture. Figure 1 shows the architecture as shells of an onion. The viewpoint or perspective of the architecture is interoperation. It is not intended as a total model covering all aspects of distributed applications.

The application layer contains the existing information systems to be made interoperable as well as entity types representing the users of these information systems. These systems interface to the infrastructure using the services and functions of the abstraction layer. The abstraction layer is populated by generic entity types called client and server entities. The purpose of the abstraction layer is to add the functionality required for enhancing or abstracting the application service interfaces to the linguistic form required by the interoperability layer.

The actual set of required abstraction layer functions will vary from application to application depending on the properties of the actual application service interface and thus in general can not standardized. For some application types, however, it is envisaged that more or less standardized abstraction layer front end server entities can be developed (e.g. front-ends to standard SQL databases [Eli90a]).

At the boundary between the abstraction layer and the interoperability layer the decentralized environment of interoperable applications appears as composed of server entities and client entities. The interoperability layer provides the necessary services that enable clients to manipulate together the services of one or more servers. Its primary service is the EVAL service. The EVAL service executes programs expressed in the functional resource integration language FRIL on request from clients. The language supports application interoperation by allowing for the formulation of expressions (i.e. global programs) combining individual operations from multiple server interfaces.

Architecturally the EVAL service is modeled as a set of cooperating Evaluation Managers (EMs) managing the evaluation of FRIL expressions as global distributed transactions (see Figure 1). The evaluation generally proceeds in a distributed and nested manner based on repeated decompositions of FIOL expressions and evaluation of subexpressions by different EM entities. Each decomposition will be done according to the locations of the data objects being addressed in the (sub)expression and possibly according to some overall optimization strategy (e.g. to reduce network traffic). The EM entities exploit the appropriate servers to evaluate the different component functions.

The EMs communicate using the asynchronous remote function call (RFC) protocol. Through this protocol an EM entity (the requestor) requests some other EM entity (the responder) to evaluate a FRIL expression for some given arguments and subsequently collects the result. The EMs communicate with the client and server entities using a similar protocol. This latter protocol defines the boundary of autonomy between a local and the global system.

3. Flexible Transactions for Interoperable Applications

One of the main difficulties to overcome in the design of a transaction model for interoperable applications is implied by the requirement for autonomy. The notion of flexible transactions have been developed to solve some of these difficulties [Kar91a]. In the following we briefly describe the flexible transaction model and its relation to the architecture described in the previous Section.

3.1. Properties of the Flexible Transaction Model

The requirement for autonomy implies that the transaction model supporting interoperation needs to distinguish between *locally and globally controlled transaction management*. Thus a global transaction will be managed partly at the global level (i.e. interoperability layer) and partly at the local level (i.e. abstraction and application layers).

Applications in a federation are *autonomous* [Eli87b]. This implies (among others) that an application may require transaction commit based on local considerations only (locally controlled commit) or it may leave commit control to the EM (globally controlled commit).

The transaction model supports both kinds of commit within one global transaction.

Requirements for fault tolerance give raise to the concept of alternative transactions [Eli87a]. This means that the specification of a global transaction can include the specification of alternative subtransactions which will be executed if a specific subtransaction is unable to complete successfully. A flexible transaction concept can thus be supported allowing users to specify alternative actions for implementing the same task. Consequently a global transaction can execute successfully and commit even when some of its subtransactions fail.

Traditional transaction models (e.g. [Ber87a, Cer84a]) does not fulfill the requirement for flexibility in the above sense. An independently developed transaction model supporting interoperability for the InterBase system [Elm90a], are in some respects similar to the transaction model presented in this paper.

3.2. Architecture for Transaction Execution

We assume the global layer management is handled by several global transaction managers (GTMs) being constituent parts of the evaluation managers (EMs) of the interoperability layer. We also assume that every autonomous application entity has a local transaction manager (LTM) which manages the execution of local transactions (see Figure 2).

The GTMs require a minimum level of functionality from the LTMs in order to be able to utilize local transaction management as a basis for the management of global transactions. This functionality is for each application entity provided by a module of the application's associated server entity called LTM$^+$. The LTM$^+$ thus enhances the functionality of the LTM to the minimum level required by the GTM.

When a global transaction T is submitted by a client to its associated EM, the transaction is decomposed into a number of subtransactions $T_1, \ldots, T_n$ by the global resource manager module GRM (see Figure 2). Each subtransaction is assigned to an appropriate subordinate EM for execution. This process is called *mapping*.

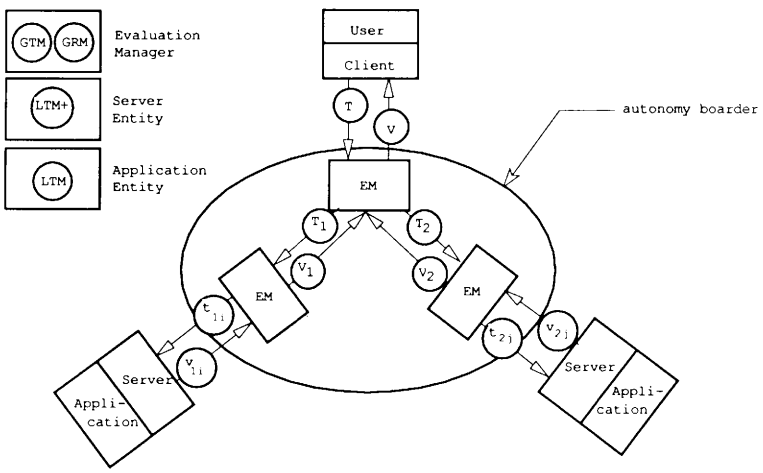


Figure 2: *Transaction execution*

A subtransaction t_{ij} mapped to a server is executed as an autonomous transaction managed by the LTM in association with the LTM$^+$. The server entity prepares the subtransaction for local execution by setting up an execution plan for t_{ij} and translates it into the transaction language of the LTM.

After the completion of a subtransaction T_i the result v_i is communicated back to the superior EM which uses the result to complete the execution of its assigned subtransaction. When the execution of the global transaction T is completed, T is commited and the final result V is returned to the client (commitment is not shown in Figure 2).

3.3. A Transaction Model for Flexible Transactions

In the flexible transaction model global transactions are executed as nested transactions where each global transaction (GT) consists of a number of (sub)transactions organized into a hierarchy. The top level GT represents the global transaction issued by the client. Each GT in the hierarchy is mapped to an EM entity and managed by the corresponding GTM.

The leaf transactions (having no subtransactions) we call *canonical local transactions* (CLT). A CLT is mapped to a server entity and managed by the LTM/LTM$^+$ as a single autonoumous transaction. A CLT is *local* in the sense that it is executed in its entirety by a single server, and it is *canonical* in the sense that it is expressed in the uniform language format of the infrastructure.

A global transaction T is represented by a tuple <ST,O,D,C> where ST is a set of subtransactions $\{T_1, \ldots, T_n\}$, O is a partial order on $\{T_1, \ldots, T_n\}$, D is a set of subtransaction dependencies among $T_1, \ldots, T_n$ and C represents consistency information. The GTM associated to the EM evaluating T is responsible for managing the set of subtransactions (represented by ST) as one global transaction.

The partial order O determines the execution order of $T_1, \ldots, T_n$ and can be derived from the set of subtransaction dependencies D. The model distinguishes between *compositional, conditional* and *failure* dependency.

- *Compositional dependency.* A transaction T_j is compositional dependent on a transaction T_i if T_j uses the result of (a successfully executed) T_i as argument. This implies that T_j can not be executed before T_i has completed successfully.
- *Conditional dependency.* A transaction T_j is conditional dependent on a transaction T_i if T_j is executed only if T_i terminates with a particular (explicitly specified) result.
- *Failure dependency.* A transaction T_j is failure dependent of a transaction T_i if T_j is executed only if T_i is tried executed and has failed. T_i is the primary transaction while T_j is an alternative transaction.

The latter kind of dependency is of particular interest for fault tolerance. The intended meaning of failure dependency is that a primary transaction is the preferred transaction (the first choice) while an alternative transaction is executed only if T_i fails. An alternative transaction can itself be a primary transaction having a set of failure dependent, alternative transactions attached.

The consistency information C is used by the GTMs to enforce consistency for concurrent transactions.

3.4. Transaction Management

The execution of a global transaction will either complete successfully or fail. This outcome depends on the success or failure of its subtransactions.

For every global transaction $T = <ST, O, D, C>$ a set Ω of subsets of ST can be derived from the dependencies D. Every member of Ω represents an alternative acceptable execution of T. A member of Ω thus consists of a set of subtransactions that together represent an acceptable result and hence a complete execution of T. The transaction T can be committed only if T is successfully executed according to these requirements.

When a subtransaction T_i terminates, the subordinate GTM returns either a positive or negative response to its superior GTM. A *negative response* indicates that T_i has failed and has been aborted and undone while a *positive response* indicates that T_i was successfully executed. The superior GTM can request either a commit or an abort action from a successfully executed subtransaction.

CLTs are either globally or locally committed. Local commit of a CLT is controlled by the LTM/LTM$^+$. The CLT is normally commited as soon as it is successfully executed. CLTs committing locally must be compensatable, meaning that there exist a compensating transaction that semantically undoes the result of the executed and committed CLT [Gar83a]. A locally committed subtransaction can on request from the GTM be compensated (by the LTM$^+$). It is assumed that the LTM$^+$ managing a compensatable CLT also constructs the corresponding compensating transaction during the execution of the CLT.

Global commit of a CLT is controlled by the GTMs and is executed when the global transaction T is successfully executed (and ready to commit). Abort is the only recovery action needed for globally committed CLTs.

If T_i is mapped as a global transaction and fails, a negative response is returned by the subordinate GTM. The superior GTM aborts T and chooses the preferred alternative subtransaction for T among the failure dependent transactions (if any). If no alternative transaction exist or every alternative has been executed and failed, T can not be successfully executed and is deemed to fail. As a result a negative response is returned to the next superior GTM.

If T_i is mapped as a CLT and fails, the GTM receives a negative response from the corresponding server. In this case the GTM chooses the preferred alternative transaction S for T_i and executes S as a subtransaction of T. If no alternative exist for T_i, T is aborted and an alternative for T is searched for (c.f. above).

A transaction T can be committed if the set of successfully executed subtransactions is in Ω. To commit a transaction T, commit requests are (recursivly) propagated to every subordinate transaction manager managing a successfully executed subtransaction. To abort T, abort or compensate requests are sent instead.

To preserve consistency, a transaction T must satisfy both global and local consistency constraints. Local constraints describe the internal consistency requirements for each application. Global constraints must be an addition to local constraints and describe consistency requirements between the interoperable applications. This implies that global consistency can only meaningfully be concerned about dynamic constraints determining the manner in which state transitions may occur.

Concurrent execution of conflicting transactions must be synchronized to prevent inconsistency. Due to autonomy requirements, synchronization of concurrent transactions spanning interoperable applications is generally difficult [Du89a]. Consistency is often achieved at the expence of concurrency and/or autonomy. For the GTMs we combine two approaches. Firstly, the decomposition of global transactions is constrained (e.g. to preserve the view of immutability, see Section 4.2.4). Secondly, a global scheduler will synchronize concurrent execution of conflicting global transactions according to global interleaving constraints. The details of this approach will be reported elsewhere [Kar91a].

4. The Functional Resource Integration Language

In the overview of FRIL we shall concentrate on semantics and only briefly describe its built-in data types and control structures. The syntax of FRIL is mainly inspired by the syntax of Backus' FP [Bac78a] and an extension of FP with a type system described in [Gut81a]. FP embodies a functional style of programming in which variablefree programs are built from a set of primitive programs by a small set of combining forms. We describe the two components of FRIL, SESSL and FIOL, separately.

4.1. Global Transaction Programs

FIOL is the glue that ties all the export schemata (in a federation) together in the sense that it allows for the formulation of global transactions combining functions of multiple export schemata in a single expression.

FIOL recognizes two kinds of functions. *Server functions* are functional operators provided by servers via corresponding export schemata while *primitive functions* of FIOL correspond to the operators of the built in primitive data types and the sequence data type. Typical ones are addition, subtraction, multiplication and division of numbers, boolean operators like logical and, or and not, as well as operators for manipulating sequences.

In FRIL, everything is considered a function. For example, the integer number 4 is considered a constant function with codomain integer. Thus the result of every function evaluation is a function. The evaluation stops when the function cannot be evaluated any further (i.e. a normal form is reached).

Combining forms are the programming constructs of FRIL and are used to construct new functions from already existing functions. They take functions as arguments and map them into "higher-order" functions. The combining forms of FRIL are basically those found in FP. Some of them are defined below. The symbols f, g, h, x, y, $x_1, \ldots, x_n$ denote arbitrary function expressions, while p denotes a function with range boolean. The sequence manipulating function *Apndl* (append left) appearing below, is defined by the equation

$$Apndl\ [y, [x_1, \ldots, x_n]\ ] = [y, x_1, \ldots, x_n]$$

Composition:	$f \mid g \mid h \mid x = f \mid (g \mid (h \mid x))$
Construction:	$[f, g, h] \mid x = [f \mid x, g \mid x, h \mid x]$
Condition:	$(p \rightarrow f ; g) \mid x =$ if $p \mid x \equiv$ *true* **then** $f \mid x$ **else** $g \mid x$
Filter:	$\& p \mid [x_1, \ldots, x_n] =$ if $p \mid x_1 \equiv$ *true* **then** *Apndl* $\mid [x_1, \& p \mid [x_2, \ldots, x_n]]$ **else** $\& p \mid [x_2, \ldots, x_n]$
Apply to all:	$*f \mid [x_1, \ldots, x_n] = [f \mid x_1, \ldots, f \mid x_n], *f \mid [\,] = [\,]$
Insert right:	$/f \mid [x_1, \ldots, x_n] = f \mid x_1, [/f \mid [x_2, \ldots, x_n]], /f \mid [x] = x$

A FIOL program denotes a global transaction definition. It is a set of equations of the form $f = e$ defining various functions where f is a simple identifier and e is an expression. A very simple example of a FIOL program is:

def

$$T = e \mid [t_1, t_2]$$

$$t_1 = e_1$$

$$t_2 = e_2$$

All except the first equation of the program are local definitions. Local definitions are evaluated when needed (lazy evaluation) and reused by reference to their identifiers (t_1 and t_2 in the example).

The combining forms define different ways of combining functions into higher order functions. In general they also express various forms of dependencies between functions corresponding to the concept of sub-transaction dependencies defined in Section 3.2. The combining forms *composition, filter, apply to all* and *insert* defines different forms of compositional dependencies, while *condition* is the combining form for expressing conditional dependency. The *construction* combining form does not imply any dependencies at all between its argument functions, while the concept of failure dependency in the flexible transaction model is supported by substitute functions and will be dealt with in Section 5.

4.2. Server Interfaces

The language for specifying abstract server interfaces we name SESSL. Below we comment on the notions of object persistency, dependency, immutability, mutability, naming and update all supported by the language. Additionally SESSL supports a number of other concepts like subtyping, inheritance, and modular and parameterized (polymorphic) specifications. For details the reader is referred to [Eli89a].

4.2.1. Programmable Server Interfaces

A server offers its services to potential clients through a *server export schema* [Hei85a]. The server export schema contains a specification of the offered services and comprises the set of operation requests it may accept from potential clients and for each operation request, the responses that can be expected as a result of the request. The export schema also groups the functions into access control groups. Clients need not be given access to all the groups.

An abstract server interface is *programmable* in the sense that a server accepts from clients complete FIOL expressions for evaluation. Such expressions correspond to the notion of canonical local transactions (CLTs) in the flexible transaction model. It is required that a CLT consists of functions all specified in the server's export schema. We

assume that the set of built-in functions in FIOL are implicitly part of any export schema.

The abstract syntax of SESSL is mainly inspired by algebraic specification languages like OBJ2 [Fut85a] and Larch [Gut85a]. A specification of a server interface providing access to a database of accounts would typically include two ADTs AccDB and Account. The signature of each ADT describes the operations that can be performed on objects of the type. The specification of the database type could include the functions

insert:[AccDB, Account] → AccDB

delete:[AccDB, Account] → AccDB

while the specification of the type Account might encompass functions like

name:Account → AccName

bal:Account → Money

hist:Account → *[Date, Money]

credit:[Account, Money] → Account

debit:[Account, Money] → Account

The notation $f:S \rightarrow T$ indicates a function with domain S and codomain T. The type constructor $[T_1, \ldots, T_n]$ denotes the type of tuples of objects of type $T_1, \ldots, T_n$, while the type notation $*T$ specifies a sequence or stream of objects of type T. The filter combining form of FIOL can be used to restrict streams of objects (i.e. retrieval by object value). An example of a derived operation (method) might be

transfer:[AccDB, AccName, AccName, Money] →AccDB

A server specification method based on the approach of ADTs, supports an object-oriented model of information services. ADTs are essentially abstract definitions for potential data objects in a system (i.e. a server). The export schema specifies the semantics of these data objects basically in terms of how they can be accessed and manipulated by clients. SESSL supports the specification of both *mutable* and *immutable* object types. A mutable object has value (i.e. state) that can change while the value of an immutable object never changes.

Following [Gut78a] the type being defined is referred to as the type of interest (TOI). In the above example the TOI for the functions insert and delete is the type AccDB while the TOI for the functions name, bal, credit and debit is the type Account. The functional operators belonging to the TOI are grouped under appropriate headings according to their class. The classes are *primitive constructors* to construct starting values of the TOI, *constructors* to construct new values of the TOI from old ones, *mutators* that modify the object value of a mutable TOI, or *extractors* for extracting component values of the TOI [Lis86a].

4.2.2. Global Object Representation

To overcome heterogeneity at the global level (i.e. in the interoperability layer) there is a need for a global object representation format independent of the various local object representation formats that can be found in existing systems and applications. Use of ADTs as the abstraction mechanism of objects means that we can represent an object by its state.

[Eli89a] introduces *unique identifiers* (UIDs) as global level object representation. When a client requests a server to create a new account (say), the global level representation of the state of the account (a UID generated by the server) will be returned to the client. The server is responsible for maintaining a mapping between the UID and some internal representation of the account's state. The internal representation is hidden for the client. A client can later reference or access the account by presenting the corresponding UID to the server. Formally UIDs are considered as constant functions. E.g. if u denotes a UID for an account, u would have signature $u:\rightarrow$ Account.

We integrate UIDs into SESSL by letting the TOI argument of a functional operator be the carrier of the global object representation (UIDs). Hence if $f:T, X\rightarrow Y$ is a function provided by some server and T represents the TOI, the first actual argument in a call on f will be a UID identifying a corresponding data object of type T on which the operation f is to be applied.

4.2.3. Object Persistency

The general mechanism of providing UIDs described above, is a way of modelling persistent server objects. However, without any additional mechanisms constraining this approach in some way, a server will be forced to maintain UIDs even for very fine grained objects. A worse problem is that the application to be made interoperable may not support an object concept at all (e.g. a relational database) and hence no concept of object identifier either.

In order not to put too strong requirements on the server functionality, the concept of *persistence dependency* allows a server to declare for which object types it will provide UIDs to clients and for which it will not [Eli89a]. This does not prevent servers and applications internally to maintain (surrogate) UIDs for dependent objects, but these UIDs are never revealed for clients other than as a particular nil-value denoted *void*. In fact, the computational model of FRIL assumes the logical existence of internal surrogate UIDs for *all* data objects provided by a server. These are needed for specifying the semantics of the execution of CLTs.

In general the persistency and life time of an object is tightly coupled to the ability to reference it. The persistency of objects of a type T declared to be persistent dependent must rely on the existence of a parent type that has an extractor having T as codomain. This means that an object of type T can only be made persistent in the context of a parent object of type Q and can only be referenced via extractors of the latter object. Both objects must be located at the same server.

The concept of attribute-relation (or α-relation for short) is introduced to capture the idea of objects having other objects as attributes. Informally an object O_1 is α-related to an object O_2 if O_1 can be retrieved via some extractor function of O_2. This relation is used to express the condition for persistency of dependent objects. A dependent object O_1 is persistent if and only if there exists a persistent object O_2 such that O_1 is α-related to O_2.

The underlaying computational model of FRIL specifies that when the computation of a CLT terminates (i.e. the CLT commits), all dependent objects accessed by the CLT that does not have an α-relationship to some persistent object, will (logically) be taken by garbage collection.

4.2.4. Modelling Update of Objects

Under a pure functional approach there is no concept of updating an object and accordingly no notion of mutability and corresponding mutators. Rather there is only immutability and the effect of applying a constructor function will be (at least conceptually) to create a *new* object from the old one referred to in the function argument. The old object will remain as before. This we refer to as the *view of immutability* and allow us to more easily model concepts like object versions and version histories. A constructor can be viewed as creating a new version of an object, and repeated applications of constructors as creating version histories.

It is required, however, that SESSL also be capable of expressing abstractions of applications that do not support immutability, but rather model update relying on side-effects (i.e. mutable objects). Most database systems are of this kind. In response to this requirement, SESSL allows an ADT appearing in an export schema to be declared mutable (c.f. above). Naturally it is only mutable types that can have mutators, but mutable types may of course also have constructors.

In order to avoid a wholesale compromise of a pure functional approach, the computational model of FRIL requires the view of immutability be supported by the servers for all objects during the execution of a transaction. For mutable objects this means that during execution of a transaction, mutators logically behave like constructors such that executing a mutator results in a new object being constructed in stead of modifying one. The binding between UID and object value does not change before the transaction commits. Informally this corresponds to establishing a shadow copy of the mutable object to be updated and performing the update on this copy. Once the CLT commits, the old object version is made unavailable for future access (and thus can be deleted) and the updated shadow copy is established as the "current one". The latter is achieved by letting the UID that previously identified the old object, now identify the updated shadow copy. Repeated updates requires a shadow copy be established for each update.

When the CLT commits, intermediate object versions that was (logically) created during the execution of the CLT and that can not be referenced by clients after commit, can be considered garbage collected by the server. This applies equally to mutable as well as immutable objects and enables complex transformations of immutable objects to be expressed without the need to store intermediate object versions as would have otherwise been the case.

Formally our approach for dealing with mutable objects within a functional framework, is similar to the idea of reflective semantics of functional languages [Gog87a]. The effect of updating a mutable object is *reflected* in a changed semantics of the UID viewed as a function. The semantics of the UID is given by the value of the extractors functions of its type when composed with the UID. The changed semantics is realized when the CLT commits.

