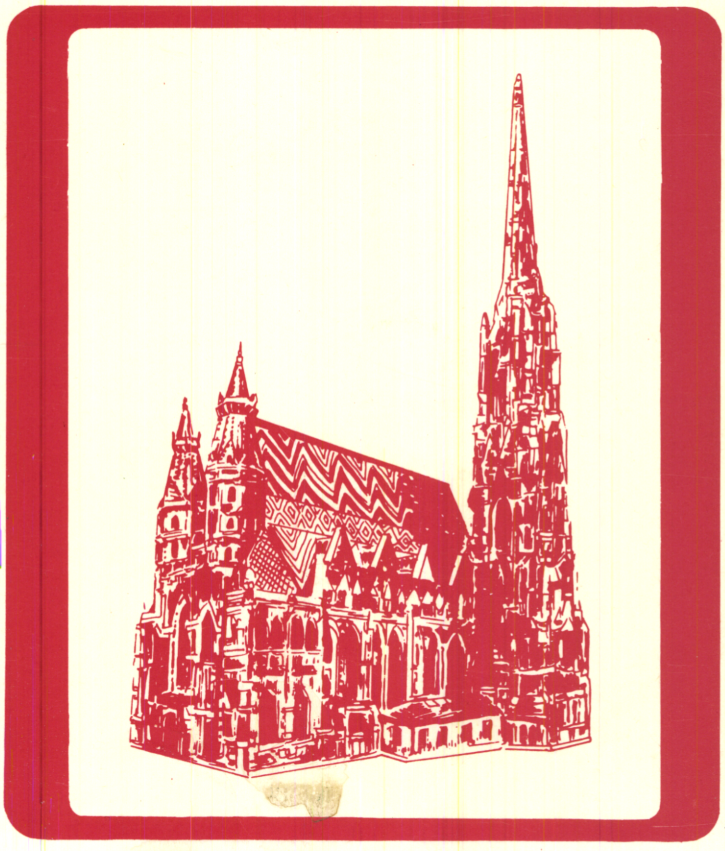
EUUG

European UNIX® systems User Group

Autumn '89

Conference Proceedings

18-22 September 1989
at
Wirtschaftsuniversität
Vienna



E U U G

European UNIX® systems User Group

Proceedings of the Autumn 1989 EUUG Conference

September 18–22, 1989
Wirtschaftsuniversität,
Vienna, Austria

ISBN 0 9513181 3 6

Further copies of the proceedings may be obtained from:

EUUG 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 Sun Workstation. PostScript was generated using **refer**, **grap**, **pic**, **psfig**, **tbl**, **sed**, **eqn** and **troff**. Laserprinters were used for some figures.

ACKNOWLEDGEMENTS

Sponsored by:	European UNIX systems User Group	
Hosted by:	UNIX systems Users Group Austria	
Programme Chair:	eva Kühn	Technische Universität, Vienna, Austria
Programme Committee:	Peter Collinson	Hillside Systems
	Hans Strack-Zimmermann	iXOS Software
Conference Executive:	Ernst Janich	Nixdorf Computer AG
Tutorial Executive:	Neil Todd	GID Ltd, London, UK
Proceedings Production:	Jan-Simon Pendry	Imperial College, London, UK
	Dave Edmondson	Imperial College, London, UK
	Stuart McRobert	Imperial College, London, UK

We would like to thank the UNIX User Group Austria for hosting this event, its chairman Friedrich Kofler of Austro Olivetti and Mrs. Hainschink of the Osterreichische Computergesellschaft. The organisers also wish to acknowledge a contribution from the Olivetti International Education Centre.

The resources used during the production of these proceedings were generously provided by the Formal Methods Group in the Department of Computing at Imperial College.

— jsp, dme, sm.

UNIX Conferences in Europe 1977–1989

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 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

Technical Programme

Keynote

Wednesday (9:30)

Standards

Wednesday (11:00 – 11:30)

Multi-processor Systems

Wednesday (11:30 – 12:30)

User Interfaces

Wednesday (14:00 – 15:30)

Security

Wednesday (16:00 – 18:00)

UNIX in German Speaking Countries

Thursday (9:00 – 9:30)

UNIX Modelling

Thursday (9:30 – 10:30)

RISC Architectures

Thursday (11:00 – 12:30)

Tools

Thursday (14:00 – 15:30)

Network Management

Thursday (16:00 – 18:00)

Graphical User Interfaces

Friday (9:00 – 10:30)

Transaction Processing

Friday (11:00 – 12:30)

Object-oriented Systems

Friday (14:00 – 15:30)

EUUG and USENIX

Friday (15:30 – 16:00)

Author Index

Are Standards the Answer?

Dominic Dunlop

The Standard Answer Ltd.
4, St. Rumbolds Road
Wallingford
OXO.OX10 0DL
UK
domo@sphinx.co.uk

ABSTRACT

Moves are afoot to standardize every aspect of the UNIX world in order that the benefits of open systems can be realised. But what *needs* to be standardized? What *are* the benefits? And who really cares anyway? The answers to these questions turn out to be rather vague, and are not always a good fit onto the standardization activity which has taken place to date.

This paper examines the forces behind standardisation, reaching the conclusion that, while standardisation is a necessary process, it cannot and should not hope to have a significant effect on the diversity of ideas in the field of computer technology — or in any other field.

The UNIX community has been busy trying to standardise itself for the best part of a decade. Starting with the *1984 /usr/group Standard* [USR84a], a document of under a hundred typewritten pages, standardisers have so far churned out around a thousand pages of *System V Interface Definition* [SVI86a], over two thousand of the *X/Open Portability Guide* [ANS89a] and a little over three hundred pages of the POSIX 1003.1 operating system interface standard [IEE88a]. Further POSIX standards already in the pipeline promise a further three thousand page deluge before the end of 1991 — and that is only from the projects which have delivery schedules. Further working groups have yet to decide how much paper they are going to produce, and when.

Setting aside for the moment the question of whether anybody is going to have the time and inclination to read all of this material, it is worthwhile to look at the whole issue from another point of view — that of evolution. The growth in POSIX working groups — twelve so far, with outgrowths into standards for windowing and open systems interconnection — suggests that, as a newly-evolved species, standards bodies have found a habitat free of predators, and are enjoying great reproductive success. Already, those who feed the standards bodies — the companies who send expensive personnel to classy hotels around the world for assignations with other standardisers — are beginning to wonder whether the time has come for a cull. Perhaps, the argument runs, we can achieve the same effect with less effort, and fewer working groups. Perhaps it might even be faster...

At first, it seems that finding a technique that does for standards bodies what DDT does for mosquitos† should be quite easy. Looking at Figure 1, which shows species competing for a habitat, all you have to do is back the ultimate winner, and leave the remainder to starve. Sadly, picking winners is difficult, as highly-profitable legal and illegal gambling industries around the world testify. Besides, gamblers like to play favourites, even after it becomes obvious to onlookers that they are backing a loser. Thus, in the domestic video recorder market, species A might be Sony's Betamax; species B Philips' VR2000; and species C — the ultimate winner — JVC's VHS. Both Sony and Philips, together with dwindling bands of supporters, continued to invest in the less successful systems long after it was clear that VHS had come out on top‡.

† That is, kills off the weak ones, leaving those which remain resistant to attack...

‡ There are actually formal international standards for both VHS and Betamax (IEC 774 and IEC 767 respectively). VR2000 didn't make it...