With respect to concurrent transactions, a sufficient condition for being able to provide the view of immutability is that all global transactions can be decomposed in such a way that there is at most one CLT per server per transaction (a constraint found in many other transaction models as well, see e.g. [Gli84a, Gli86a, Du89a]. Under this condition it is sufficient to consider CLTs only for providing the view of immutability. It is possible, however, sometimes to relax this constraint.

In order to avoid inconsistencies w.r.t. the meaning of FIOL programs containing applications of mutators, there is a need to constrain the permissable threads of applications of mutators within a transaction. This need mainly stem from the possibility of formulating FIOL programs producing inconsisent results. For mutable objects, there is the global constraint that a complex update of a mutable object performed by a transaction, involving several applications of mutators, must form a linear version history and that (under the view of immutability) no intermediate object versions satisfies the conditions for persistency. The constraint guarantees that there never can be more than one candidate object version to be established as the new object value of a mutable object when the transaction commits. Servers may optionally specifiy a similar constraint for immutable object types as well. For objects constrained to linear version histories, servers can also specify that intermediate object versions can not be "observed" during program execution, i.e. extractor functions can not be applied to intermediate object versions. This reduces or eliminates the need for maintaining shadow copies during transaction execution and when applied properly, allows single threaded object constructions to be safely implemented as a series of destructive updates.

5. Linguistic Support for Flexible Transactions

The requirement for a flexible transaction model for FRIL is based on the assumption that the evaluation of a CLT can be refused by the server for whatever reason (e.g. violation of behaviour rules or constraints or lack of access rights). This also normally leads to that the subtransaction enclosing the CLT fails. The execution of subtransactions may also fail due to a variety of other exceptional conditions like communication failure, type errors, etc. This may lead to situations where a global transaction can only be partially executed. In this Section we present language constructs of FRIL supporting the flexible transaction model w.r.t. alternative subtransactions.

5.1. Exception Handling

The support for alternative transactions is represented by the FRIL signalling mechanism and the facility for specifying substitute functions.

The FRIL signalling mechanism allows a server to signal error conditions when a "normal" result cannot be obtained. Our approach is based on the concepts of named terminations (or named returns as we call them) [ANS89a] and *errorsorts* [Gog87a]. Named returns are a generalization of the exception handling scheme found in many programming languages. In FRIL named returns for an object type T are represented by signalling functions of the form

$$E:\rightarrow T$$

The codomain of E is actually $T?$ and is called the error type of T. When the execution of a CLT results in a named return E for one of its component function, the actions taken by the LTM$^+$ is to abort the CLT, recover from the potential effect of the already evaluated component functions of the CLT and issue an error return to the GTM conveying information about the reason for the abort (i.e. some server function f reduced to the named return E).

In addition to the named returns specified by the servers, there are a number of implicit built-in named returns which may be generated by

the FRIL run time system itself corresponding to system failures of different kinds. It is assumed that every function may reduce to system generated named returns.

In a FIOL program it is possible to specify for each occurrence of a server function which named returns the program is willing to accept. If named returns not specified as acceptable to the program should occur, a standard exception (or error recovery) action is started (usually the transaction is aborted).

5.2. Substitute Functions

The concept of alternative subtransactions and the corresponding concept of failure dependency introduced in Section 3, is in FRIL supported by the notion of substitute functions. When specifying acceptable named returns, a substitute function to be executed in stead of the function F that failed (or produced the named return E) must be identified.

Any function that has a substitute function we call a primary function. If no substitute function is specified for a given server function, the default substitute function D will be evaluated. Usually D will correspond to an abort action. A requirement to a substitute function H is that its signature must conform with the signature of its primary function F.

The exception handling mechanism of FRIL is in fact more flexible and general than the mechanism sketched above since not only single server functions can be substituted for but also complete expressions. This will become clearer through the following example illustrating an abstract syntax for declaring substitute functions in FIOL:

def

$T = ...e....$

$e = ...f....g....$

subst

$s_1 = ...e_1....$

$e_1 = ...h....$

$s_2 = ...$

....

rules

$e : (f \Rightarrow n) \rightarrow s_1$ (r1)

$e : (f \Rightarrow \textbf{any}) \rightarrow s_2$ (r2)

$e_1 : (h \Rightarrow k) \rightarrow s_3$ (r3)

We assume that f, g and h denote server functions which all may produce named returns. The **def** and **subst** sections of the program defines the primary and substitute functions respectively, while the **rules** section defines the rules for function substitution for some named returns.

The notation $f \Rightarrow n$ denotes the event that the evaluation of a server functions f reduces to the named return n while $e{:}(f \Rightarrow n)$ denotes the event $(f \Rightarrow n)$ occurring while f is being evaluated as a component function of the expression e (i.e. within the context of e). We call the expression e the *substitution context* for $f \Rightarrow n$. The notation $e{:}(f \Rightarrow n) \rightarrow s$ denotes the rule stating that if the event $f \Rightarrow n$ occurs in

the substitution context e, the expression e will be substituted for by the expression s. The symbol **any** denotes either **any** function or **any** named return depending on textual context.

Substitute functions may themselves fail. Since the same exception handling mechanism applies to substitute functions, exception handling can be nested to arbitrary depth. In the above example we see that the subexpression e_1 of the substitute function s_1 has substitute function s_3 for the event $h \Rightarrow k$. If the execution of the substitute function itself fails, the further exception handling will be based on the named return that originally caused the attempted execution of the substitute.

When two different substitution rules for the same event have overlapping substitution contexts (i.e. where the one context is a subexpression of the other) a set of precedence rules for deterministically choosing between the substitution rules is applied. Generally the substitution rule with the smallest context is applied first. Within one context the substitution rule with the most specific named return has precedence. (i.e. $(f \Rightarrow n)$ before $(f \Rightarrow \textbf{any})$ before $(\textbf{any} \Rightarrow \textbf{any})$).

In the above example suppose the evaluation of f produces the named return n. Any of the substitution rules r_1 and r_2, can be applied. The precedence rules give the applications an explicit order

$$r_1 < r_2$$

If s_1 fails with the named return $(h \Rightarrow k)$ the application order is

$$r_1 < r_3 < r_2$$

If all the substitutes fail, T fails with the named return $(f \Rightarrow n)$.

When applying the substitution rule $e{:}(f \Rightarrow n) \rightarrow s_1$ (say), the potential durable effects of already executed component functions of e at the time of the event $f \Rightarrow n$, are (semantically) undone by performing compensating and/or abort actions, before the evaluation of s_1 is started. Hence a substitution context is atomic. The effect of the transaction will be the result of evaluating $T[s_1/e]$ rather than T. Given the above presedence rules, the potential different alternative effects of the transaction are in order of presedence

$$T < T\,[s_1/e] < T\,[s_2/e]$$

And as above, if s_1 fails with the named return $(h \Rightarrow k)$ the order becomes

$$T < T\,[s_1/e] < T\,[s_1[s_3/e_1], e] < T[s_2/e]$$

We do not present any details of the distributed exception handling in this paper. The interested reader is referred to [Eli90b] in which is presented a detailed abstract algorithm of the distributed exception handling of FRIL as an amendment to the transaction execution algorithm presented in Section 3.4. The algorithm is based on the set of parse trees of the global transaction program; one for the defining expression of the global transaction and one for each of the defined substitute functions.

6. Conclusion

An infrastructure for interoperable heterogeneous applications must support a wide variety of data models to allow for easy and flexible manipulation and combination of information stored in different heterogeneous systems. By considering entity types of data models as

abstract data types, the problem of achieving interoperability can be expressed as the problem of combining data conforming to different abstract data types.

Interoperability can be guaranteed through the provision of homogenized abstract interfaces (export schemata) and the availability of a common kernel set of primitive data types. The common kernel set of abstract data types must be supported by all component information systems as well as by the infrastructure itself and enables the exchange of structured values between information systems so that outputs from one system can be used as inputs to an other system, and outputs from different systems can be computationally combined.

This paper has presented an architecture for the interoperability infrastructure. Within this framework was presented the functional resource integration language FRIL with its two language components FIOL and SESSL as a linguistic tool conforming to the above approach. The specification of abstract interfaces (export schemata) is formulated in the abstract data type language SESSL while the functional language FIOL is used to formulate global operations combining operations from multiple abstract interfaces. The infrastructure also provides mechanisms for the evaluation of FIOL expressions as distributed flexible transactions. This mechanism has been named the EVAL service.

The presentation has focused on the support for important database concepts as object persistency and object update, and the novel concept of alternative transactions. Our goal has been to make FRIL mappable to other existing data models and data languages so that it can be used as a unified language format for achieving interoperability. A first study of making FRIL interfaces to relational databases seems very promising [Eli90a]. A similar study of object-oriented databases is under way. These studies will give us valuable feed-back for the further development of FRIL. For the FIOL language component and the associated EVAL service, the integration of the flexible transaction model is well under way as reported in this paper. However, several issues need to be further studied. These are mainly related to requirements for program transformation and decomposition strategies, and access control. In relation to SESSL, details of mechanisms for publication (export) and import of schemata also need to be studied.

For our first prototype we consider ANSA testbench as our implementation basis [ANS89a] since this system contains many of the infrastructure functions that are required for federations. A FRIL server generator to be used to generate FRIL server front-ends to relational database systems is under design and implementation. The server generator takes a given relational schema as input and produces a corresponding SESSL specification as well as code for mapping between FIOL and SQL. Thus our first experiments on application interoperability will be based on relational databases.

References

[ANS89a] ANSA, *Reference Manual, Volume C, Release 01.01*, April 1989.

[Atk84a] M. P. Atkinson and K. G. Kulkarni, "Experimenting with the Functional Data Model," pp. 311-338 in *Databases: Role and Structure, (Stocker, Gray, Atkinson eds.)*, Cambridge University Press (1984).

[Bac78a] J. Backus, "Can Programming languages be liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *CACM* **21**(8) (August 1978).

[Ber87a] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley (1987).

[Bun84a] P. Buneman and R. Nikhil, "The Functional Data Model and its Uses for Interaction with Databases," pp. 359-380 in *On Conceptual Data Modelling (eds: Brodie, Mylopoulos, Schmidt)*, Springer Verlag (1984).

[Bun90a] P. Buneman, "Functional programming and Databases," pp. 155-169 in *Research Topics in Functional Programming, (Turner, D. A., Ed.)*, Addison Wesly (1990).

[Cer84a] S. Ceri and G. Pelagatti, *Distributed Databases Principles and Systems*, Mc-Graw Hill (1984).

[Day89a] U. Dayal, "Queries and views in an object-oriened data model," pp. 80-102 in *Proc. 2nd Int. Workshop Database Programming Languages, (Hull, R., Morrison, R., Stemple, D., Eds.)*, Morgan Kaufmann (1989).

[Du89a] W. Du, A. K. Elmagarmid, Y. Leu, and S. Osterman, "Effects of Autonomy on Global Concurrency Control in Heterogeneous Distributed Systems," pp. 113-120 in *Proceedings of 2nd Int'l Conf. on Data and Knowledge Systems for Manufactoring and Engineering* (October 1989).

[Eli87a] F. Eliassen and J. Veijalainen, "An S-Transaction Definition Language and Excution Mechanism," Arbeitspapiere der GMD, GMD/Fokus Berlin (1987).

[Eli87b] F. Eliassen and J. Veijalainen, "Language Support for Multi-Database Transactions in a Cooperative, Autonomous Environment," pp. 277-281 in *Proc. IEEE TENCON'87*, Seoul (1987).

[Eli88a] F. Eliassen and J. Veijalainen, "A Functional Approach to Information System Interoperability," pp. 1121-1135 in *Research into Networks and Distributed Applications, R. Speth (editor)*, North Holland (1988).

[Eli89a] F. Eliassen, "FRIL Linguistic Support for Interoperable Information Systems," CSR 89-02, University of Tromsø (1989).

[Eli90a] F. Eliassen, *Notes on Making FRIL Interfaces to Relational Databases*, University of Tromsø (1990).

[Eli90b] F. Eliassen and R. Karlsen, "Distributed Exception Handling for Flexible Transactions," CSR 90-07, University of Tromsø (1990).

[Elm90a] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz, "A Multidatabase Transaction Model for Interbase," in *Proc. 16th VLDB* (1990).

[Fis87a] D. H. Fishman, "Iris: An Object-Oriened Database Management System," *ACM TOIS* **5**(1), pp. 48-69 (January 1987).

[Fut85a] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," pp. 52-66 in *Proc. ACM Conf. on Principles of Programming Languages* (1985).

[Gar83a] H. Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM TODS* **8**(2), pp. 186-213 (June 1983).

[Gli84a] V. Gligor and G. Luckenbaugh, "Interconnecting Heterogeneous Database Management Systems," *Computer* **17**(1) (1984).

[Gli86a] V. Gligor and R. Popescu-Zeletin, "Transaction Management in Distributed Heterogeneous Database Management Systems," *Information Systems* **11**(4) (1986).

[Gog87a] J. Goguen and J. Meseguer, "Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics," pp. 417-478 in *Research Directions in Object-Oriented Programming, (B. Shriver and P. Wegner eds.)*, The MIT Press (1987).

[Gut78a] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Types," *Acta Informatica*(10), pp. 27-52 (1978).

[Gut81a] J. V. Guttag, J. J. Horning, and J. Williams, "FP with Abstraction and Strong Typing," in *Proceedings Func. Prog. Lang. and Comp. Arch., ACM* (1981).

[Gut85a] J. V. Guttag, J. J. Horning, and J. Wing, "Larch in Five Easy Pieces," Digital System Research Center Report (1985).

[Hei85a] D. Heimbigner and D. McLeod, "A Federated Architecture for Information Management," *ACM Transactions on Office Information Systems* **3**(3), pp. 253-278 (July 1985).

[Kar91a] R. Karlsen, "A Flexible Transaction Model for Interoperable Information Systems," Ph.D. Thesis (in preparation), Dept. of Computer Science, University of Tromsø (1991).

[Kel85a] R. M. Keller and G. Lindstrom, "Approaching Distributed Database Implementations through Functional Programming Concepts," in *Proc. 5th Int'l Conf. on Distributed Computing Systems (IEEE)* (1985).

[Kim89a] W. Kim and F. H. Lochosky, *Object-Oriented Concepts, Databaseis, and Applications*, Addison Wesley (1989).

[Lis86a] B. Liskov and J. Guttag, *Abstractions and Specifications in Program Development*, MIT Press (1986).

[Lit86a] W. Litwin and A. Abdellatif, "Multidatabase Interoperability," *IEEE Computer* **19**(12), pp. 10-18 (December 1986).

[Shi81a] D. W. Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Transactions on Database Systems* **6**(1), pp. 140-173 (March 1981).

Merits of Language-Oriented Approaches for Constructing Distributed Systems

Uwe Baumgarten

University of Oldenburg, Germany

Uwe.Baumgarten@arbi.informatik.uni-oldenburg.de

Abstract

The need for high quality distributed systems is a great challenge the computer community is faced with. The quality attributes are, for instance security, reliability, and maintainability. In this paper a few questions will be answered in order to *(i)* gain a deeper understanding of the problems of distributed systems and to *(ii)* propose a methodology to overcome some of these problems. The questions which we are concerned with are:

> *What can we learn from current and recent approaches in* **Distributed Ada** *for the specification, design, construction, and implementation of* **distributed systems**?
>
> How can we implement distributed systems in an elegant and impressive way using a methodology called **concept's stepwise refinement**?

Some answers to these two questions will be given in this paper. Two ideas predominate in our approach for constructing distributed systems. The former idea states, that abstractions should be defined by means of languages. The latter idea says, that the implementation strategy should be guided by concept's stepwise refinement.

1. Introduction

One great challenge the computer community is faced with is the need for distributed systems with attributes, which express high quality. These attributes are, for instance security, reliability, and maintainability. The main subject of this paper is the construction of distributed systems. In this context the following questions should be answered.

> *What can we learn from current and recent approaches in* **Distributed Ada** *for the specification, design, construction, and implementation of* **distributed systems**?
>
> *Can* **concept's stepwise refinement** *be used in order to implement distributed systems in an elegant and impressive way?*

This work has been partially supported by Competence Center Informatik (CCI), Meppen, Germany.

What is the motivation for our own OlDiLa / VERITOS approach and what are the conclusions, which can be drawn from distributed Ada?

The answers to these questions will help to gain a deeper understanding of the problems in the area of constructing distributed systems. A methodology will be proposed, as well, in order to overcome some of the existing problems.

In our opinion *"specification, design, construction, and implementation of distributed systems should start at a very high level of abstraction"*. Therefore, the orientation towards high level languages is the best way to produce high quality distributed systems. Simplicity, generality, and expressive power of language concepts form a sound basis for the construction of distributed systems at a high level of abstraction. In addition, concept's stepwise refinement supports and improves the implementation.

Ada is a programming language which defines a powerful and expressive set of concepts. Ada can be used to specify and program various kinds of applications. Compilers and programming environments are implemented for a large number of computer systems. But only a few approaches exist in the area of distributed Ada. These approaches support programming of distributed Ada applications as well as their distributed execution in a distributed target hardware configuration.

Our recent investigations are carried out in the area of distributed systems, especially distributed operating systems. Programming languages for specifying distributed systems and distributed applications are in the center of interest.

On the one hand we are elaborating an integrated approach for specifying, designing, constructing, and implementing distributed systems. Work in this area has been done within the VERITOS project at the University of Oldenburg [Bau90a]. The project is still in progress and will be continued in the future. We have developed an experimental programming and specification language, called *OlDiLa – Oldenburg Distributed Language* – which is an essential part of our project.

On the other hand we have investigated in detail current approaches in *Distributed Ada* which made substantial efforts in the area of distributed execution of Ada program systems and which will clearly influence this area. A detailed comparative study has been elaborated in 1990 [Bau90b]. Some of its main results will be presented. Conclusions will be drawn with respect to specification, design, construction, and implementation of distributed systems in general.

Having both activities in mind this paper

(a) Presents some of the main results of the study in distributed Ada,

(b) Identifies significant problems in at least two areas, namely in

  (i) The definition of distributed solutions of application problems and in

  (ii) The distributed execution of these solutions,

(c) Discusses the solutions which are elaborated for Distributed Ada, and

(d) Presents conclusions drawn from the area of Distributed Ada, which have effects on the design and construction of distributed systems as well as on our solution including further levels of concept's stepwise refinement.

A summary of the topics, which are discussed in this paper, will be given in section 2 based upon the central Figure 1. As to these topics

some further details will be presented in section 3. These details include the motivation of the language-oriented approach within the VERITOS project. Our investigations about distributed Ada will be outlined. A few characteristic approaches will be mentioned. The concepts of *OlDiLa* will illustrate the objectives of our own project. Some examples will explain concept's stepwise refinement. Finally, some conclusions will be given in section 4.

2. Summary of Topics' Discussion

The design and implementation of high quality distributed systems is a time-consuming and difficult process. This experience has been published by Schlichting et al. [Pet87a] as well as many other projects. Therefore, the need for structuring this process becomes evident. Design and implementation can be divided and structured in many ways. In our opinion four important distinctions lead to an improvement.

- First of all different levels of abstraction have to be identified. This distinction leads to at least two levels of abstraction, namely abstractly distributed and physically distributed systems.
- Secondly an in accordance to the first, application programming, which can be done independent of any physical distribution, and distribution programming which takes into account the physical hardware configuration can be separated. This is a distinct separation of concerns.
- Thirdly, the distinction between partitioning and configuration follows the same direction. Units which define possible distribution units have to be identified. Their definition should not take into account the mapping onto physical nodes. By definition, a distribution unit is a unit, which is localized as an entire unit at a node. No possibility is supported to move parts or to place parts of a distribution unit on other nodes. The mapping onto physical nodes is done by configuration.
- The fourth distinction leads to an implementation method. Implementation decisions which are mutually correlated will be grouped together by means of concept's stepwise refinement. Concepts will be refined in order to reflect these decisions mainly by adding appropriate details.

Distributed systems have to be modeled at various levels of abstraction. Distributed systems with a high degree of abstraction are called **abstractly distributed systems**. They are positioned at the top in Figure 1. In contrast, distributed systems with little abstraction which are very close to the hardware are called **physically distributed systems**. They are positioned at the bottom in Figure 1.

In our approach the models which are used to specify abstractly distributed systems are language-oriented. The functionality of the whole system will be expressed by means of language concepts. As well, the functionality of components, the structures of components, and the cooperation among components including communication are specified likewise. Candidates for such languages are Ada and *OlDiLa* as well as languages like SR [And86a], Emerald [Jul88a], and LADY [Wyb90a].

A small number of abstractions can be found in physically distributed systems. Their characteristics are defined by the properties of the hardware configuration (processors, memory model, communication

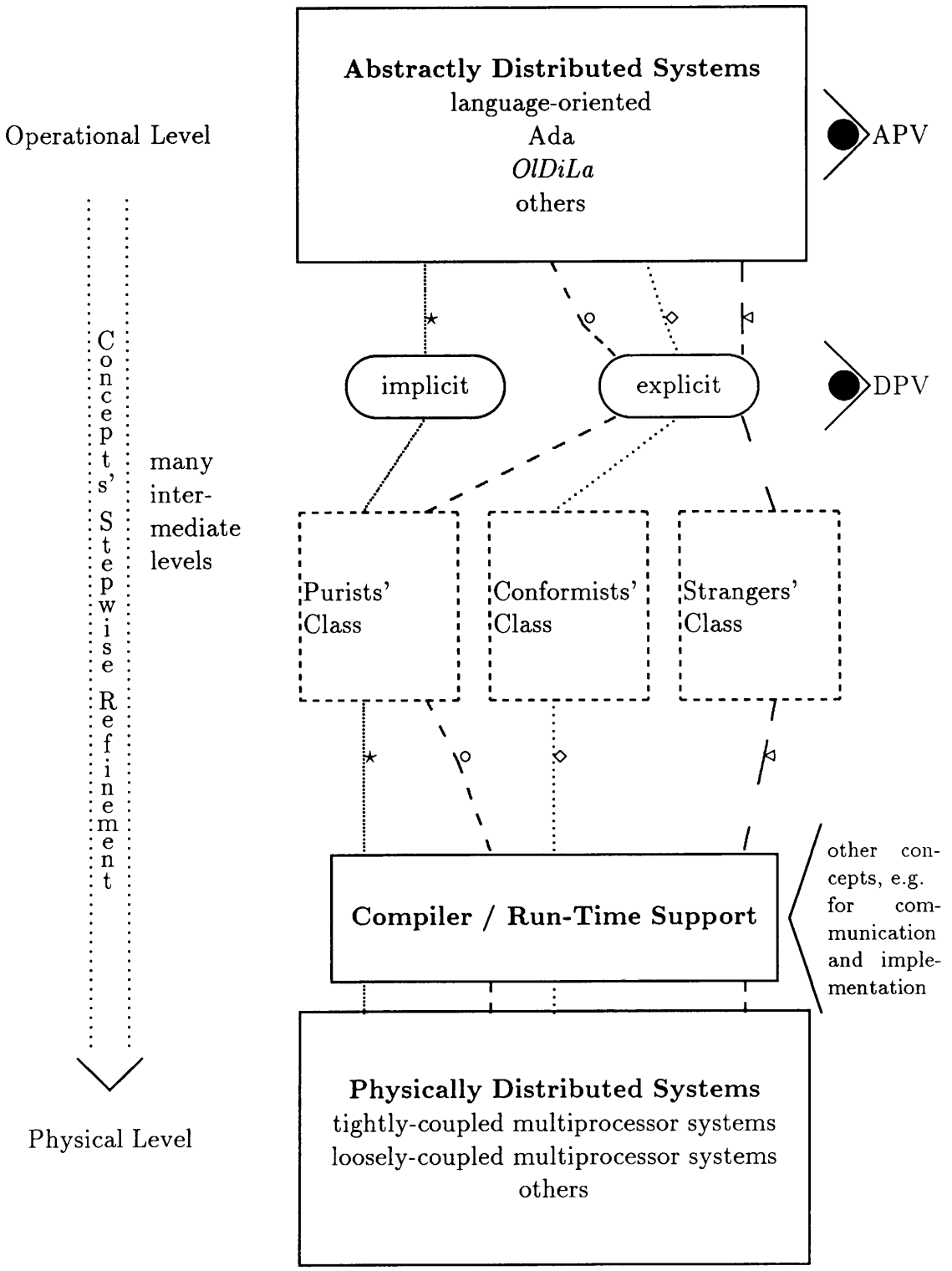


Figure 1: *Summary of topic's discussion*

system, network configuration). Physically distributed systems may be tightly-coupled multiprocessor systems, loosely-coupled computer systems, which may be multiprocessor systems, connected by a communication system, or other hardware configurations like embedded systems. Examples for the abstractions that can be found at this level are network protocols.

The central questions for the implementation of distributed systems are

> *What is the appropriate method for transforming abstractly distributed systems into physically distributed systems and what are the methods and tools which can be used to perform these transformations?*

In other words, the relationships and mappings which define the connections between systems at different levels of abstraction have to be stated. These relationships can be established by bringing together the models resp. the concepts at different levels. This is exactly the purpose of what we call **concept's stepwise refinement**. Concepts of a less abstract level result from refinement of the concepts of a more abstract level. Thus, the approach of **concept's stepwise refinement** bridges the gap between operational (i.e. language-oriented) levels models and concepts and physical levels models and concepts by defining intermediate levels. Examples are given below.

The level of abstraction, whose concepts define distributed systems in a language-oriented manner, is called *operational level* within the VERITOS project. Semantics of components are defined in an operational way. The bottom level is called physical level in order to indicate that the characteristics of the physical hardware configuration predominate. In Figure 1 concept's stepwise refinement is pictured left-hand.

The approach of concept's stepwise refinement is necessary and useful. Both can be motivated from a simple example.

Physically distribution is fully transparent to application programmers in abstractly distributed systems. These programmers should solve given problems in terms of concurrency and communication concepts. Abstractly distributed systems can be designed and constructed as sets of active and passive components. These sets of components can be structured in many ways. Components can cooperate by means of communication, operation invocation, or access of shared components.

Abstractly distributed systems which are constructed using Ada are defined as Ada programs, which consist of Ada packages, tasks, and library units. These components may be nested. They may cooperate, for instance, by means of the operation-oriented rendezvous concept.

Abstractly distributed systems have to be partitioned in order to prepare and integrate physical distribution. The resulting partitions (distribution units) have to be mapped onto the set of computer systems or processors. This mapping is called system configuration. The concepts of the operational level have to be refined with respect to distribution. This refinement of concepts can be performed as follows. The choice of suitable partitioning and configuration concepts may be constrained by the concepts of the operational level. These constraints result from the requirement, that semantics of systems resp. components may not be altered by transitions from one levels of abstraction to another. In other words, the properties of a system at the operational level – like functionality and structure – have to be maintained at each intermediate level. Therefore, concepts can be modified by adding appropriate distribution information.

Possible solutions in the area of distributed Ada are surveyed in [Bau90b]. The refinement of concepts may be very simple. Tasks can be selected as distribution units. Their mapping onto the nodes of a hardware configuration can be expressed using additional pragmas determining their locations. On the other hand, distribution units which are groups of Ada programming units may be introduced. The concept of virtual nodes [Atk88a] proceeds in this direction. Virtual nodes define distribution units. They are closely related to Ada library units. But several restrictions may be imposed on them. They are mapped onto the nodes of physical configurations.

Partitioning and configuration can be performed **implicitly** or **explicitly**. A distribution programmer's view will be added, if partitioning is done explicitly. In the Honeywell distributed Ada approach [Jha89a] a separate language, APPL, Ada Program Partitioning Language, is used for this purpose. Otherwise, pragmas and precompilers are used for the specification of distribution units. If the implicit way is favoured the whole work of partitioning and configuration has to be done by the compiler and the underlying run-time system. The responsibility and burden are taken away from the programmer. In Figure 1 implicit approaches as well as explicit approaches are represented.

Three different classes exit, into which all approaches can be divided. These classes will be described now.

Approaches belong to the first class – the **purists' class** – if at most concepts of the language at the operational level can be used to define distribution units. The repertoire of partitioning and configuration concepts is a subset of the repertoire of pure language concepts. The definition of distribution units can be done either explicitly or implicitly. In the latter case distribution is completely transparent to the application if these declarations are done implicitly. Abstractly distributed systems reflect the solution of problems from the application programmer's point of view. Physical distribution is not visible at this level. Partitioning and configuration of programs are left to the compilers and to the run-time systems. Partitioning and configuration concepts have to be defined in the frame of given language concepts. Examples for this class (cf. *-path and o-path) are presented in section 3.

Abstractly distributed systems can be partitioned in an explicit manner in agreement with the concepts of the language used at the operational level. These approaches belong to the second class – the **conformists' class**. Concepts for program partitioning and configuration are derived from the concepts of the original language using extensions or restrictions. They are conform with these language concepts.

Further strange concepts, which are not included in the language, of partitioning and communication can be used additionally. The approaches which define an extended repertory of concepts belong to the third class – the **strangers' class**. Strange concepts are added. Partitioning is done explicitly by application programmers, too.

Figure 1 outlines these classes. The application programmers' point of view (APV) and the distribution programmers' point of view (DPV) are added explicitly. Examples of the conformists' and the strangers' class will be given in section 3 (cf. ♦-path and Δ-path).

Functionality which is in general provided by run-time systems or operating systems has to be integrated in lower levels of abstraction in all approaches. Clearly, additional implementation concepts (like message passing, scheduling algorithms, etc) may be involved.

3. Motivation and Details

Some further details of the topics mentioned above will be presented in this section.

3.1. Motivation

In the introduction we stated, that

> *Specification, design, construction, and implementation of distributed systems have to start at a very high level of abstraction.*

This statement includes many aspects, which will be separated now.

Distributed systems are very complex systems. Therefore, the amount of work which must be done to solve the whole set of problems has to be reduced. One possible way consists in the separation of the set of problems into different problem areas. Problems of these areas can be solved independently. The specification of distinct levels of abstraction is an appropriate approach.

Programmers of distributed applications are interested in problem solutions. They specify their solutions, for instance, in terms of functions, relations, effects and properties. They usually are not interested in the way systems are producing these results. Therefore, very high abstractions are needed.

In addition, a methodology is needed which guides the implementation of distributed systems. In VERITOS this methodology is called concept's stepwise refinement. Abstractly distributed systems are the starting points within this method. Concepts are refined step by step. Finally, the method leads to an implementation at the level of physical distribution. This approach can be described as a **top-down** approach. In contrast, distributed systems can be built from scratch following a bottom-up approach. Many existing distributed systems follow this approach. The starting points of these approaches are hardware configurations with fixed properties. These properties, which include primitive mechanisms, are used to provide higher abstractions. Kernel-oriented approaches define low level abstractions for basic memory management, process management, scheduling, communication among processes, and security. At a second level of abstraction several servers can be constructed based upon the kernel in order to provide functionality of a higher level. Mach [Bla90a], Chorus [Roz89a], and Amoeba [Tan90a] are well-known kernels for distributed operating systems. The mechanisms which are provided by these bottom-up approaches are very flexible. Nevertheless, they neglect higher level attributes and properties, which, we believe, cannot be implemented in a provable and efficient way, if no provisions are made at the very first levels. Examples are context-dependent migration of components, dependability, and many other properties, which heavily rely on the semantics of applications. Therefore, we favour a top-down approach.

The abstractions chosen have to be integrated in a language in order to facilitate the specification of distributed systems and distributed applications. SR, Emerald, LADY, and Ada are examples for the language-oriented construction of distributed systems. The language SR is used for programming distributed systems from the systems programmers' point of view. The distributed operating system Saguaro [And87a] is an example which is implemented by means of SR. Emerald is a

language for programming distributed applications. Many distributed real-time systems are programmed using Ada. Examples are given in [Nie90a] and [Bur90a].

In the preceding paragraphs we have elaborated the advantages of top-down approaches. Nielsen presents results in the same direction. There, the two terms "top-down" and "bottom-up" are replaced by "hardware-first" and "software-first". He favours a top-down approach, too, and says in [Nie90a] page 167:

> *Whenever the hardware dictates the software design approach, we should expect a suboptimal correspondence of the software solution for the problem specification.*

So far, the motivation for a top-down language-oriented approach is given. Additionally, further pros and cons should be mentioned. Pros are

(a) That formal models can be applied to model different aspects and attributes,

(b) The possibility of a clear formal definition of semantics of distributed systems,

(c) The possibility of verification of implementation steps and the assurance of various properties,

(d) The integration of software engineering methods and tools,

(e) The modeling of the whole system and its implementation by a repertoire of homogeneous concepts, and

(f) Typing as very useful mechanism.

Cons may be

(a) That special characteristics of special purpose systems (like embedded systems and real-time systems) are reflected inadequately or

(b) Performance flaws.

3.2. Results of the Investigations in Distributed Ada

Our investigations in the area of distributed Ada will be outlined shortly. A few characteristic approaches will be mentioned.

In these investigations about distributed Ada we have used another classification scheme which is slightly different from that used in the section above. The counterpart of the purist's class, which we have defined in section 2, is the first class in the investigations about distributed Ada. But the first class shows some differences to the purists' class.

Partitioning and configuration are completely transparent to the applications for those approaches of distributed Ada which belong to the first class. The first class is a subset of the purists' class including only those approaches, which use implicit declarations of distribution units. Ada tasks are preferred as distribution units. Examples are the NYU Ada/Ed Project [Dew89a], the MUMS approach [Ard89a], the Encore approach [Ric89a], and Honeywell Distributed Ada [Jha89a]. The *-path in Figure 1 describes distributed Ada systems like NYU Ada/Ed, MUMS, and Encore. They are implemented on a shared memory architecture, where shared memory is either physically present based upon a tightly-coupled multiprocessor configuration (NYU Ada/Ed and Encore) or emulated by the operating system (MUMS). Physical distribution is transparent to the application programmer. The tasks are the distribution units. They are prepared for separate execution by the

compiler and they are represented in the shared (virtual) memory. The tasks are dispatched by the operating system which is supported by the Ada run-time system. The Honeywell Distributed Ada approach follows the o-path in Figure 1. Many different kinds of Ada program units are explicitly eligible for distribution units.

Partitioning is done in an explicit manner in agreement with the concepts of Ada in those systems which belong to class 2. Virtual nodes [Vol89a] are the distribution units. Examples are DIADEM [Atk88a], ASPECT-YDA [Hut89a], and Michigan Ada [Vol89b]. They follow the ♦-path in Figure 1.

Further concepts for partitioning and communication will be used in addition to the concepts of Ada for those approaches belonging to class 3. Partitioning is done explicitly by application programmers. Entire Ada programs are the distribution units. In this sense, distribution units are specified using concepts, which do not belong to Ada's repertoire of concepts (they are strange), because entire Ada programs cannot be parts of other Ada programs. These distribution units can cooperate by means of interface modules which, among other things, provide facilities for communication. Examples are DARK [Sco90a], Alsys Transputer [Dob89a], Chorus [Gui89a], and RTAda/OS [Rab89a]. They follow the Δ-path in Figure 1.

3.3. Concepts of VERITOS and *OlDiLa*

The concepts of *OlDiLa* will be described shortly in this section. They illustrate the main ideas of the VERITOS project. The highest level of abstraction is, at the moment, expressed in terms of programming language concepts. This is done, because distributed operating systems are of primary interest in our current activities. Nevertheless, functional abstractions and abstractions describing the behaviour of distributed operating systems as reactive systems [Pnu86a] are in consideration. *OlDiLa* is an experimental programming language for the specification of abstractly distributed systems. Distributed systems, which are specified by means of *OlDiLa*, are *sets of active and passive components*. Both kinds of components are objects referring to the principle of object-orientation (cf. [Mey88a]) and each of them offers a set of operations. Simple objects like constants or variables of elementary (e.g. integer) or structured types (e.g. arrays or records) are not handled as separate self-reliant components; they exist local to components only.

Components are used (by other components) following the *concept of operation invocation*. Passive objects are used by means of procedure calls, whereas active components, which are capable of communication, can communicate among each other following the operation-oriented rendezvous concept. At this level of abstraction, the client-server model does not predominate the cooperation concepts. The set of active components is divided into two sorts. Active components with communication facilities are mentioned above. They are called c-actors in *OlDiLa*. Active components without communication facilities cannot be controlled by other active components. They define a single ("mono") operation and are called m-actors in *OlDiLa*. These m-actors can be influenced by other components only on instantiation and by means of parameter passing to these m-actors at the moments of their initial and their final synchronization (similar to a procedure call).

All components are specified following the *class concept* in such a way that strong type checking is possible.

The set of components which constitutes a distributed system is structured in different ways. These structures form important properties of distributed systems in the VERITOS approach. Both the development methodology and the development process are built around these structures. Therefore, they will be sketched below.

The general concept behind all structures is the *concept of nesting*. Components may be nested along their definitional dependencies, according to their executional dependencies in a sequential or a parallel manner, along their localities and lifespans.

In accordance to the class concept components are defined using generators, which are comparable to classes or types in other languages. Generators define those properties of components which are common to all instances of that class represented by the generator. Generators themselves must be defined local to components. A dependency is established between any component instantiated referring to the generator and the component which defines this generator. In this way the definitional dependencies determine the *definitional structure* of components. This structure is the base for the visibility, usability and further dynamical development.

Executional dependencies are established by means of invocation of operations of passive components or by cooperation with c-actors. The same dependencies result from instantiating active components (c-actors and m-actors). They determine the *executional structure* with its sequential, its parallel and its communicational parts. Synchronization of active components is another aspect in this area which is mentioned earlier for example in the context of parameter passing for m-actors.

Not only generators but also components may be defined in a manner that they are local to other components. By means of locality usage from outside can be hidden or restricted. The resulting dependency is called the *locality structure* of components.

The phase between instantiation (start) and deletion (termination is a necessary precondition) is the lifespan of a component. The lifespans of many components may depend on each other caused by the principle of nesting. In this way the *lifespan structure* is determined.

3.4. Concept's Stepwise Refinement

In section 2 concept's stepwise refinement has been introduced. In the following a level of abstraction which has been developed within the VERITOS project will be outlined. Additionally, the method itself will be illustrated in this section by means of a simple example.

The *hardware independent level* is an example of concept's stepwise refinement. *OlDiLa* components are represented in a single abstract storage. Every component of the operational level corresponds to a group of typed segments at the hardware independent level. Activities are summarized in a coarsened manner in form of classes of equivalent processors [Eck90a].

Another step towards further levels in concept's stepwise refinement may deal with cooperation between components. At the operational level the cooperation among components is determined by operation-oriented communication concepts (cf. Ada's rendezvous concept), whereas message-oriented communication concepts may be used at the level of run-time support by the operating system.

OlDiLa components are defined at the operational level. There, they belong to abstractly distributed systems. Therefore, they cannot have a

location attribute, which expresses their physical location. The questions are now:

> *Can* OlDiLa *components have a location? Which is the level of abstraction expressing the location of a component?*

In the simplest solution an additional level of abstraction which just adds location information to each *OlDiLa* component can result from refinement.

Another solution can be based upon the hardware independent level which is mentioned above. *OlDiLa* components are represented in form of sets of typed segments in a single non-distributed storage. Now, this level may be refined, again, in order to add location (distribution) information. At least two alternative solutions exist. In the first solution an *OlDiLa* component stays a distribution units. The consequence is, that the segments of the group which represent this component are located in the same way at the same node of the hardware configuration. In the second solution the segments may be located individually and independent form each other. The result in this second solution is that segments which belong to the representation of the same *OlDiLa* component may have different locations.

Both solutions are more complicated, if the locality structures, which relate many *OlDiLa* components, are taken into consideration. Are those components which are local to a distinct component integral part of that distribution unit to which the distinct component belongs? Possible solutions for distribution can be simplified, if *OlDiLa* components and all of their local components must have the same location.

4. Conclusions

We have outlined our opinion that

> *Specification, design, construction, and implementation of distributed systems should start at a very high level of abstraction.*

Physical distribution should be as transparent as possible. Useful support for the implementation is provided by concept's stepwise refinement. Programming languages are important tools for specifying various kinds of abstractions. But their importance will increase in the near future, because they are valuable tools for enforcing quality requirements.

5. Acknowledgements

The author wishes to thank the members of the "systems architecture group", especially C. Eckert and M. Lange, for many helpful suggestions and for reading earlier drafts of this paper.

This work has been partially supported by Competence Center Informatik (CCI), Meppen, Germany.

References

[And86a] G. R. Andrews and R. A. Olson, "The evolution of the SR language," pp. 133-149 in *Distributed Computing* (1986).

[And87a] G. R. Andrews, R. D. Schlichting, R. Hayes, and T.D. Purdin, "The Design of the Saguaro Distributed Operating System," pp. 104-118 in *IEEE Transactions on Software Engineering* (January 1987).

[Ard89a] A. Ardo and L. Lundberg, "The MUMS Multiprocessor Ada Project," in *Distributed Ada 1989, Proceedings of the Symposium* (1989).

[Atk88a] C. Atkinson, T. Moreton , and A. Natali, "Ada for Distributed Systems," in *Cambridge University Press* (1988).

[Bau90a] U. Baumgarten, "Veritos Distributed Operating System Project – An Overview," pp. Springer Verlag in *Rosenberg, J, Keedy, J.L., Security and Persistence. Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information,*, Bremen (1990).

[Bau90b] U. Baumgarten and P. P. Spies, "Ansaetze zu verteiltem Ada und ihre Beitraege zur Konstruktion von Verteilten Systemen," in *Interne Berichte, Fachbereich Informatik, Universitaet Oldenburg, Bericht SA/90/2* (November 1990).

[Bla90a] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," pp. 35-43 in *IEEE Computer* (May 1990).

[Bur90a] A. Burns and A. Wellings, "Real-Time Systems and their Programming Languages," in *Addison-Wesley Publishing Company, Wokingham, England* (1990).

[Dew89a] R. Dewar, S. Flynn, E. Schonberg, and N. Shulman, "Distributed Ada on Shared Memory Multiprocessors," pp. 229-241 in *Distributed Ada 1989, Proceedings of the Symposium* (1989).

[Dob89a] B. J. Dobbing and I. C. Caldwell, "A Pragmatic Approach to Distributing Ada for Transputers," in *Distributed Ada 1989, Proceedings of the Symposium* (1989).

[Eck90a] C. Eckert, "Homogeneous Memory-Management in the Context of the VERITOS-Project," pp. Springer Verlag in *Rosenberg, J, Keedy, J.L., Security and Persistence. Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information,*, Bremen (1990).

[Gui89a] M. Guillemont, "Chorus: A Support for Distributed and Reconfigurable Ada Software," in *Technical report, Chorus Systemes, CS/TR-89.40.2, Presented at the ESA Workshop on Communication Networks and Distributed Operating Systems within the Space Environment, 24-26 October 1989* (1989).

[Hut89a] A. D. Hutcheon and A. J. Wellings, "The York Distributed Ada Project," in *Distributed Ada 1989, Proceedings of the Symposium* (1989).

[Jha89a] R. Jha, M. Kamrad, and D. T. Cornhill, "Ada Program Partitioning Language: A Notation for Distributed Ada Programs," pp. 271-280 in *IEEE-TSE* (March 1989).

[Jul88a] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," pp. 109-133 in *ACM TOCS* (February 1988).

[Mey88a] Bertrand Meyer, *Object-oriented Software Construction*, 1988.

[Nie90a] K. Nielson, *Ada in Distributed Real-Time Systems*, 1990.

[Pet87a] L. L. Peterson, N. C. Hutchinson, R. A. Olsson, R. D. Schlichting, and G. R. Andrews, "Observations on Building Distributed Languages and Systems," in *Experiences with Distributed Systems, International Workshop,*, Kaiserslautern, FRG (September 1987).

[Pnu86a] A. Pnueli, "Specification and Development of Reactive Systems," pp. 845-858 in *Information Processing 86, Proceedings of the IFIP 10th World Computer Congress*, Dublin, Ireland (September 1-5, 1986).

[Rab89a] H. M. Rabbie and D. A. Nelson-Gal, "An Operating System for Real-Time Ada," in *Proceedings, TRI-Ada'89, David L. Lawrence Convention Center – Pittsburgh, PA* (23-26 October 1989).

[Ric89a] V. F. Rich, "Parallel Ada for Symmetrical Multiprocessors," in *Distributed Ada 1989, Proceedings of the Symposium* (1989).

[Roz89a] M. Rozier, "CHORUS Distributed Operating System," in *Technical report, Chorus Systemes, Technical Report, CS/TR-88-7.8* (February 1989).

[Sco90a] R. Van Scoy, J. Bamberger, and R. Firth, "A Detailed View Of DARK," pp. 68-83 in *ACM, Ada LETTERS* (July/August 1990).

[Tan90a] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, and G. van Rossum, "Experiences with the Amoeba Distributed Operating System," pp. 46-63 in *Communications of the ACM* (December 1990).

[Vol89a] R. A. Volz, "Virtual Nodes and Units of Distribution for Distributed Ada," pp. 85-96 in *Ada LETTERS, A Special Edition from SIGAda, The ACM Special Interest Group on Ada, Spring 1990, Third International Workshop on Real-Time Ada Issues, Nemacolin Woodlands, Farmington, PA* (26-29 June 1989).

[Vol89b] R. A. Volz, P. Krishnan, and R. Theriault, "Distributed Ada – A Case Study," pp. 17-59 in *Distributed Ada 1989, Proceedings of the Symposium* (1989).

[Wyb90a] D. Wybranietz, "Multicast-Kommunikation in verteilten Systemen," in *Springer-Verlag, Berlin, Informatik-Fachberichte 242* (1990).

Domain-based Support for Service Administration and Server Selection

V. Tschammer

GMD FOKUS, Berlin, Germany
Tschammer@fokus.berlin.gmd.dbp.de

Abstract

Concepts and related support services for co-operation in an open services environment are described. The environment will develop as a consequence of the installation of wide-spread fiber optical networks and future communication systems and the development of open distributed processing standards. A domain concept is introduced for structuring the environment and applications according to organisational and operational aspects. A service concept provides means for specification and administration of service types, services, and servers. Server identification and selection is a basic functionality considered for delegation to a supporting environment of co-operating open systems.

We describe a domain-oriented concept and an information system for the support of service administration and server selection and its implementation on the basis of standard directory services.

1. Introduction

Standardisation bodies and research projects currently work on concepts and architectures for an Open Services Environment which is characterised by a large, heterogeneous number of service users and service providers connected via future communication systems. ISO is working on a Basic Reference Model and related concepts for Open Distributed Processing (ODP). CCITT is developing a Distributed Application Framework (DAF) and ECMA a Support Environment for ODP (SE-ODP). The European Community sponsors work on open services architectures within the framework of RACE and ESPRIT, other research projects elaborate on specific aspects such as trading or domain management.

The environment considered is open and unlimited. It includes numerous components belonging to a variety of interconnected subnets and organisational domains. Mostly, the components will be installed, operated, and used according to local requirements and, therefore, will be characterised by design heterogeneity, different administration and management policies, and different operational characteristics such as availability, performance, and load. In contrast, distributed applica-

tions require that groups of components co-operate according to a common plan, co-ordinate their activities, share common resources, and adhere to common concepts and agreements. For structuring the open environment and co-operating groups of components, the domain concept has been introduced. [Slo89a] provides an overview on recent approaches. Generally, domains are a means for grouping components according to common characteristics or for particular purposes.

Co-operating components must have a common view and understanding of the functionality which can be delegated to or which is offered by other components. Generally, the term "service" is used to describe the functionality offered or requested. A service administration provides means for specification and administration of service types, service offers, and service requests, as well as for administration and management of service requestors and service providers.

The open environment may contain a large number of service offers of a given type, differing only in certain service attribute values or properties of the service providers. Basically, each service request may be directed to any of these offers, provided that the attribute values requested match the values offered, and the quality and cost of the service are acceptable. The selection of suitable service offers for a given request is one of the basic functionalities which are considered for delegation to support services.

We describe a domain-oriented concept and an information system for the support of service administration and server selection and its implementation on the basis of standard directory services.

The concepts and implementation guidelines were developed within the framework of BERCIM, a project sponsored by BERKOM, the broadband-ISDN research and development initiative of the German PTT. The approach can be distinguished from related work by the combination of the following design principles: integration of autonomous components, inclusion of dynamic server properties, integration of standard directory and management services, use of broadband-ISDN communication facilities, and an implementation in a heterogeneous, multi-organisational environment.

2. The Open Services Environment

An open services environment encompasses a large community of users and components which are interconnected via advanced communication facilities. Applications in such an environment usually arise from a loose coupling of existing entities rather than being implemented by dedicated components. Distribution is less a design principle than an outcome of geographical and organisational conditions. The user community and the work are distributed and, therefore, the applications must be distributed, too. Most of the components have been designed for local purposes. They have developed separately and independently within their local environment and thus have specific properties. Local authority and local tasks influence the characteristics and the behaviour of the components and must be described and handled properly if the components are to be integrated in common activities.

The following design principles support the integration of distributed applications from existing components by combining the local interests and tasks with the requirements of global co-operation [Tsc90a]:

1. A methodology for identifying the structure of distributed applications, i.e. which functions are distributed in the application, which services may be received from existing components, which supporting system services are available, and which services must be newly developed.
2. A methodology for describing the context of such applications so that the designer can determine which organisational, administrative, and operational constraints exist, for example access restrictions, processing and communication costs, hardware constraints, etc.
3. A methodology for matching the interests of component owners to the interests of application owners, so that configurations can be constrained by application development goals, e.g. performance, fault tolerance, security, and by local operational goals, e.g. local tasks and loads, privacy of resources, etc.
4. A methodology for defining relations between components in terms of when they are bound together, and which support service must apply to the selection of servers, server invocation, error recovery, etc. That means that the control thread and the degree of supervision of the activity flow from one component to another must be specifiable.
5. The removal of directional constraints between different component classes as regards assumptions of who is a server and who is a client. In principle, each component may adopt the role of a client within one interaction and the role of a server within another, i.e. all components may be servers of each other.

Most of the design principles developed so far build on the concept of "roles" and "services" to describe the different parts which components play within joint, co-ordinated activities, and the functionality which is offered and used in co-operations.

A role designates a specific part within the overall activity. It implies dedicated rights and duties and requires particular interactions with others. The definition of roles within a distributed application originates from the application structure and context. The assignment of users and components to roles is constrained by the application context and the interests of application and component owners, i.e. the organisational and operational environment. Human users or managers of applications usually occupy specific roles or positions within their organisations and acquire specific authority as occupants of such positions. We regard humans to be the initial source of activity within an application and assume that they are ultimately responsible for the activities initiated. Usually, humans are represented by automated agents, i.e. dedicated components, which perform roles on behalf and under control of the humans. The agents then acquire the specific rights and duties of the humans they represent, such as managers, administrators, service users and service providers.

Components occupying a specific role have to perform dedicated functions. From the designer's point of view these functions have interrelationships with the functions of other components in the course of the overall activity. From a component's point of view, from which the global context is hidden, it is important to know about which assistance from others can be expected and which functionality can be delegated to others. The term "service" is often used as an abstract notation of a component's view of a functionality which another component has made available for external use. There must be a common understanding of the services offered and requested within an application, and

common interfaces and protocols for the use of these services must be defined. An adequate service administration must provide for common agreements on services, for information about existing service offers and service providers, and for the adequate handling of the organisational context of interactions between service users and providers.

3. The Domain Concept

The following domain concept is an adaptation of the principles introduced by [Slo89a] to our scheme of structuring an open services environment. Generally, a domain is a collection of entities which have been explicitly grouped together by system administrators for particular purposes. Currently, entities may be other domains, components, and services. Other entities may be introduced if necessary. A domain has at least one attribute, called its "member set", which defines the set of member entities. The minimum representation of a member entity in a domain is its unique identifier which can be used to locate the entity. Additional entries for entities may be defined. An entity is referred to as a member of a domain if its identifier is a member of the domain's member set. In addition to the "member set" attribute, domains may have other attributes which support the positioning of domains within the domain hierarchy or which describe characteristics common to all members or common management policies. Examples are manager-domains which group managers or administrators, service-domains which contain abstract or concrete services, and definition or search areas which group components for service administration or server selection. Another useful attribute of a domain or component is the "domain set". It defines the set of domains an entity is direct member of. We distinguish between two types of domains: hierarchical and federal domains.

Hierarchical domains form a domain hierarchy according to long term organisational structures, like companies, sites, departements, and their computing environment. Hierarchical domains have distinctive identifiers and may have additional attributes which reflect organisational aspects and which support service administration and server selection mechanisms. As a hierarchical domain reflects a dedicated organisational structure, the domain may exist independent of the existence of its elements as long as the structure remains unchanged. Hierarchical domains are basically long living because they change only when the organisational structure changes. Changing this structure requires dedicated authority. It is granted by trusted components to particular components called "domain managers" which are responsible for specific subtrees of the domain hierarchy. A hierarchical domain is defined as a set of components and/or other hierarchical domains. The latter are called subdomains of the domain. The position of a hierarchical domain is defined by its superdomain which the domain is direct member of, and by its subdomains.

Each entity is assigned to a dedicated "home domain", i.e. a specific hierarchical domain which allows to locate the entity unambiguously within the domain hierarchy. Each entity must be member of one and only one hierarchical domain, i.e. its home domain. An entity is created and destroyed in its home domain. After creation, an entity is present, may perform its roles in distributed applications, and may become a member of federal domains. After destruction the entity does not further exist and consequently, it must vanish not only from its home domain but automatically from all federal domains as well.

Federal domains reflect groups of entities with common attributes and/or management policy, independent of the entities' membership in hierarchical domains. Their member sets may be grouped into further federal subset domains. Federal domains have distinctive identifiers and may have attributes that support the administration and server selection mechanisms. A federal domain may exist independent of the existence of its elements. They are created and destroyed at administrators' need and will. This requires dedicated authority that is granted by trusted components to particular domain manager sets. Adding members to and removing members from a federal domain is a typical domain manager activity.

4. The Service Concept

Service use in an open services environment requires a common understanding about the syntax and the semantics of a service between the interacting entities. In our approach, that is achieved by means of the following service concept. It includes specifications of abstract services, and concrete service offers and service requests.

The formal specification of a service is called "abstract service". If we take an object-oriented view, abstract services are mapped on types. The abstract service or service type defines what can be used by other components, i.e. the service semantics, and how it can be used, i.e. the service interface. The specification includes a service name, a set of logically interrelated operations, a list of attributes and a list of pre- and post-conditions for each operation. Attributes are typed, and default values, limits, and ranges are given. An explanation of the functionality represented by the service type is added in plain text so that human users and application designers can be informed about the functionality available.

The implementations of the service specification are called "concrete services". In an object-oriented approach, concrete services could be mapped on classes. All concrete services implementing a particular abstract service have the same interface but their internal structure and implementation may vary.

Associated with an abstract service is a specific federal domain, called the definition area of the service. Its member set includes all those components which have access to the information about the abstract service and, therefore, can offer or request concrete services. These concrete service offers and requests are characterised by specific values of attributes and links to the service offering or service requesting component, respectively. Components have specific properties such as location, load, and status, which define the context of the concrete service and its qualities.

Associated with each concrete service offer is a specific federal domain, called "scope", which defines the group of components to which the service is offered. Associated with each concrete service request is another specific federal domain, called "search area", which delimits the search for suitable offers within the open environment.

If the service administration uses subset domains of the definition area to define the scopes and the search areas of service offers and requests, it supports the fact that both, the requesting and the offering component have the same understanding of the service semantics and each interacting entity can be sure that the partner which it has been bound to uses the interface properly.

Service use across the borders of service definition areas is not possible with the current implementation of the service concept. In advanced structures, however, negotiation between service administrators of different definition areas may achieve a common understanding even across these borders. But this is for further study.

5. An Information System for the Support of Service Administration and Server Selection

We classify the services which support the interactions between components of an open services environment into information handling and decision making services. This does not necessarily anticipate that both categories must be implemented by separate components. However, the separation of concerns has certain advantages:

1. The system is more flexible because different decision making entities may access and use the information services independent of each other.
2. The configuration of the information system, i.e. the distribution of the storage and processing of different types of information between information sources, information sinks, and third parties may be optimised independent of the distribution of decision making and control [Wil89a].
3. The implementation of the information services via standardised support services such as directories is facilitated.

5.1. Information Service Types

The information system administers information about domains, abstract services, concrete services, and components.

The information about domains reflects the structure of hierarchical and federal domains. Domain entries contain the name and type of the domain and type specific attributes. The name of a domain must be unique within the organisational hierarchy. This is achieved by the managers creating the domain. Mandatory attributes of hierarchical domains are the name of the home domain and the domain manager, and the member set. The home domain and the member set attribute define the position of the domain in the organisational hierarchy. The manager attribute provides a reference to the entity which is allowed to create and destroy entities within the domain, i.e. to add and remove elements to and from the member set. An optional attribute of hierarchical domains is the domain set. It defines the set of federal domains which the domain is member of. Federal domain entries must contain the domain manager set and the member set. The manager set represents the board of managers from different organisations which are allowed to add and remove entities to and from the member set and to create and destroy subset domains. Optional attributes of federal domains are the subset and the superset attribute. They include the names of the federal domains which are subsets of the domain, or where the domain is a subset of, respectively.

Operations on domain information allow the registration and deletion of hierarchical and federal domains and their elements and managers. Further operations provide information about domain attributes and allow modifications. Very useful for domain management, server selection and service administration are operations that check the member and domain set attribute or the domain manager attribute.

The information about abstract services contains service specifications together with their definition areas, and templates for service requests and offers. Operations on the information about abstract services allow the registration and deletion of services in or from a given hierarchical domain. Service requestors and service providers may use operations resulting in the delivery of the machine readable templates for correct service offers and requests. Other operations list services defined within a given domain, or provide a service manual, i.e. a description of the service readable by human users.

The information about concrete services contains entries for service offers together with information about the components that made the offers, i.e. the service providers. Each entry includes the service type implemented, the associated service offer template filled with actual attribute values, including the scope of the service offer, and a reference to the service provider. Operations on the information about concrete services allow the registration and deletion of service offers in or from the home domain. Further operations read and modify selected attribute values. A search operation supports a yellow pages service which lists all concrete services of a given service type that are offered within a given search area and satisfy requested attribute values. The result of this operation also includes a reference to the service provider associated with a concrete service.

The information about components includes entries for application and system components which take the role of service users and/ or service providers. Component entries contain the identifier and a list of mandatory and optional attributes. Entries must contain the home domain. Entries may contain the domain set, the managed object set, the set of service offers, and a set of static attribute values. The component name must be unique within the home domain. Together with the home domain name it identifies and localises the component unambiguously within the open environment and provides an address useable for contacting the component via a given communication service. The domain set defines the set of federal domains which the component is member of, the managed object set defines the set of domains managed by the component, the set of service offers defines the set of concrete services provided by the component.

Operations on the information about components allow the registration and deletion of a component within its home domain as well as in other federal domains. Name server operations provide identification and location services. Operations of the type "get value" provide attribute values. Further operations list the sets of domains, managed objects, or concrete services associated with the component. A useful variant of these is the "is member of" operation which checks whether a component is within the member set of a given domain.

5.2. Support for Service Administration

The information system is able to support a service administration scheme in a hierarchical, domain-structured environment where the static information is stored and administered by a specialised third party. Static in this context means that the rate of update operations is very much lower than the rate of read operations. The usefulness of such an approach has been discussed elsewhere, e.g. in [Lor90a]. The approach becomes even more attractive, if standard directory services are involved in the third-party's operation.

Our approach supports an administration scheme for domains, services, and components that can be installed at the enterprise level. An initial structure of <country><locality><organisation> is assumed. Such a structure is generally supported by common services like standard directories. In this basic structure our hierarchy of domains can be included, with hierarchical domains representing the specific organisation of enterprises, departements, and their computing environment. A typical structure that can be mapped on hierarchical domains and components is <institute><department><network> <node><process>.

The information system provides the information base for the service administration and domain concept described above. Its operations can be used to check the pre- and postconditions of domain management operations and to store the effects of these operations.

For example, the management operation

destroyHierarchicalDomain 'D' *inHomeDomain* 'X'

issued by manager 'M' has the preconditions:

- 'M' isManagerOf 'X'
- 'D' isMemberOf 'X'
- memberSetOf 'D' is "empty"
- domainSetOf 'D' is "empty" or "notEmpty"

These conditions can be checked by information system operations reading the "manager" and "member set" attributes of 'X', and the "member set" and "domain set" attributes of 'D'.

The effects of the "destroy" operation are stored by modifying the the "member set" attribute of 'X' and of all federal domains listed in 'D's "domain set" attribute.

Likewise the configuration of abstract and concrete services is supported. The information system operations can also be used to check pre- and postconditions of operations performed by service administrators and to store their effects.

For example, the administrator operation

create ServiceOffer 'E' *ofType* 'S' *inHomeDomain* 'Y' *withProvider* 'P'

issued by manager 'Q' has the preconditions:

- 'Q' isManagerOf 'Y'
- 'S' isAbstractService in 'P's homeDomain
- 'E' isCorrectServiceOffer of 'S'
- scopeOf 'E' isSubsetOf 'S's definitionArea
- 'E' isNotMemberOf 'Y'

These conditions can be checked by information system operations reading the "manager" and "member set" attributes of 'Y', the "member set" attribute of the "services-domain" of manager 'Q', the "definition area" and the "templates" attribute of 'S' and the "subset" attribute of the federal domain identified as the definition area of 'S'.

The effects of the "create" operation are stored by modifying the "member set" attribute of 'Y'.

In this way the information system keeps track of the configuration of domains, components, and services, and supports the constraints that reconfigurations are made by authorised managers only and that services are not offered and used outside their definition areas.

5.3. Support for Server Selection

The services provided by the information system are particularly useful for traders, mediators, and brokers. etc. These are specialised entities which support the association of service users and service providers. Depending on the intelligence of the support entity the following services with increasing functionality are provided:

1. A simple "name" service which locates a specified server and provides its home domain and communication address.
2. A "list" service which lists the service offers of a given type within a given search area which match the values of static attributes requested.
3. A "dispatcher" service which selects one suitable service offer according to its static attributes and given criteria, and identifies and locates the appropriate server.
4. A "mediator" service which preselects suitable service offers according to static attributes and given criteria, identifies and locates appropriate servers, updates the values of dynamic attributes, and selects one suitable service offer according to given criteria.
5. A "negotiation" service which identifies and locates providers of suitable service offers, negotiates service modalities and qualities, and makes an agreement with one selected server that associates it with the requestor.
6. An "invocation" service which invokes a service at a selected server according to the agreement made.

These support services have to operate at run time and, therefore, must perform effectively. With the aid of the information system they can rely on a preconfigured structure of domains, services, and components which maps "yellow pages" and "white pages" services on simple and effective "list" and "search" operations on the "member set" attributes.

So far, the information system administers only static and long term information, i.e. the configuration of domains, services, and components, and the values of static attributes. Dynamic information is kept at the information sources, i.e. the service providers themselves, and must be interrogated there. Therefore, the information system is able to support only the name, list, and dispatcher services directly, while more intelligent service use the system for preselection based on static attributes and for localisation of servers.

An example for the support provided is the operation

getSuitableServiceOffers ofType 'S' *inSearchArea* 'A' *forRequest* 'R'

invoked by a "list" service provider on behalf of client 'C'. It has the preconditions:

- 'S' isAbstractService in 'C's homeDomain
- 'R' isCorrectServiceRequestof 'S'
- searchArea 'A' isSubsetOf 'S's definitionArea

With the aid of information system operations these conditions can be checked already at 'C's creation time and, thus, need not effect the performance of the server selection and service invocation. The search for concrete services which match the requested attributes and the identification and location of the related service providers is achieved

by a simple sequence of "search" and "get attribute value" operations on the information base.

5.4. Information System Implementation

The application environment of the information system requires that the information about domains, abstract and concrete services, and service providers has global significance and is globally available although the individual entries are stored and administered locally by autonomous agents. This is a typical requirement that is satisfied by a directory. Therefore, the information system was implemented by means of standard directory services.

The preconditions of such an implementation are the following: The information must be long living, required extensions to the directory scheme must be allowed by the standard, and the information system services must be mapped effectively on standard directory operations.

Most of the information considered is long living except that on dynamic service provider properties. Therefore, the information about domains and services, and most of the information about service providers, i.e. components, can be considered for inclusion in the directory scheme. The dynamic service provider properties are available from the components themselves via references stored in the directory.

The static information is mapped on directory object classes and attribute types. Mostly, existing classes and types are used. For the rest new classes and types are introduced within the constraints prescribed by the standard [Hal90a]. As far as possible, the new entries and their attributes are structured with respect to performance aspects. Off line configurations, like "register", "delete", and "modify entry", are less critical than run time "search" and "list" operations which influence the performance of the server selection mechanisms.

The entries for domains are mapped on a generic class "general domain" with two subclasses "hierarchical domain" and "federal domain". The generic class defines the "domain type" and the "domain manager". The domain type is hierarchical or federal, and the domain manager attribute contains the distinguished names of the managers. The hierarchical domains are mapped directly on directory nodes and subtrees. A federal domain which generally comprises different subtrees of the organisational hierarchy must explicitly define its member set. In the directory hierarchy, federal domains are administered as elements of hierarchical domains. Federal domains inherit the attributes of the generic classes "general domain" and "group of names". The latter is a general means of defining a set of directory entries under specific aspects. It must contain a common name and the member set, and may contain further descriptive attributes such as "definition area of service" or "scope of service offer".

For abstract services the new generic object class "service description" is introduced. The class must contain a common name, the definition area, and a set of descriptive attributes which specify the service interface, mandatory attributes, the templates for requests and offers, and the service description. The only attribute of specific interest is the "definition area" which contains a reference to the domain wherein the abstract service is defined.

For concrete service offers the new generic object class "service offer" is introduced. The class must contain a common name, the service name, the scope, the service provider, and a set of descriptive attributes according to the mandatory part of the service offer template. The ser-

vice name provides the reference to the abstract service and the service provider attribute provides the reference to the component offering the service. The scope contains the name of the domain wherein the offer is valid. It must be a subset domain of the definition area of the abstract service.

The object class "service provider" is a subclass of the existing generic class "application process". It must contain the references to its service offers. References to the managed object encapsulating the dynamic service provider information are currently implemented as optional attributes. Directly associated with the application process class is another existing generic class "application entity" which represents the communication with other OSI-applications. It must contain the presentation address and may contain additional information about locality, organisation, etc., i.e. the "domain set". Entries of the type application entity are arranged directly below the service provider entries within the directory hierarchy.

Service descriptions and federal domains may be arranged below "country", "locality", "organisation" or "organisational unit" in the directory scheme. Service provider entries may be arranged only below "organisation" or "organisational unit".

In this paper we do not further elaborate on naming schemes, reponsibilities for manipulations, etc. General information about these subjects is found in the directory standard, specific aspects are described in [Hal90a].

Access to the directory information is provided by directory user agents (DUAs) which are connected to directory acces points. Acces points are provided by directory system agents (DSAs). The set of cooperating DSAs represents the directory system. A DUA may get access via any access point provided that he is authorised to do so. The link between DUAs and DSAs is established by means of the "directory bind" operation and released by the "directory unbind", respectively.

The configuration of domains, services, and components performed by the domain and service administration is supported by the information system via operations that allow the manipulation of system entries under the given consistency constraints.

Creation, modification, and destruction of domains, services, and components is supported via "add entry", "modify", and "remove entry" operations which check the constraints on the membership and unique naming. "Read" operations on the domain manager attributes support authority checks. Reading domain entries allows checks of the existence of federal or hierarchical domains named as definition areas in service specifications. "Compare" operations on the templates in service specifications support checks of the correctnes of service offers. "Read" and "list" operations check the existence of a service provider for a given service offer, the membership of a service provider in the definition area of the related service, and the subset relation between the definition area of the service and the scope of a service offer or the search area of a service request.

The server selection is supported by the information system via search operations on available service offers in a predefined search area. These operations are mapped directly on "directory search" operations which search the directory for entries that have the attribute values requested. The consistency between the search area and the definition area of the requested service is checked via a directory read of the member set attribute of the domain representing the definition area. Identification and localisation of service providers is supported via

directory read operations on the application entity entry associated with a service provider.

6. Conclusion and Future Work

We have described an information system which supports domain and service administration in an open services environment. The system is part of a supporting environment for the co-operation of autonomous components which is developed as a work bench for the investigation and realisation of design concepts for domain-oriented structuring and management, service administration, server selection and service access control. The concepts allow the configuration of the environment by domain and service managers so that run time search and selection mechanisms perform more effectively.

The system serves as an information base which supports the actions of managers and administrators by operations that check pre- and postconditions, and that store the effects of these actions.

The separation of information handling from decision making and control entities allow the information base to be used my managers, administrators and run time support services independently. It further facilitates implementations via standard directory services.

Future work will primarily elaborate on the domain management and service administration scheme. We shall investigate how concepts developed for the delegation of authority to domain manager agents can be applied to our system. We shall further work on the concept of defining abstract components similar to the concept of abstract services. Such a concept would provide a framework for the implementation of concrete components. The component types will probably contain templates for attribute sets associated with the various component types, including mandatory and optional attributes, and static as well as dynamic attributes. A classification of attributes in static and dynamic attributes, for example, would facilitate the reference to classes of information services provided for handling the actual values of these attributes.

A domain-based access control scheme which operates within service definition areas and their subset domains is another subject for future work. It is to provide access rules between a group of service requestors belonging to a given scope and a group of service offers found in a specified search area.

Acknowledgements

The author thanks his colleagues U.W. Brandenburg, C. Burmester, K.P. Eckert, J. Hall, L. Strick, and M. Tschichholz for the very helpful discussions about the subjects described, and J. Sacher for the assistance in preparing the paper.

References

[Hal90a] J. Hall and M. Tschichholz , "Management fuer Verteilte Anwendungen im ISDN-B," in *BERKOM Report on BERMAN*, Berlin (August 1990).

[Lor90a] H. Lorin, "Application Development Software Engineering and Distributed Processing," *Computer Communications* **13**(1) (Jan 1990).

[Slo89a] M. S. Sloman and J. D. Moffett, "Domain Management for Distributed Systems," in *Proceedings of the IFIP Integrated Network Management Symposium*, Boston, MA (May 1989).

[Tsc90a] V. Tschammer, A. Wolisz, and J. Hall, "Support for Cooperation and Coherence in an Open Service Environment," pp. 222-228 in *Proceedings of the 2nd IEEE Workshop of the Future Trends of Distributed Computing Systems*, Cairo, Egypt (Sept 1990).

[Wil89a] C. E. Wills, "Locating Distributed Information," pp. 303-311 in *Proceedings of the INFOCOM* (1989).

The Evolution of the *Kerberos* Authentication Service

John T. Kohl

Project Athena, MIT, USA
jtkohl@mit.edu

Abstract

The Kerberos Authentication Service, developed at MIT, has been widely adopted by other organizations to eliminate the trusted-host problem in open networks. While a step up from traditional security in networked systems, Kerberos version 4 is not sufficiently flexible for some environments. These inflexibilities and the remedies introduced with the Kerberos version 5 are described.

Introduction

The Kerberos Authentication Service was originally developed at the Massachusetts Institute of Technology (MIT) for its own use to protect Project Athena's emerging network services. Versions 1 through 3 were internal development versions; protocol version 4 has achieved widespread use. However, it was designed for the envisioned use at MIT, and does not completely "fill the bill" for sites with different models of computer use and administration. Protocol version 5 incorporates new features suggested by experience with version 4 which make it useful in more situations. Version 5 was designed by Clifford Neuman of the University of Washington and the author, based in part upon input from many contributors familiar with version 4.

The first section of this paper briefly discusses the Kerberos model and basic protocol exchanges. Section 2 discusses the shortcomings of version 4. The third section reviews the new features found in version 5. Section 4 discusses the implementation of the new protocol and the compatibility support for converting existing applications from version 4 to version 5. The final section concludes with a status update and considerations of future work.

Terminology and Conventions

A *principal* is the basic entity which participates in network authentication exchanges. A principal usually represents a user or the instantiation of a network service on a particular host. Each principal is uniquely named by its *principal identifier*.

Systems like the Data Encryption Standard (DES) [Com77a] which use a single key for both encryption and decryption are referred to as *secret-key cryptosystems*. The keys used in such systems are called *secret* keys. Encryption systems like RSA [Riv78a] which use different keys for encryption and decryption are referred to as *public-key cryptosystems;* their encryption keys are referred to as *public* or *private* depending on whether the key is widely known or known only to a single entity.

Plaintext refers to an unencrypted message, while *ciphertext* refers to the encrypted form of the message.

In figures, encryption is denoted by showing the plaintext surrounded by curly braces ({ }) followed by a key (K) whose subscript denotes the principal(s) who possess or have access to that key. Thus, "foo" encrypted under c's key is $\{foo\}K_c$.

1. The *Kerberos* Model

Kerberos was developed to enable network applications to securely identify their peers. To achieve this, the initiating party (the client) conducts a three-party message exchange in order to send the contacted party (the server) an assurance of the client's identity. This assurance takes the form of a *ticket* (shown in figures as $T_{c,s}$) which identifies the client, and an *authenticator* (shown in figures as $A_{c,s}$) which serves to validate the use of that ticket and prevent an intruder from replaying the same ticket to the server in a future session. A ticket is only valid for a given time interval, called its *lifetime*. When the interval ends, the ticket expires; any later authentication exchanges would require a new ticket.

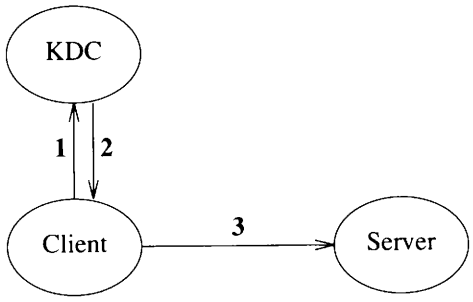
Tickets are issued by a trusted third party *Key Distribution Center* (KDC). As suggested by the Needham and Schroeder protocol [Nee78a], the KDC is trusted to hold in confidence secrets known to each client and server on the network (those secrets are established either out-of-band or through an encrypted channel). That trust forms the basis upon which clients and servers can believe the authenticity of the messages they receive.

Each installation establishes its own autonomously administered KDC. Each such installation comprises a *realm*. Most currently-operating sites have chosen realm names that parallel their names under the Internet domain name system (e.g. Project Athena's realm is `ATHENA.MIT.EDU`). Clients in separate realms can authenticate to each other if the administrators of those realms have previously arranged a shared secret.

1.1. The Initial Ticket Exchange

Figure 1 shows graphically the messages† exchanged in an application's authentication process. Both Kerberos versions 4 and 5 share the same framework for messages (although the encoding details of the messages differ). A typical application requires a three-message exchange with each server to establish authentication on its first invocation and a single message on subsequent invocations (client caching eliminates the need for the first two messages until the ticket expires).

† The figures actually show a simplified version of the messages for clarity. Other message fields are present in the actual messages, but are primarily for "bookkeeping" purposes not relevant to the present discussion.



1. Client → KDC: c, s
2. KDC → Client: $\{K_{c,s}\}K_c$,$\{T_{c,s}\}K_s$
3. Client → Server: $\{A_c\}K_{c,s}$,$\{T_{c,s}\}K_s$

(In version 4, message 2 is $\{K_{c,s}$,$\{T_{c,s}\}$ $K_s\}K_c$)

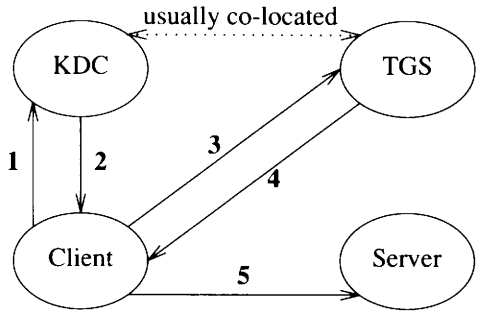
Figure 1: *Getting and Using an Initial Ticket*

An application client contacts the KDC to obtain a ticket and associated credentials. The KDC generates a new ticket by selecting a random encryption key $K_{c,s}$, called the *session key,* to include in the ticket, setting the start and expiration times in the ticket as requested, and encrypting the ticket with the server's key K_s. It assembles the ticket and session key into the response and encrypts it with the client's secret key K_c. The client decrypts the response using its key (which may be algorithmically derived from a password) and caches the ticket and associated session key for future use. It then presents the ticket and a freshly-generated authenticator to an application server formatted as a KRB_AP_REQ (application request) message. The server can decrypt this ticket using its own secret key (which is kept in secure storage on the server's host) and verify the identity of the client. If the client desires authentication of the server, the server can send a reply to the client using the key $K_{c,s}$ from the ticket, enabling the client to verify the identity of the server (only the proper server could obtain this key, as it is inside the encrypted ticket, and no intruder can gain the server's secret key). More detail on the formats of the messages used in version 4 can be found in [Ste88a] and [Mil87a]; detail on version 5 formats are in [Koh90a].

1.2. The Additional Ticket Exchange

In order to reduce the risk of exposure of the client's secret key K_c and make the use of Kerberos more transparent to the user, the above exchange is primarily used to obtain a ticket for a special *ticket-granting server* (TGS). Once this ticket-granting ticket (TGT) is obtained, the client erases the copy of the client's secret key to prevent its disclosure.

The TGS is logically distinct from the KDC which provides the initial ticket service, but runs on the KDC host and has access to the same database of clients and keys used by the KDC. A client presents its TGT (along with other request data) to the TGS as it would present it to any other application server (in a KRB_AP_REQ); the TGS verifies the ticket, authenticator and accompanying request, and replies with a new ticket for the application server. The protected part of the reply is encrypted with the session key from the TGT, so the client need not retain the original secret key K_c to decrypt and use this reply.



1. Client → KDC: c, tgs_l
2. KDC → Client: $\{K_{c,tgs}\}K_c$,$\{T_{c,tgs}\}K_{tgs}$
3. Client → TGS: $\{A_c\}K_{c,tgs}$,$\{T_{c,tgs}\}K_{tgs}$, s
4. TGS → Client: $\{K_{c,s}\}K_{c,tgs}$,$\{T_{c,s}\}K_s$
5. Client → Server: $\{A_c\}K_{c,s}$,$\{T_{c,s}\}K_s$

(In version 4, message **2** is $\{K_{c,tgs}$,$\{T_{c,tgs}\}K_{tgs}\}K_c$, and message **4** is $\{K_{c,s}$,$\{T_{c,s}\}K_s\}K_{c,tgs}$)

Figure 2: *Getting a Service Ticket*

The client then uses these new credentials to authenticate itself to the server, and perhaps to verify the identity of the server. Once the authentication is established, the client and server share a common session key $K_{c,s}$, which has never been transmitted over the network without being encrypted. They may use this key to protect or obscure their messages. Kerberos provides message formats which an application may generate as needed with the session key to assure the integrity or both the integrity and privacy of a message.

2. Why Change It? Version 4 Limitations

Although Kerberos version 4 is in widespread use, it is not sufficiently flexible to meet the needs of some sites. As a result, work on Kerberos version 5 commenced in 1989, fueled by discussions with version 4 users and administrators about their experiences with the protocol and MIT's implementation.

2.1. Environmental Shortcomings

Since Kerberos version 4 was targeted primarily for the Project Athena environment (described in [Tre88a]), it has several features which can be troublesome in other environments:

Encryption system dependence: The version 4 protocol uses only the Data Encryption Standard (DES) to encrypt messages. The export of DES from the USA is restricted by the U.S. Government, making truly widespread use of version 4 difficult.

Internet protocol dependence: Version 4 requires the use of Internet Protocol (IP) addresses, which makes it unsuitable for some environments.

Message byte ordering: Version 4 uses a "receiver makes right" philosophy for encoding multi-byte values in network messages, where the sending host encodes the value in its own natural byte order and the receiver must convert this byte order to its own

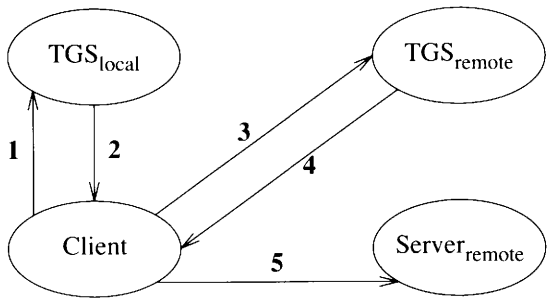
native order. While this makes communication between two hosts with the same byte order simple, it does not follow established conventions and will preclude interoperability if some machine with an unusual byte order not understood by the receiver is used.

Ticket lifetimes: The valid life of a ticket in version 4 is encoded by a UNIX timestamp issue date and an 8-bit lifetime quantity in units of five minutes, resulting in a maximum lifetime of 21¼ hours. Some environments require longer lifetimes for proper operation (e.g. a long-running simulation which requires valid Kerberos credentials during its entire execution).

Authentication forwarding: Version 4 has no provision for allowing credentials issued to a client on one host to be forwarded to some other host and used by another client. This may be useful if an intermediate service needs to access some resource with the rights of the client (e.g. a print service needs access to the file service to retrieve a client's file for printing), or if a user logs into another host on the network and wishes to pursue activities there with the privileges and authentication she had on the originating host.

Principal naming: In version 4, principals are named with three components: name, instance, and realm, each of which may be up to 39 characters long. These sizes are too short for some applications and installation environments. In addition, due to implementation-imposed conventions the normal character set allowed for the name portion excludes the period (.), which is used in account names on some systems. These same conventions dictate that the account name match the name portion of the principal identifier, which is unacceptable in situations where Kerberos is being installed in an existing network with non-unique account names.

Inter-realm authentication: Version 4 provides cooperation between authentication realms by allowing each pair of cooperating



1. Client → TGS$_{local}$: $\{A_c\}K_{c,tgs}$,$\{T_{c,tgs}\}K_{tgs}$, tgs_{rem}
2. TGS$_{local}$ → Client: $\{K_{c,tgs_{rem}}\}K_{c,tgs}$,$\{T_{c,tgs_{rem}}\}K_{tgs_{rem}}$
3. Client → TGS$_{remote}$: $\{A_c\}K_{c,tgs_{rem}}$,$\{T_{c,tgs_{rem}}\}K_{tgs_{rem}}$, s_{rem}
4. TGS$_{remote}$ → Client: $\{K_{c,s_{rem}}\}K_{c,tgs_{rem}}$,$\{T_{c,s_{rem}}\}K_{s_{rem}}$
5. Client → Server$_{remote}$: $\{A_c\}K_{c,s_{rem}}$,$\{T_{c,s_{rem}}\}K_{s_{rem}}$

(In version 4, message **2** is $\{K_{c,tgs_{rem}}$,$\{T_{c,tgs_{rem}}\}K_{tgs_{rem}}\}K_{c,tgs}$, and message **4** is $\{K_{c,s_{rem}}$,$\{T_{c,s_{rem}}\}K_{s_{rem}}\}K_{c,tgs_{rem}}$)

Figure 3: *Getting a Foreign Realm Service Ticket*

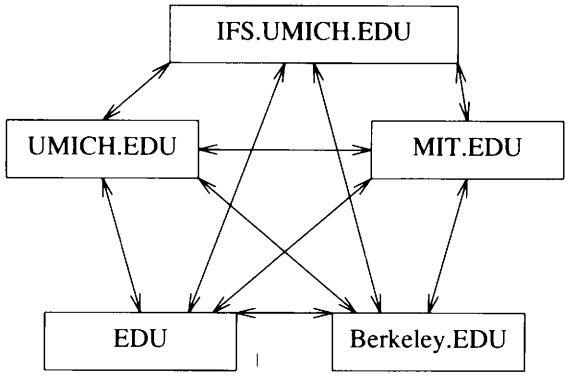


Figure 4: *Version 4 Realm Interconnections*

realms to exchange an encryption key to be used as a secondary key for the ticket-granting service. A client can obtain tickets for services from a foreign realm's KDC by first obtaining a ticket-granting ticket for the foreign realm from its local KDC and then using that TGT to obtain tickets for the foreign application server (see Figure 3). This pair-wise key exchange makes inter-realm ticket requests and verification easy to implement, but requires $O(n^2)$ key exchanges to interconnect n realms (see Figure 4). Even with only a few cooperating realms, the assignment and management of the inter-realm keys is an expansive task.

2.2. Technical Deficiencies

In addition to the environmental problems, there are some technical deficiencies in version 4 and its implementation. [Bel90a] provides detailed analyses of some of these problems.

Double Encryption: As shown in Figure 1, the ticket issued by the Kerberos server in version 4 is encrypted twice when being transmitted to the client, and only once when sent to the application server. There is no need to encrypt it in the message from the KDC to the client, and doing so can be wasteful of processing time if encryption is computationally intensive (as will be the case for most software-based encryption implementations; see [Mer90a] for discussion of fast software-based encryption methods).

PCBC encryption: Kerberos version 4 uses a modified mode of DES to encrypt its messages. [Com80a] describes the normal cipher-block-chaining (CBC) mode of DES. Kerberos version 4 uses a non-standard modified version called plain- and cipher-block-chaining mode (PCBC). This mode was an attempt to provide data encryption and integrity protection in one operation. Unfortunately, it is flawed since an intruder can modify a message with a special block-exchange attack and have this modification pass undetected to the recipient [Koh89a].

Authenticators and replay detection: Kerberos version 4 uses an encrypted timestamp method to verify the freshness of messages and prevent an intruder from staging a successful replay attack. If an authenticator (which contains the timestamp) is out of date or is being replayed, the application server rejects the authentication. However, maintaining a list of unexpired authenticators which have already been presented to a service can be hard to

implement properly (and indeed is not implemented in the version 4 implementation MIT distributes).

Password attacks: The initial exchange with the Kerberos server encrypts the response with a client's secret key, which in the case of a user is algorithmically derived from a password. An intruder is able to record an exchange of this sort and, without alerting any system administrators, attempt to discover the user's password by decrypting the response with each password guess. Since the response from the Kerberos server includes plaintext she can verify, the intruder can try as many passwords as she has available, and she will know when she's found the proper password, since the decrypted response will make sense [Lom89a].

Session keys: Each ticket issued by the KDC contains a key specific to that ticket, called a session key, which may be used by the client and server to protect their communications once authentication has been established. However, since many clients use a ticket multiple times during a user's session, it may be possible for an intruder to replay messages from a previous connection to clients or servers which do not properly protect themselves (again, MIT's version 4 implementation does not properly implement this protection for the KRB_SAFE and KRB_PRIV messages). Additionally, there are situations in which a client wishes to share a session key with several servers. This requires special non-standard application negotiations in version 4.

Cryptographic checksum: The cryptographic checksum (sometimes called a message authentication code or hash or digest function) used in version 4 is based on the quadratic algorithm described in [Jue85a]. The MIT implementation does not perform this function as described; the suitability of the modified version as a cryptographic checksum function is unknown.

3. Remedies and Changes Introduced with Version 5

Version 5 of the protocol has slowly evolved over the past two years based on implementation experience and discussions within the community of Kerberos version 4 users. Its final form is now nearing closure, and a draft description of the protocol is available [Koh90a]. It addresses the concerns above and provides additional functionality.

3.1. Changes between Versions 4 and 5

Use of Encryption

To modularise the system and ease export-regulation considerations for version 5, the use of encryption is separated into distinct software modules which can be replaced or removed by the programmer as needed. When encryption is used in a protocol message, the ciphertext is tagged with a type identifier so that the recipient can identify the appropriate decryption algorithm necessary to interpret the message.

Each encryption algorithm is responsible for providing sufficient integrity protection for the plaintext so that the receiver can verify that the ciphertext was not altered in transit. If the algorithm does not have such properties, it can be augmented by including a checksum in the plaintext before encryption. By doing this, we can discard the flawed

PCBC DES mode, and use the standard CBC mode with an embedded checksum over the plaintext.

Encryption keys are tagged with a type and length when they appear in messages. Since it is conceivable to use the same key type in multiple encryption systems (e.g. different variations on DES encryption), the key type may not map one-to-one to the encryption type.

Network Addresses

When network addresses appear in protocol messages, they are similarly tagged with a type and length field so the recipient can interpret them properly. If a host supports multiple network protocols or has multiple addresses of a single type, all types and all addresses can be provided in a ticket.

Message Encoding

Network messages in version 5 are described using the Abstract Syntax Notation One (ASN.1) syntax [Sta87a] and encoded according to the basic encoding rules [Sta87b]. This avoids the problem of independently specifying the encoding for multi-byte quantities as was done in version 4. It makes the protocol description look quite different from version 4, but it is primarily the presentation of the message fields that changes; the essence of the Kerberos version 4 protocol remains.

Ticket Changes

The Kerberos version 5 ticket has an expanded format to accommodate the required changes from the version 4 ticket. It is split into two parts, one encrypted and the other in plaintext. The server's name in the ticket is in plaintext, since it need not be encrypted to provide secure authentication. The server's name is retained since it may be needed to select a key with which to decrypt the ticket if a server has multiple identities (such as an inter-realm TGS). Everything else remains encrypted. The ticket lifetime is encoded as a starting time and an expiration time (rather than a specific lifetime field), affording nearly limitless ticket lifetimes. The new ticket also contains a new flags field and other new fields used to enable the new features described below (such as authentication forwarding).

Naming Principals

Principal identifiers are multi-component names in Kerberos version 5. The identifier is encoded in two parts, the realm and the remainder of the name. The realm is separate to facilitate easy implementation of realm-traversal routines and realm-sensitive access checks. The remainder of the name is a sequence of however many components are needed to name the principal. The realm and each component of the remainder are each encoded as an ASN.1 `GeneralString`, so there are few practical restrictions on the characters available for principal names.

Inter-Realm Support

In version 5, different Kerberos realms cooperate by establishing a hierarchy of realms (based on the name of the realm). Any realm can interoperate with any other realm in the hierarchy as long as they can interoperate with the realms between them in the hierarchy. Each realm exchanges a different inter-realm key with its parent node and

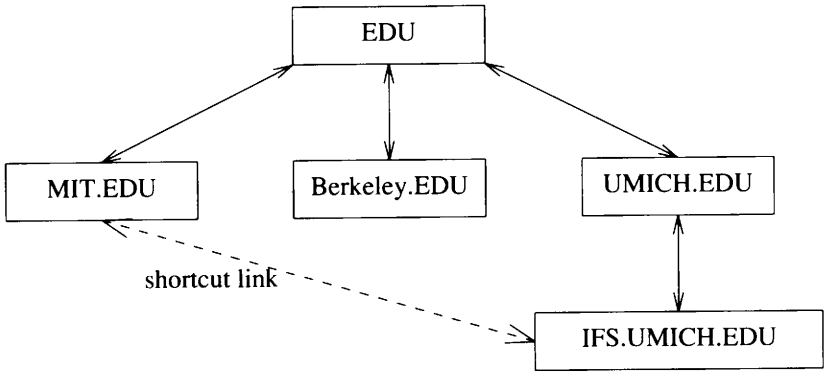


Figure 5: *A Version 5 Hierarchy of Realms*

each child node, and uses that key and a common encryption system to obtain tickets for each successive realm along the path. This arrangement reduces the number of key exchanges to $O(\log(n))$.

When an application needs to contact a server in a foreign realm, it "walks" up and down the tree toward the destination realm, contacting each realm's KDC in turn, asking for a ticket-granting ticket to the foreign realm. In most cases, the KDC will issue a ticket for the next node in the proper direction on the tree. If a realm has established a "shortcut" spanning link with some realm further in the path, it issues a ticket-granting ticket for that realm instead. This way every realm can interoperate, and heavily-traveled paths can be optimised with a direct link.

When a ticket for the end service is finally issued, it will contain an enumeration of all the realms consulted in the process of requesting the ticket. An application server which applies strict authorization rules is permitted to reject authentication which passes through certain untrusted realms.

3.2. New Protocol Features in Version 5

Ticketing Options

In addition to the ticket changes discussed above, there are a set of timestamps and flags which allow more flexible use of tickets than was available in version 4.

Each ticket issued by the KDC is flagged as having been issued based on an initial ticket exchange or an additional ticket exchange. Some application servers (such as password changing programs) may require that a client present a ticket obtained by direct use of the client's secret key K_c, so that intruders who might try to abuse that service cannot simply steal cached credentials from an unattended user session.

Tickets may be issued as renewable tickets with two expiration times, one for a time in the near future, and one for a farther point. The ticket expires like normal at the earlier time, but if it is presented to the KDC in a renewal request before this earlier expiration time, a replacement ticket is returned which is valid for an additional period of time. The KDC will not renew a ticket beyond the second expiration indicated in the ticket. This mechanism has the advantage that although the credentials can be used for long periods of time, the KDC may refuse to renew

tickets which are reported as stolen and thereby thwart their continued use.

A similar mechanism is available to assist in authentication during batch processing. A ticket issued as postdated and invalid will not be valid until its post-dated starting time passes and it is replaced with a validated ticket. The client validates the ticket by presenting it to the KDC as described above for renewable tickets.

Authentication forwarding can be implemented by contacting the KDC with the additional ticket exchange and requesting a ticket valid for a different set of addresses than the TGT used in the request. The KDC will not issue such tickets unless the presented TGT has a flag set indicating that this is a permissible use of the ticket. When the entity on the remote host is granted only limited rights to use the authentication, the forwarded credentials are referred to as a *proxy* (after the proxy used in legal and financial affairs). Proxies are handled similarly to forwarded tickets, except that new proxy tickets will not be issued for a ticket-granting service; they will only be issued for application server tickets.

In certain situations, an application server (such as an X Window System server) will not have reliable, protected access to an encryption key necessary for normal participation as a server in the authentication exchanges. In such cases, if the server has access to a user's ticket-granting ticket and associated session key (which in the case of single-user workstations may well be the case), it can send the server's ticket-granting ticket to the client, who then presents it and the user's own ticket-granting ticket to the KDC. The KDC then issues a ticket encrypted in the session key from the server's ticket-granting ticket; the application server has the proper key to decrypt and process this ticket. [Dav90a] provides details on the fine points of this exchange.

Authorization Data

Some network operating system applications need to provide tamper-proof arbitrary data to an application server (for example, such information might include group membership information). It is convenient to collect or generate such information at a KDC and insert it into a ticket as *authorization data*, where it is encrypted and protected from any client or intruder tampering. In the protocol's most general form, a client may request that the KDC include or add to such data in a new ticket. The KDC does not remove any authorization data from a ticket; the TGS always copies it from the TGT into the new ticket, and then adds any requested additional authorization data. Upon decryption of a ticket, the authorization data is available to the application server. While Kerberos makes no interpretation of these data, the application server is expected to use the authorization data to appropriately restrict the client's access to its resources.

This field can be used in a proxy ticket to create a capability. The client requesting the proxy ticket from the KDC specifies any authorization restrictions in the authorization data, then securely transmits the proxy ticket and session key to another entity, which can then use the ticket to obtain limited service from an application server. [Neu91a] discusses in more detail some uses of this field.

Pre-Authentication Data

In an effort to help alleviate the ever-present problem of stolen passwords, the Kerberos version 5 protocol has fields available in the

initial- and additional-ticket exchange messages to enable alternative identification methods, such as hand-held authenticators (devices which have internal circuitry to help a user identify herself to the system). In the initial ticket exchange, these fields might be used to alter the client's key K_c in which the reply is encrypted; they may also be used to implement a challenge/response protocol which must be completed before the issuance of a ticket. Both alternatives can help alleviate the password attack problems discussed above, if they make the derivation of the key from a typed password hard or impossible to compute without the additional information utilised in the exchange, or if they eliminate the use of passwords to derive or protect encryption keys.

This pre-authentication data field is used by the client in the additional ticket exchange to pass the ticket-granting ticket to the KDC; since it is a variable-length array, other values may also be sent in the additional-ticket exchange.

Subsession Key Negotiation

Tickets may be cached by clients for later use before their expiration dates. In order to avoid problems caused by re-using a ticket's session key (which is held for the duration of the user's login session) for several application sessions, a server and client can cooperate to choose a new *subsession key* to protect just that application session. This subsession key is discarded after the application session concludes.

A clever use of this negotiation allows an application to use a broadcast medium while protecting its messages to several recipients. The application can negotiate individually with each recipient to use the same subsession key before beginning its broadcasts.

Sequence Numbers

Kerberos provides two messages for applications to protect their communications. The KRB_SAFE message uses a cryptographic checksum to insure data integrity. The KRB_PRIV message uses encryption to insure integrity and privacy. In version 4 these messages included as control information a timestamp and the sender's network address. With version 5, an application may elect to use a timestamp (as before) or a sequence number. If the timestamp is used, the receiver must record the known timestamps to avoid replay attacks; if a sequence number is used the receiver must verify that the messages arrive in the proper order without gaps. There are situations where one choice makes applications simpler (or even possible) to implement; see the discussions in [Koh90a].

4. Implementation Features

4.1. The Base Implementation

The MIT implementation of the version 5 protocols is composed of several run-time libraries with which a program may link. The core library functions will probably be used by all applications; other libraries or subsystems may be replaced or omitted as needed by an application programmer. All code is currently written in "C."

The base functions: The core Kerberos library contains the routines which assemble, disassemble and interpret the network messages. This includes ASN.1 encoding and decoding functions

Field name	*Field use*
`encrypt_func()`	Entry point to encrypt an input.
`decrypt_func()`	Entry point to decrypt an input.
`process_key()`	Entry point to perform any necessary key processing (must be called before either `encrypt_func` or `decrypt_func`).
`finish_key()`	Entry point to clean up from any key processing (called after `encrypt_func` or `decrypt_func`).
`string_to_key()`	Entry point to convert a string to a key.
`init_random_key()`	Entry point to initialise state for generating random keys.
`finish_random_key()`	Entry point to clean up state from random key generation.
`random_key()`	Entry point to generate a random key.
`block_length`	The minimum size of input and output for this encryption system.
`pad_minimum`	The minimum padding space required of any input (used to insert integrity checks).
`keysize`	The length (in octets) of keys used by this system.
`proto_enctype`	The encryption type value used in the protocol.
`proto_keytype`	The key type value used in the protocol.

Table 1: *A Cryptosystem Table Entry*

which convert from a machine's native format to the network encoding (currently based on the ISODE library, but used in a way to allow easy replacement of the ASN.1 routines), routines which verify that requests are answered as expected, and routines to determine which messages are necessary. This core set of routines calls out to the remaining portions of the library as required. A programmer may replace those portions at certain specified interfaces.

Encryption routines: Since multiple encryption types may be in use simultaneously, the core functions call encryption routines through a function table which has entries provided by each encryption system implementation. Table 1 shows the fields in a cryptosystem table entry. The core library provides a default cryptosystem table, initialised to list the known encryption types. A programmer may load his own cryptosystem table to replace the default table and avoid linking with the default encryption libraries.

In an attempt to alleviate some possible export restrictions, MIT's implementation distributes its encryption systems separately from the remainder of the system. Only DES is currently available from MIT.

Checksum routines: In a similar fashion to the encryption routines, the core routines call any needed checksum functions through a function table, and compute any necessary sizes based on the information in the table. Certain applications of checksum tech-

Field name	*Field use*
`sum_func()`	Entry point for the checksum function.
`checksum_length`	The length (in octets) of the checksum produced by the `sum_func`.
`is_collision_proof`	Binary value indicating whether this checksum is collision proof.
`uses_key`	Binary value indicating whether this checksum is keyed.

Table 2: *A Checksum Table Entry*

Field name	*Field use*
`prefix`	The string prefix used to name this variety of credentials cache.
`get_name()`	Entry point to return the name of a key table.
`resolve()`	Entry point to reserve a cache name, prepare to access it, and return an access handle.
`gen_new()`	Entry point to generate a unique credentials cache, prepare to access it, and return an access handle.
`init()`	Entry point to create or erase a cache.
`destroy()`	Entry point to destroy a cache and invalidate the access handle.
`close()`	Entry point to close a cache and invalidate the access handle.
`store()`	Entry point to store an entry in the cache.
`retrieve()`	Entry point to retrieve an entry from the cache.
`get_princ()`	Entry point to retrieve the primary principal named in the cache.
`get_first()`	Entry point to prepare to sequentially read all entries in the cache.
`get_next()`	Entry point to read the next entry in the cache.
`end_get()`	Entry point to stop reading every entry in the cache.
`remove_cred()`	Entry point to remove an entry or entries from the cache.
`set_flags()`	Entry point to set various flags for the cache routines.

Table 3: *A Credentials Cache Table Entry*

nology require that the checksum have certain properties. The table entry indicates whether the checksum is keyed (its algorithm is perturbed by an encryption key which cannot be discovered with knowledge only of the algorithm and the checksummed text) and whether the checksum is collision proof (it is computationally infeasible to discover a different checksum text which has the same checksum). Table 2 shows the fields in a checksum table entry. The core library provides a replaceable default checksum table.

Three checksums are currently available from MIT: the CRC-32, which is neither keyed nor collision proof (but it is useful for integrity checks within encryption systems); the DES message authentication code (MAC), which is both keyed and collision proof, and MD4 [Riv90a], which is collision proof but not keyed.

Credentials cache and key table routines: When clients store tickets and credentials in a cache, the core routines call out through a credentials cache table entry to a separate library module which implements the storing and searching routines for credentials caches (see Table 3). An environment variable can be used to specify the default type and location of a credentials cache, so a user can switch between different types and locations of caches as needed (perhaps if she is working in two roles and wants to keep the credentials for each role separate). MIT's implementation provides two credentials cache implementations, one built on C "standard I/O" routines and the other built on UNIX file-descriptor semantics. Other implementations could provide shared-memory or kernel-resident caches.

Servers likewise store their secret keys K_s in key tables accessed by the core routines through a key table function table entry (see Table 4). MIT's implementation provides a key table library built on C "standard I/O" routines.

KDC database support: All accesses to the KDC's principal database by the KDC and administrative programs are mediated by a database library which can be replaced if needed. MIT's implementa-

Field name	*Field use*
`resolve()`	Entry point to resolve a key table name, prepare to access it, and return an access handle.
`get_name()`	Entry point to return the name of a key table.
`close()`	Entry point to close a key table and invalidate its handle.
`get()`	Entry point to search the key table and return a requested entry.
`start_seq_get()`	Entry point to prepare to read every key in the table.
`get_next()`	Entry point to read the next key in the table.
`end_get()`	Entry point to stop reading every key in the table.
`add()`	Entry point to add an entry to the table.
`remove()`	Entry point to delete an entry from the table.

Table 4: *A Key Table Function Table Entry*

tion uses the UNIX *dbm* database system. Since *dbm* does not do any record or database locking, it is augmented with separate locking code to mediate between writers and readers. Administrative requests (e.g. adding entries, changing keys or passwords) can be handled on-line.

Operating system support: Although it is targeted for UNIX systems, the MIT implementation is careful to access operating system features only in a few well-contained modules. An operating system support library performs all the accesses required by the rest of the code, such as transmitting and receiving network messages, examining configuration files, checking the system's time-of-day, translating from account names to Kerberos names (and *vice versa*), and performing rudimentary account access checks.

4.2. User Interaction

If all parts of Kerberos are working properly, a user will normally not be aware that Kerberos authentication is in use by her applications. The normal login process obtains and caches an initial ticket-granting ticket, and applications automatically obtain and cache service tickets as required. Only when authentication fails will the user become aware of the underlying use of Kerberos.

If the user needs to refresh tickets (say, if they expire), then she can use the *kinit* program, which will get a new ticket-granting ticket after reading her password from the keyboard. She may examine the cached tickets with *klist* and destroy the cache with *kdestroy*.

When principal names need to be displayed to human users, by convention† they are represented as the sequence of name components separated by slashes (/), followed by an at-sign (@), and the realm name. Thus, a principal with two name components `jtkohl` and `role2` in the realm `ATHENA.MIT.EDU` would be represented as `jtkohl/role2@ATHENA.MIT.EDU`.

Password to Key Conversion

Since users are not good at remembering binary encryption keys, we provide routines to convert passwords into keys. The algorithm used to convert a password into an encryption key performs a non-invertible

† Please note that this is only a *convention*, and other implementations may display the principal names differently.

transformation, so that an attacker cannot discover a user's password if he knows the K_c. The conversion can be seeded with an additional string which perturbs the output key, so that a user who is registered in multiple realms and uses the same password in two of those realms will have a different K_c in each realm. Without this perturbation, someone discovering the user's key in one realm could impersonate that user in the other realm (without knowing her password!). When no additional perturbation string is supplied, the resulting key is the same as the key produced by the version 4 algorithm.

4.3. Compatibility Support for Version 4

There is a small but growing base of Kerberos version 4 applications, and a number of sites running a Kerberos version 4 authentication server. Several features of MIT's implementation of version 5 can help sites and programmers convert to using the newer protocol.

Interface compatibility: MIT's implementation of version 5 includes a "glue library" which may be used to convert applications which are coded to use the version 4 application programming interface (API) to use version 5 protocol messages and routines. This library converts data structures as much as possible between the differing version 4 and version 5 data structures. In many cases (especially those that use only a common subset of the version 4 library functions), an application need only be re-linked with this library and the remainder of the version 5 code to use version 5 protocols. However, such applications will no longer be compatible with older peer processes, which would still expect the version 4 messages, and continued maintenance may be made more difficult.

A generic authentication interface: An authentication-system independent programming interface [Lin90a] has been discussed by representatives from several computer system manufacturers. MIT provides a binding of this interface to the Kerberos version 5 implementation. For new applications which desire the most flexibility to have different authentication systems (even ones not yet invented) supplied by the system, this offers an attractive abstraction boundary. If an application needs more detailed access to a particular authentication system, it would probably do better to code to that system's native interface.

Protocol compatibility: For those sites which wish to convert the Kerberos server to provide the features of version 5, a compatibility mode may be enabled on the server to access the version 5 style authentication database but provide version 4 format tickets and messages. This allows an administrator to convert a version 4 installation to version 5 slowly, by supporting the old users with the compatibility code. After some grace period, the version 4 compatibility would be turned off. If a user wishes to use both version 4 and version 5 programs simultaneously, the compatibility code can utilise the pre-authentication data in the ticket responses to indicate which algorithm should be used to convert her password to an encryption key.

Interface coexistence: The MIT version 5 libraries were purposely designed to allow an application to simultaneously support both versions 4 and 5, and this is the suggested compatibility mode. The telnet [Pos83a] program distributed with the MIT code can automatically choose an authentication system to use when it

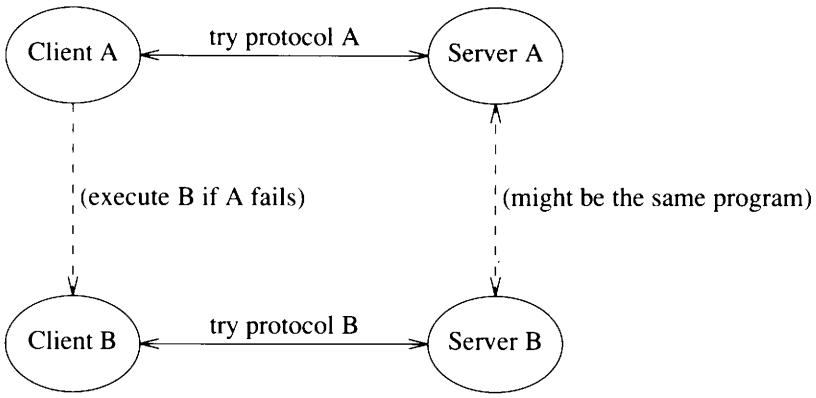


Figure 6: *Implementing Protocol Compatibility by Executing Separate Programs*

connects to a remote system, based on what credentials the user holds and what versions of authentication the remote telnet server will accept. It implements the current draft specifications of the authentication [Bor90a] and encryption [Bor90b] options.

Program compatibility: Another possible compatibility mode can be fabricated by maintaining separate copies of network applications which use version 4 and version 5 protocol messages. The user would use a generic name for the application, and the application would try each authentication system in turn, by executing a separate copy of the program for each system (see Figure 6). When authentication is successfully completed, the application would proceed as normal. On both the client and server sides of the application, this approach requires two copies of the same program, each linked with a different authentication system. The different versions of the server would each accept requests at different network ports, and the different clients would only send a request to the server which supports its authentication type.

This approach could be mixed with the glue library and/or single-server approaches, by creating the separate clients using the glue library and/or using a single server program which understands both protocols.

5. Status and Future Work

Kerberos version 5 is a large step forward toward generalising Kerberos to make it globally useful. We believe its framework will be flexible enough to accommodate future requirements. Some items we expect to be incorporated into Kerberos in the near future include:

Public-key cryptosystems: The encryption specifications in Kerberos version 5 are designed primarily for secret-key cryptosystems, but there is some ongoing work into the integration of public-key cryptosystems into Kerberos, and we hope to be able to better support them in future code releases. However, public-key cryptosystems have different characteristics than secret-key systems, and their use in Kerberos may not take advantage of those characteristics.

"Smartcards": Several companies manufacture hand-held devices which can be used to augment normal password security

methods, and there is strong interest within the industry to integrate one or more of these systems with Kerberos.

Remote administration: The current protocol specifications do not specify any administrative interface to the KDC database. MIT's implementation provides a sample remote administration program which allows administrators to add and modify entries and users to change their keys. We would eventually like to standardise such a protocol. Some features we would like to add include remote extraction of server key tables, password "quality checks," and a provision for servers to change their secret keys automatically every so often.

Directional inter-realm keys: The protocols will support the use of a different inter-realm key for each direction of an inter-realm link, but our implementation only allows for the same key to be used for both directions. We would like to allow separate keys in our implementation, to reduce the exposure from a disclosed key.

Database propagations: The current implementation provides reliable KDC service by a periodic bulk-copy of the KDC database to slave KDC machines. It might be more convenient and/or efficient to build the KDC on a distributed database technology. However, the technology must provide a secure, private transmission of the database elements to each server, to insure that an attacker cannot illegitimately obtain any database entry.

Validation suites: The current implementation does not include a complete validation suite to verify that the protocol is properly implemented. Such a suite could prevent future security problems in the case of a faulty implementation, and would help facilitate interoperation of diverse implementations.

Applications: There are many more network applications that would benefit from the addition of authentication which we have not had time or resources to convert. Among the highly visible examples are electronic mail and popular bulletin-board systems (such as Usenet).

Acknowledgements

The work described here has been the result of many MIT Project Athena and MIT Network Services staff members' visions, ideas, and hard work.

The author would especially like to thank Steve Bellovin, Clifford Neuman, Jennifer Steiner, and Ralph Swick for their comments on early drafts of this paper.

References

[Bel90a] S. M. Bellovin and M. Merritt, "Limitations of the Kerberos Authentication System," *Computer Communications Review* **20**(5), pp. 119-132 (October 1990).

[Bor90a] D. Borman, Editor, "Telnet Authentication Option," Internet-Draft, Internet Engineering Task Force, Telnet Working Group (July 1990).

[Bor90b] D. Borman, Editor, "Telnet Encryption Option," Internet-Draft, Internet Engineering Task Force, Telnet Working Group (April 1990).

[Com77a] National Bureau of Standards, U.S. Department of Commerce, "Data Encryption Standard," Federal Information Processing Standards Publication 46, Washington, DC (1977).

[Com80a] National Bureau of Standards, U.S. Department of Commerce, "DES Modes of Operation," Federal Information Processing Standards Publication 81, Springfield, VA (December 1980).

[Dav90a] Don Davis and Ralph Swick, "Workstation Services and Kerberos Authentication at Project Athena," Technical Memorandum TM-424, MIT Laboratory for Computer Science (February 1990).

[Jue85a] R. R. Jueneman, S. M. Matyas, and C. H. Meyer, "Message Authentication," *IEEE Communications* **23**(9), pp. 29-40 (September 1985).

[Koh89a] John T. Kohl, "The Use of Encryption in Kerberos for Network Authentication," in *Crypto '89 Conference Proceedings*, International Association for Cryptologic Research, Santa Barbara, CA (August 1989).

[Koh90a] John T. Kohl and B. Clifford Neuman, "The Kerberos Network Authentication Service," RFC DRAFT 4, Project Athena, Massachusetts Institute of Technology (December 1990).

[Lin90a] John Linn, *Generic Security Service Application Program Interface,* Digital Equipment Corporation (September 1990). Version C.3.

[Lom89a] T. Mark A. Lomas, Li Gong, Jerome H. Saltzer, and Roger M. Needham, "Reducing Risks from Poorly Chosen Keys," *Operating Systems Review* **23**(5), pp. 14-18 (December 1989).

[Mer90a] Ralph C. Merkle, "Fast Software Encryption Functions," in *Crypto '90 Conference Proceedings*, International Association for Cryptologic Research, Santa Barbara, CA (August 1990).

[Mil87a] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, *Section E.2.1: Kerberos Authentication and Authorization System,* M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).

[Nee78a] Roger M. Needham and Michael D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM* **21**(12), pp. 993-999 (December, 1978).

[Neu91a] B. Clifford Neuman, "Proxy-Based Authorization and Accounting for Distributed Systems," Technical Report 91-02-01, Department of Computer Science and Engineering, University of Washington (February 1991).

[Pos83a] J. Postel and J. Reynolds, "TELNET Protocol Specification," RFC 854, University of Southern California, Information Sciences Institute (May 1983).

[Riv90a] R. Rivest, "The MD4 Message Digest Algorithm," RFC 1186, MIT Laboratory for Computer Science (October 1990).

[Riv78a] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM* **21**(2), pp. 120-126 (February 1978). See also U.S. Patent 4,405,829.

[Sta87a] International Organization for Standardization, "Information Processing Systems – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)," IS 8824 (December 1987). First Edition.

[Sta87b] International Organization for Standardization, "Information Processing Systems – Open Systems Interconnection – Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)," IS 8825 (November 1987). First Edition.

[Ste88a] J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," pp. 191-202 in *Usenix Conference Proceedings*, Dallas, Texas (February, 1988).

[Tre88a] G. W. Treese, "Berkeley Unix on 1000 Workstations: Athena Changes to 4.3BSD," pp. 175-182 in *Usenix Conference Proceedings*, Dallas, Texas (February, 1988).

Architechture and Implementation of a User-Space NFS

Benoy DeSouza
Nawaf Bitar

Apollo Systems Division, Hewlett-Packard Company

Abstract

Traditionally, filesystems under UNIX have been built into the operating system kernel. The Sun *Network File System (NFS)* has not been an exception. The NFS client is implemented under the *vnode* layer, a filesystem implementation-independent layer that allows access to multiple filesystems through a common interface. The NFS server is merely a kernel daemon that services request from NFS clients.

In addition to consuming physical memory resources, a kernel implementation has the disadvantage of reducing extensibility and portability. These problems can be overcome by implementing NFS in user-space. This, however, may result in a severe performance degradation unless the underlying operating system provides specialized support for an extensible I/O mechanism.

This paper explores the architechtural and implementation issues involved in constructing a user-space NFS over such a mechanism, following which current performance data and possible enhancements for further performance improvements are presented.

1. Introduction

I/O services are increasing in complexity as they strive to provide increased network access, transparency of access and high performance [Gol90a]. For example, NFS provides in-place access to remote networked filesystems allowing users to access these remote filesystems as if they were local. Since UNIX I/O has typically been implemented in the kernel, this increased complexity has resulted in larger kernels which in turn result in an increased consumption of physical memory resources at the cost of memory that would have otherwise been available to applications.

In addition, introducing new I/O services such as NFS into the system has required modification to kernel source modules and a subsequent kernel rebuild. Coupled with the increased consumption of memory, this restriction has resulted in reduced extensibility: the memory limitation places a limit on the number of new I/O services that may be introduced and the requirement for a kernel rebuild makes the addition of these services more difficult. Finally, portability is reduced as kernel resident modules will most likely be forced to use operating system

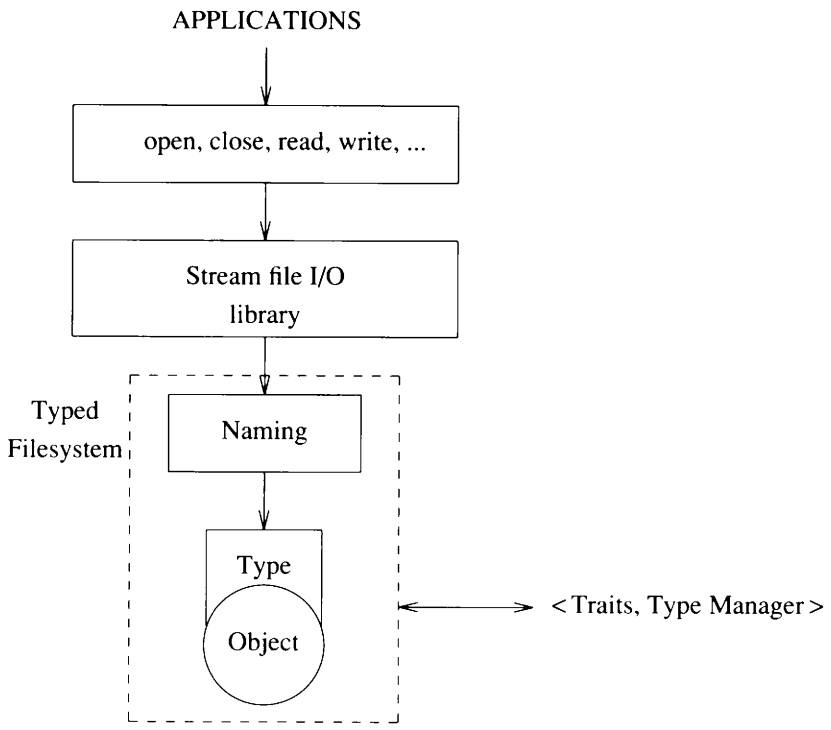


Figure 1: *The Domain Typed Object Management System*

dependent services as standard application programming interfaces are not typically available to them.

These problems can be ameliorated with the introduction of an extra-kernel I/O mechanism. Such a mechanism allows I/O services to be built as pageable user programs and thus those services do not require locked physical memory. Furthermore, introducing these services into the system does not require kernel modifications and moreover, they can be developed and debugged using conventional programming tools. As a result, the extensibility of the system is significantly increased. Since these services can be implemented using standard application programming interfaces, portability is likewise increased.

The Domain operating system has such an extensible file I/O framework for stream operations. It consists of a typed object management system [Ree86a] on which a stream file I/O facility is layered as a user-state library that is mapped into every process' address space. It allows users to define new types of I/O objects and to associate user-defined sets of generic file operations, *traits*, with these types. For example, an *io* trait could be *read*, *write*, and *seek*. *Type managers* are programs that implement these traits. After a type manager for a given set of typed objects is created it will be dynamically bound to those typed objects on demand. The relationship between type managers and their objects is illustrated in Figure 1.

The Domain NFS implementation is built on this framework. An object of type **NFS** is defined and NFS mount points (local filesystem entries that point to remote NFS mounted filesystems) are objects of this type. The NFS client is a type manager that implements a set of *open/close*, *io*, and *directory* traits, that initiate NFS server operations via Sun RPC/XDR. Basically, a client I/O operation against an object of type **NFS** results in the NFS type manager being dynamically loaded into that client's address space to handle the operation. The NFS server is

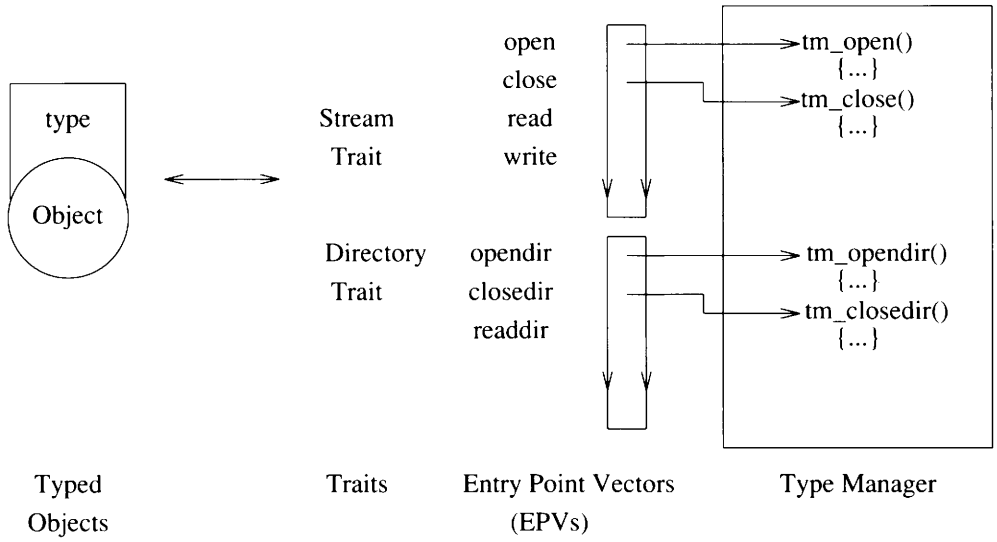


Figure 2: *Types, Traits, Managers (TTM) and EPVs*

merely a collection of user-space daemons that authenticate and service NFS requests from clients.

2. An Extra-Kernel I/O System

2.1. The Client-Side

The principal responsibility of the I/O client side is to identify and categorize each object and to invoke operations that define each object as necessary. This can be accomplished by strongly typing each object with a set of states and maintaining well-defined traits that implement the state transitions. A trait can thus be considered a specification of the state transitions of an object and represents the class of behaviour that the object can support (e.g. files, directories, block devices, etc.). As previously mentioned, the program that implements the traits for an object type is known as that object's type manager and an object supports a trait only if its type manager implements those trait operations. Support for a trait is accomplished by associating an *entry point vector (EPV)* with an object type. An EPV is an ordered list of pointers to functions that implement the trait operations as illustrated in Figure 2.

The usefulness of this framework hinges on the ability to uniquely identify an object and its type. This is best accomplished by means of a typed filesystem. A type $\leftrightarrow$ trait database that maintains a list of traits that each type supports must also exist and finally, some mechanism to bind the < object, trait > tuple to an EPV is required. This will allow for a quick transalation from < object, trait > $\rightarrow$ type manager.

2.2. The Server-Side

Exporting server-side I/O to user-space is much simpler. All that is required is a daemon to service client requests. For better throughput it is advisable to use either a multi-threaded daemon or multiple cooperating daemons.

2.3. Vnode Layer +

The vnode layer [Kle86a] in traditional UNIX kernels is a separation of generic filesystem operations from their implementation characteristics. Applications invoking vnode supported calls will cause the vnode layer to invoke the appropriate kernel-resident filesystem-dependent procedure. The previously described extensible I/O system exhibits this same funcitonality in user-space. In additon to the greater extensibility and portability that results from a user-space system, the extensible I/O system also enjoys greater flexibility in the granularity at which new types can be introduced into the system. While the vnode layer can only define new filesystem types (coarse granularity), the extensible I/O system can add a large number of file types within any filesystem (fine granularity).

3. System Support for Extensible I/O

The implementation of the described I/O systen in user-space requires support from several system facilities. In particular:

- Object/type identification
- Object mapping into process address spaces
- User-space storage pools
- Object *concurrency* data
- Dynamic program loading
- Type manager pathname resolution
- User-space synchronization

3.1. Object/Type Identification

Every I/O object must have a type associated with it. A typed filesystem enables files to be divided into classes of objects with each class corresponding to an object type. Since the type information is part of the filesystem, type managers can use the filesystem to store objects.

3.2. Object Mapping into Process Address Spaces

Type managers must have controlled low-level access to the raw data that their objects represent. This enables them to efficiently implement their own caching and buffering schemes. Mapping objects into a process' address space affords a type manager such access. Furthermore, it causes an address space to be characterized in terms of objects that are mapped. When an object mapping occurs, a binding between that object and its type manager can be created.

3.3. User-space Storage Pools

Since the extensible I/O system is implemented in user-space, an object's state information should also be stored in user-space. Multiple pools are required: one for per-process information, another for ancestor-common processes, and a third for global storage. The per-process pools are used to store process stacks, while the ancestor-common pools store stream information, such as the position in a byte stream, that must be available to both the parent and child processes after a fork. The global pool is used for storage that must be visible in

all process address spaces, such as the file handle and attributes of an **NFS** object.

3.4. Object Concurrency Data

Kernel-resident NFS implementations have used the vnode structure to store file/data concurrency information such as an indication as to whether the data is being accessed for read or write. A similar user-space facility is required for a user-space NFS implementation in order to coordinate access by multiple processes to the same object. In addition to being associated with a particular object, this data must be stored in a global pool. Thus, there must exist a mechanism for any process to aquire the virtual address of the gloabl data. The first process to access the object allocates the storage. Subsequent processes increment a reference count and are allowed access to the data. The storage space is reclaimed when the reference count falls to zero indicating that all processes have completed their access.

3.5. Dynamic Program Loading

A dynamic program loader allows for a type manager to be loaded when an object of a type that it implements is accessed. This has the advantage of not consuming virtual address space for type managers of unaccessed object types.

3.6. Type Manager Pathname Resolution

While not strictly required for a user-space NFS implementation, type manager pathname resolution is useful as part of an extensible I/O system. It allows type managers to resolve names that may not exist in the filesystem namespace and thus could not be resolved by traditional kernel pathname resolution algorithms. Thus applications have the ability to use object specific names without kernel knowledge of them.

3.7. User-space Synchronization

Since multiple processes may be waiting for I/O events involving the same object, a user-space synchronization facility must exist.

4. The NFS Protocol

The NFS protocol provides transparent, in-place access to remote networked filesystems by means of remote procedure calls (RPC) [San85a]. A *mount* operation results in the insertion of a foreign filesystem subtree into the native filesystem. Any client operations below the new mount point will transparently be converted to operations on the remote filesystem. Every NFS object (files and directories) is identified by a thirty-two byte file handle that the server passes to a client. An NFS server exports the synchronous procedures listed in Table 1.

The NFS protocol is transaction oriented. Each NFS request from client to server is independent of any other request (see Figure 3). That is, each request provides all the necessary information for the server to perform the operation. Thus, the NFS server is stateless and does not maintain any information about its clients. This results in a simple, lightweight server and a trivial crash recovery algorithm. The disad-

NFS Protocol Server Procedures
Null
GetFileAttributes
SetFileAttributes
LookupFileName
ReadSymbolicLink
ReadFile
WriteFile
CreateFile
DeleteFile
RenameFile
CreateLinkToFile
CreateSymbolicLink
CreateDirectory
RemoveDirectory
ReadDirectory
GetFilesystemAttributes

Table 1: *NFS Protocol Server Procedures*

vantage of a stateless server is that all state has to be written to stable storage before a request is acknowledged, thus forcing a synchronous style of interaction. Consequently, a significant performance degradation can result for disk intensive operations such as writes.

5. Implementation of a User-space NFS Client

A user-space NFS client can be implemented with an **NFS** typed mount point in the native filesystem and an NFS type manager to manage the **NFS** objects as illustrated in Figure 4. The mount point stores sufficient information to contact the remote NFS server. This includes the server's internet address, file handle of the remote filesystem mount point and any mount options such as timeout values and network packet sizes.

During pathname resolution, when an NFS mount point is encountered the remainder of the pathname is passed on to the NFS type manager and it asssumes responsibility for resolving the pathname. This

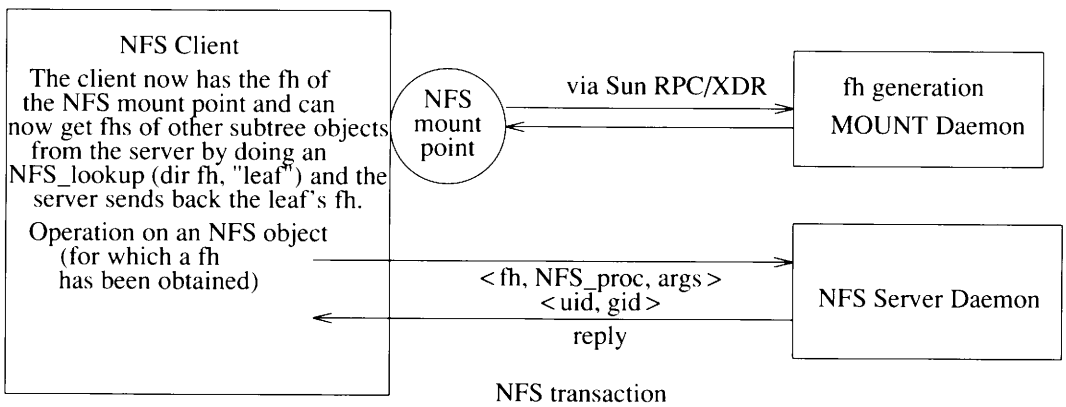


Figure 3: *NFS Server/Client Interaction*

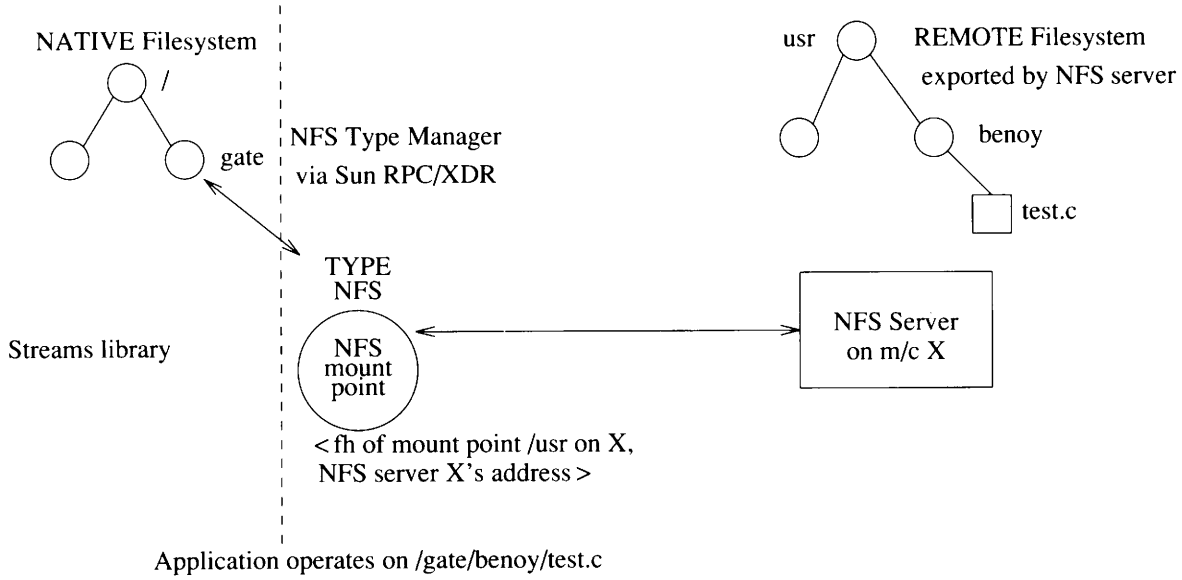


Figure 4: *NFS Client*

removes the need for the kernel to have knowledge of object specific pathname syntax and leads to further extensibilty in naming.

In order to manage **NFS** objects, the NFS type manager maintains state in per-process, ancestor-common and global storage pools (see Figure 5). The per-process data are process-specific items such as the Sun RPC client handle and owner credentials, while the ancestor common information is essentially a stream position pointer and a reference count to track active clients. The global data such as the object's NFS file handle, the parent directory file handle and their associated attributes are stored in the object concurrency data facility in order to regulate access among multiple processes.

6. Implementation of a User-space NFS Server

A user-space NFS server can be implemented as a daemon that invokes native filesystem operations on receipt of an NFS request. In order to increase throughput, it is desirable to run multiple cooperating NFS server daemons. However, to allow consecutive NFS operations on the same file, as in sequentially reading pages from a file, in a multi-daemon environment raises issues that merit additional thought.

It is not deisrable to perform a file open/close on each NFS operation as this introduces a significant expense. The open/close operation can be avoided by keeping the file open for a few seconds after a single request has been completed. Subsequent NFS operations will typically arrive within a second and will thus not require an additional open. The multi-daemon environment introduces an additional complication: the system must keep track of which daemon has which files open.

The design of the multideamon system raised many issues. A decision had to be made regarding whether a single multi-threaded server or separate multiple servers should be constructed. Since NFS operations pass along a <uid, gid> tuple and since in user-space credentials are defined by the process' credentials, it would be difficult for multiple threads to service requests from different users. Hence the NFS server side was constructed as multiple cooperating daemons.

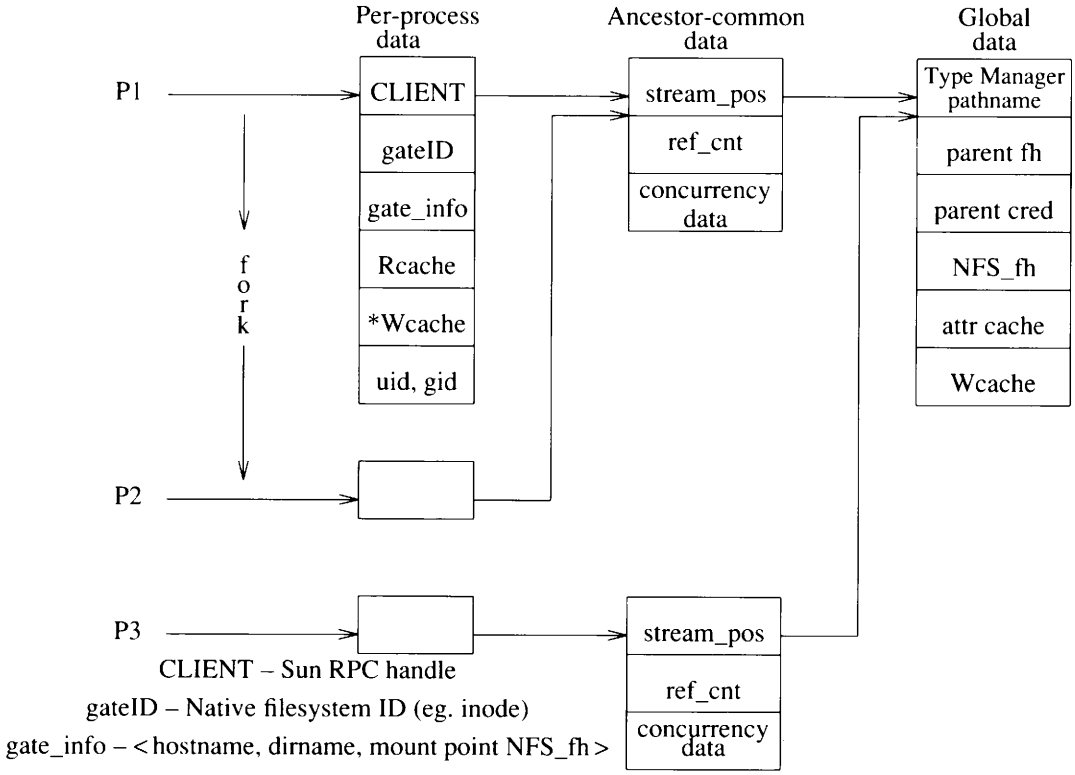


Figure 5: *NFS Client Data Structures*

The multi-daemon system could have been designed such that all daemons waited for requests on a single socket or such that one daemon acted as a dispatcher to other slave daemons. The former approach would have resulted in excessive context switching: all daemons would have been awakened on receipt of a request and all but one would have to return to sleep pending arrival of another request. The latter approach allows a dispatcher to send requests out to slave daemons with little additional overhead. The dispatcher must track which daemon has which files open.

The multi-daemon dispatcher/slave NFS server design is illustrated in Fugure 6. An NFS request described by <file handle, NFS procedure, arguments> and <uid, gid> tuples arrives. The dispatcher validates the <uid, gid> pair and passes the request to one of the slaves. Since the request can be up to 8K in size, the request is not physically sent to a slave but rather stored in global mapped memory described by a slot number. The dispatcher sends the slave the client address and slot number allowing the slave to respond directly to the client after fulfilling the request. The dispatcher/slave architechture involves an additional data transfer of approximately 50 bytes, resulting in a small delay that is far outweighed by the increased availability.

7. Performance

Table 2 shows the results of running the NFS regression tests between two kernel implementations and between kernel and user-space implementations. All times are in seconds.

These results show that in most cases there is no significant penalty for using user-space I/O services. It is important to note, however, that the

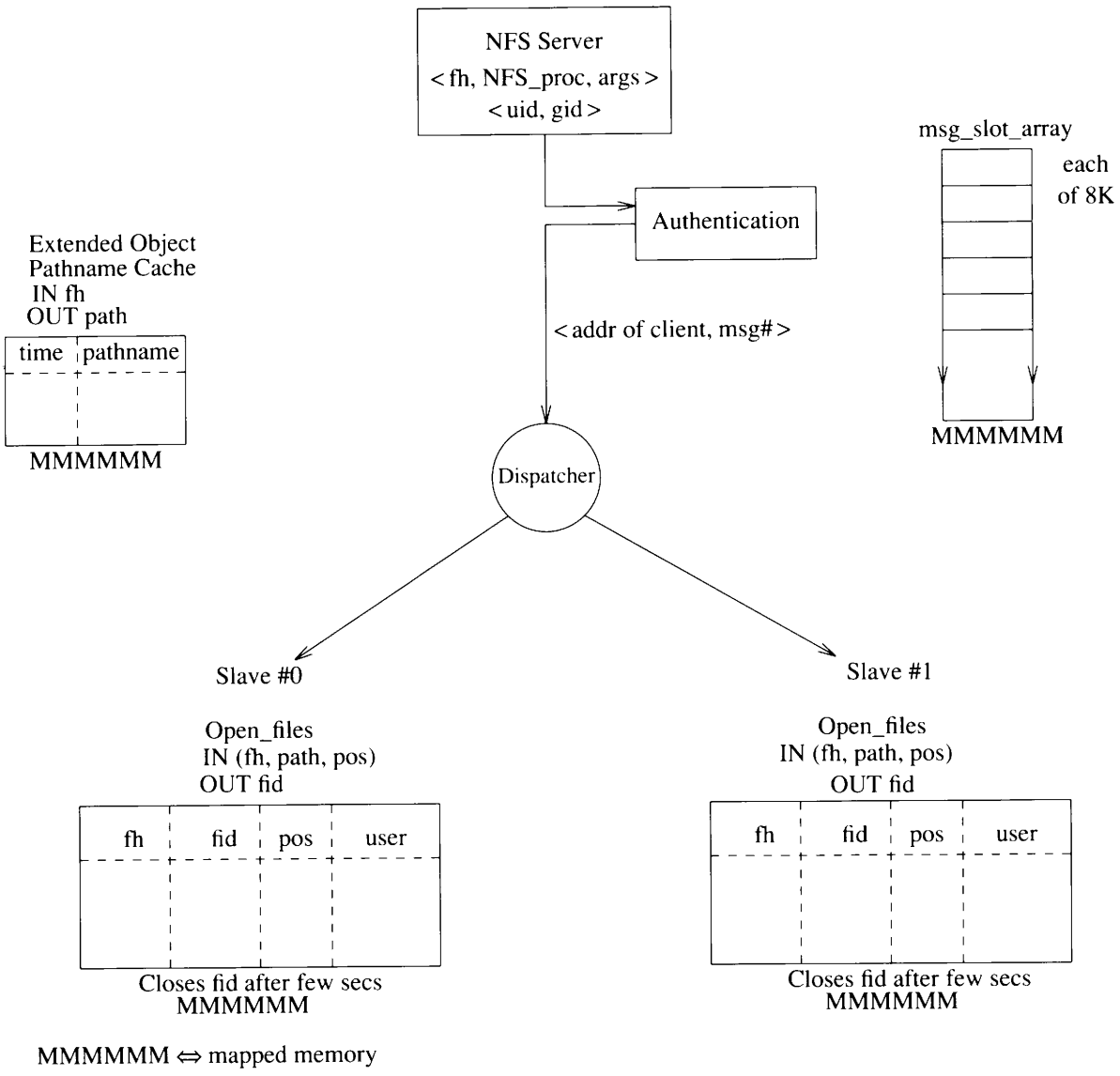


Figure 6: *Multi-daemon Dispatcher/Slave NFS Server*

User-space Client & Server Performance			
Regression Test	Kernel ↔ Kernel	Kernel ← User Server	User Client ← Kernel
File & Dir Creation	36.41	40.1	43.4
File & Dir Removal	33.1	35.2	37.2
Lookup	1.3	1.5	1.9
Setattr, Getattr	40.26	55.2	53.3
Write	215.32	285.46	260.64
Read	3.42	7.5	6.8
Readdir	46.31	57.86	54.72
Small Compile	10.5	12.4	13.3
Tbl	1.1	1.3	1.4
Nroff	5.8	6.1	6.9
Large Compile	14.2	18.2	20.2
4 Large Compiles	65.3	73.6	high

Table 2: *Client and Server performance*

lack of a BIOD facility has resulted in degraded performance for tests involving many reads and writes.

8. Future Optimizations

Both the NFS client and server can be optimized in a fairly straightforward fashion. In particular, a user-space client BIOD daemon could asynchronously perform read-ahead and write-behind, storing the resulting information in the object's concurrency data. On the server side, a cache can avoid duplicate work in the case of lost replies. It is, in essence, a work avoidance technique [Jus89a] that both increases server bandwidth and avoids destructive reapplication of non-idempotent operations such as write.

9. Summary

This paper has shown that given an extensible I/O system it is possible to build new user-space I/O services such as NFS, with a minimum of effort, achieving greater portability with no significant performance degradations.

References

[Gol90a] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid, "Unix as an Application Program," *USENIX Conference Proceedings*, Anaheim, CA (June 1990).

[Jus89a] Chet Juszczak, "Improving the Performance and Correctness of an NFS Server," *USENIX Conference Proceedings*, San Diego, CA (February 1989).

[Kle86a] Steve Kleiman, "Vnodes: An Architechture for Multiple File System Types in Sun UNIX," *USENIX Conference Proceedings*, Atlanta, GA (June 1986).

[Ree86a] Jim Rees, Paul Levine, Nathaniel Mishkin, and Paul Leach, "An Extensible I/O System," *USENIX Conference Proceedings*, Atlanta, GA (June 1986).

[San85a] Russel Sandberg, "Design and Implementation of the Sun Network Filesystem," *USENIX Conference Proceedings*, Portland, OR (June 1985).

XEUS Director: A Distributed Shell for an Intelligent Terminal System

Laszlo Biczok Kalman Szeker

Central Research Institute for Physics
Budapest, Hungary
h1096bic@ella.UUCP h1097sze@ella.UUCP

Abstract

XEUS is a UNIX based intelligent terminal system that makes it possible for PC (DOS) users to work under a more powerful operating system without having to change from their familiar user interface and hardware environment. Additionally, the system provides methods for a UNIX program running on the host machine to download routines to a terminal. By offering synchronization and parameter passing facilities, XEUS can support distributed processing.

XEUS Director is a distributed shell that works on the XEUS intelligent terminal system. It is designed to provide a common user interface for both the UNIX and the DOS mode of the terminal while fully utilizing the features of the XEUS system.

1. General Description of the XEUS System

XEUS is a software environment that makes it possible for PC based systems to be more powerful than individual or networked MS-DOS machines.

The XEUS system is based on a host machine running UNIX, surrounded by IBM PCs or compatibles used as intelligent terminals under MS-DOS. The computers are connected over an ARCNET network, which provides an appropriate data transfer rate for host terminal communication.

The goal of the system is to integrate the well known DOS environment into the UNIX one, while maintaining the advantageous characteristics of the DOS system.

Therefore the XEUS system:

- Makes it possible for DOS programs running on terminals to access, under appropriate protection, the UNIX file system;
- Unloads the UNIX host by using the PCs as intelligent terminals, in this way increasing the maximum number of users that can be handled by the host;

- Makes a DOS like user interface possible for UNIX applications by expanding the UNIX user interface with PC services;
- Supports distributed processing between the host and the terminals. By using XEUS library calls it is possible to utilize the advantageous characteristics of both the UNIX and the DOS environment.

The "heart" of the system is the special terminal emulator. This program is an ANSI terminal emulator, capable of supporting all the features a PC can offer, including total keyboard and screen handling. Additionally, the terminal emulator – with the help of a C library supplied with the XEUS package – makes it possible for processes running on the host to download routines and even complete programs to the terminal. These programs (or routines) can receive and send parameters from/to the process that invoked them (synchronization and parameter passing).

Taking into account the process downloading possibilities, XEUS can be regarded as a special kind of distributed system.

The XEUS system can simplify the change from DOS to UNIX in the IBM PC world. Using the unique facilities of the terminal emulator this "painless" change can be combined with better performance. The special features provided by the XEUS system are designed to support distributed processing and, by using these capabilities, an advanced programmer can write programs that combine the unique features of the UNIX and the DOS systems.

2. The Need for a Distributed Shell

The XEUS terminal emulator program provides an alternative to the terminal user: working under UNIX with the emulator, or running DOS and using the UNIX file system of the host computer – as a DOS logical drive – for background storage. This means that users who wish to fully utilize the capabilities of the system have to deal with two different operating systems – not to mention the special XEUS services; it is not very convenient to work in such an environment. While developing the XEUS system, the need soon emerged for an integrated environment that provides a common, easy to use, user friendly environment for both alternatives.

XEUS itself is not a distributed, but rather an intelligent terminal system. In this way it does not have all of the symptoms given by Schroeder to define a distributed system. XEUS has, however, symptoms that are similar to those can be found in a distributed system.

The XEUS system

- Consists of *multiple processing elements*, that can run independently, i.e. each processing element (the UNIX host and the IBM PCs as terminals) contains at least one CPU and memory;
- Has *interconnection hardware* to make communication possible between the processing elements. It allows processes (UNIX processes and downloaded intelligent routines) running in parallel to communicate and synchronize;
- Is structured in such a way that *processing elements fail independently*. This means that even if a host machine fails, terminal (IBM PC) users have the possibility to continue working on their computer under DOS, or to connect to another host if there is any.

The only symptom XEUS does not have is that the nodes do not keep *shared state* for the distributed system. In this way a node failure causes some parts of the system's state to be lost. If a host crashes, the information in it is inaccessible until the host comes back up; and when a terminal crashes, the whole of its internal state is lost.

From another point of view the XEUS system does not provide transparency in the manner defined by Tanenbaum and van Renesse, i.e. terminal users do not view the system as a "virtual uniprocessor" but rather as a collection of distinct machines. Moreover, in the base system terminal users see only the host machines attached to the network, but they know little about other terminals, and there is no way for them to attach themselves to other terminals or copy files from there, etc.

While keeping shared state for the system would require major changes in the XEUS concept, the utilization of the process downloading facilities of XEUS makes it possible to provide a more transparent view of the system. Moreover, if the system supplies primitives for terminal-terminal interaction, users are able to view XEUS as a "virtual uniprocessor" system.

3. Capability Code System

The XEUS system's Remote Procedure Call mechanism is based on a method called Capability Code System (CCS). In order to describe this method at first we must take a look into XEUS's process downloading facility.

XEUS makes it possible for UNIX programs running on the host to download (DOS format) routines or even complete programs to the terminal. As XEUS does not provide compiler level RPC support these programs can receive and return parameters from/to the host process through library functions supplied with the system. Processes to be downloaded must be complete DOS style executable programs. Such a remote procedure gets its parameters in the "main" function's parameter list similarly to the "`argc, argv`" parameters in common C programs. In XEUS, however, the main function looks something like this:

```
void main( int command, union pars *parameters )
```

where *command* is a request code supplied to the procedure by the system, and *parameters* is a pointer to the actual parameter structure sent by the host process.

Furthermore the main function of the downloaded program looks like a big "case" table which is "switched" on the *command* parameter. The program should support at least two command values: XS_CCS_INIT and XS_CCS_TERM.

When a UNIX process loads a program into the terminal's memory, the XEUS system immediately calls that program with XS_CCS_INIT in the *command* parameter. At this point the downloaded program has the opportunity to process all of the initializations necessary for the correct execution (memory allocation, initializing variables, etc.). Moreover, the remote procedure can define the command values it supports. These command values are called "Capability Code". The host process can then invoke a remote procedure by "requesting" it from the system.

As XEUS does not provide compiler level RPC support there is no way for compile time parameter checking between the host and the terminal routines. In order to support some parameter checking, the system requires the downloaded program to define its parameter structure

along with the capability codes. This method makes it possible for XEUS to provide run time parameter check, in order to locate parameter passing errors and to make developers' life longer. This checking can be disabled by the developer to gain better performance after the debugging sessions.

The XS_CCS_TERM code is supplied to the procedure in case the host process terminates, or when the system detects fatal errors that make error free communication impossible between the remote processes. If a downloaded routine gets this *command* parameter, it should clean up (free allocated memory, release interrupts, etc.) in order to enable the system to remove the program safely from the memory.

The current version of the system supports capability codes between the terminal on which the remote procedure resides and the host process which loaded the program into the terminal's memory. There is no real reason against implementing network wide capability codes as well.

4. The XEUS Director

The XEUS Director is a distributed shell that works on the XEUS intelligent terminal system. It is designed to provide common user interface for both the UNIX and the DOS mode of the terminal while fully utilizing the features of the XEUS system. The shell consists of a UNIX and a DOS program, communicating over the network.

The UNIX part:

- Makes it possible for the user to use UNIX commands,
- Handles the UNIX file system requests of the shell,
- Provides access to the local storage of the other terminals,
- Keeps track of the maintenance operations, changes, etc. made by the shell itself.

The DOS part:

- Provides an easy to use user interface,
- Makes it possible for the user to use DOS commands,
- Handles the DOS file system requests of the shell,
- Handles the local storage requests sent by other terminals.

In order to provide better memory usage, the DOS part of the XEUS Director was developed by using dynamic linking techniques.

The shell has a window and menu driven user interface (with mouse support) to hide the operating systems' details from the user. The shell, however, makes it possible for the user to use all the UNIX or DOS commands in the same manner as he/she would use them normally.

The XEUS Director provides methods for users to combine UNIX and DOS commands. This can be done by writing UNIX shell scripts and using the DR (dosrun) utility supplied with the XEUS system, or by using the XEUS Director's macro language.

The system enables the user to link files and actions (executable files) together. The appropriate action is activated if the user selects a file associated with a valid action.

The shell makes it possible to "tie" files or even directories. This means that every change that have occurred in these files and subdirectories is also made in the "tied" files or subdirectories by the Director.

(The user can use this facility to keep an always up to date copy of his/her files in his/her local storage device.)

5. The Filesystem of XEUS Director

In order to provide common user interface for both the UNIX and the DOS mode of the terminal, and to hide the details of the different operating systems from users, XEUS Director manages a common filesystem for both operating modes of the terminal. This filesystem is totally transparent to the terminal user. And as XEUS is a UNIX based intelligent terminal system it was obvious that this filesystem must be the UNIX one.

Why the UNIX filesystem? When the user turns on the terminal, the system immediately loads the XEUS terminal emulator. The user should login into UNIX (of course) to get access rights to the filesystem. It seemed to be natural to use these rights in terminal-terminal communication, too. By making it possible for XEUS to use the same processing of access rights between two terminals as does UNIX, the XEUS system can benefit from an existing, well understood access method. Moreover, as the DOS style naming method is only a subset of the UNIX one, terminal filesystems can easily be viewed as ordinary UNIX directories. In this way XEUS is capable of showing a unified filesystem both to hosts and to terminals.

The realization of this method is closely related to the UNIX filesystem. A special directory called **xeusterms** is created by the XEUS Director installation procedure. This directory contains nothing apart from some other directories. Each of these directories is logically connected to one of the terminals. The name of these directories must be unique for each terminal. When the user decides to work in another terminal's filesystem, he/she has to open the appropriate directory in the same manner as he/she would work with any other directories. (As XEUS Director is a window oriented shell, this happens by moving the selection bar to the desired directory name and pressing "enter", or by double clicking the mouse on the appropriate name.) If the user has the right to open the directory, he/she is allowed to work in the desired terminal's filesystem. After the successful opening, however, the user does not see the real components of that directory, but the elements of the selected terminal's root directory.

Every directory in **xeusterms** contains a file named **types.ter.** This file holds information on the files residing in that terminal's filesystem. Some part of this information makes it possible for the shell to perform the appropriate action when somebody clicks on that file.

As this file resides in the UNIX host's filesystem while the files on which it contains information stays on the terminal, this file may not hold consistent information. As users can use their terminal without the help of XEUS Director, new files can be created, old ones can be deleted, etc., between two shell sessions. Although the information in the file becomes obsolete, it will be found out in time (e.g. somebody tries to use a file not listed in types.ter). For this reason the XEUS Director treats these files like hints, rather than trying to update them every time when the shell starts. This results in performance improvement in the system.

6. Filesystem Problems

While users work with XEUS Director, they view the system as a continuous hierarchical filesystem. The shell hides all differences, incompatibilities of the different operating systems. Well, almost all of them. There are situations when differences just cannot be hidden.

Let us suppose that a user, while working with XEUS Director, decides to copy a file from one directory to another. While he/she is copying between real UNIX directories (that reside physically in the host machine's filesystem) there will not be any problem. Similarly he/she will find it easy to copy from one terminal to another, or from a terminal to a host. The only situation when something strange happens is when the user tries to copy from a host to a terminal. The reason is the difference of the file naming schemes in UNIX and DOS.

DOS file names length can be a maximum of 12 characters divided into two parts:

- The name of the file, which can be up to 8 characters, and
- The extension, which is maximum 3 characters long.

The two parts of the name are separated by a '.' character which must reside in the 9th position. Taking into account that UNIX has a more flexible naming scheme, it will not take long for a user to try to copy such a UNIX file to a terminal, which name violates the DOS naming conventions. The XEUS system itself had to face this problem in order to make it possible for terminal users to use the host machine's hard disk drive as background storage even in the DOS mode of the terminal. The conclusion was the need for a kind of filename conversion.

The solution which provided satisfactory results in that situation, however, cannot be applied directly to the problem mentioned above. First of all, users will not be able to recognize the copied file with the converted name. Second, there are files that cannot be converted at all.

One solution could be to let users make a decision:

- Users themselves name the new file explicitly, or
- Let the system convert the filename.

Although this method has some drawbacks, it provides an acceptable solution to the problem.

Another problem is that the UNIX access rights method ensures the rights only for the whole filesystem of the terminal. Users cannot determine the rights of individual files or directories residing on the terminal.

Although this problem is not solved in general, XEUS Director makes it possible for users to define which files and/or directories can be seen from another terminal (if it has access rights to this filesystem at all).

7. Conclusion

The XEUS development has indicated the need for a shell that hides the details of the different operating systems in a distributed environment. By providing such an easy to use, window and menu driven shell, users can utilize the services of this environment without too deep knowledge of the different operating systems.

Such a shell can best be implemented by building upon the special characteristics of the distributed system.

The XEUS Director is such a shell built upon the process downloading and RPC facilities of XEUS. Although this shell does not provide every symptom necessary for a real distributed system, it provides a common view of the filesystems of the different machines. Hiding away most differences among operating systems, XEUS Director provides faster success in using the XEUS intelligent terminal system.

Bibliography

1. A.D. Birrel, R. Levin, R.M. Needham, and M. Schroeder. "Grapevine: An Excercise in Distributed Computing". Communications of the ACM 25: 260-274, April 1982.
2. A.S. Tanenbaum and R. van Renesse. "Distributed Operating Systems". ACM Computing Surveys 17(4): 419-470, December 1985.
3. S. Mullender: "Distributed Systems". ACM Press Frontier Series 1989.
4. L. Biczok, K. Szeker, "XEUS: An Intelligent Terminal System," in UKUUG proceedings, Summer 1990

European Forum for Open Systems

The Secretariat
Owles Hall
Buntingford
Hertfordshire SG9 9PL
United Kingdom

Telephone +44 763 73039
Facsimile +44 763 73255
Email europen@EU.net

ISBN 1873611005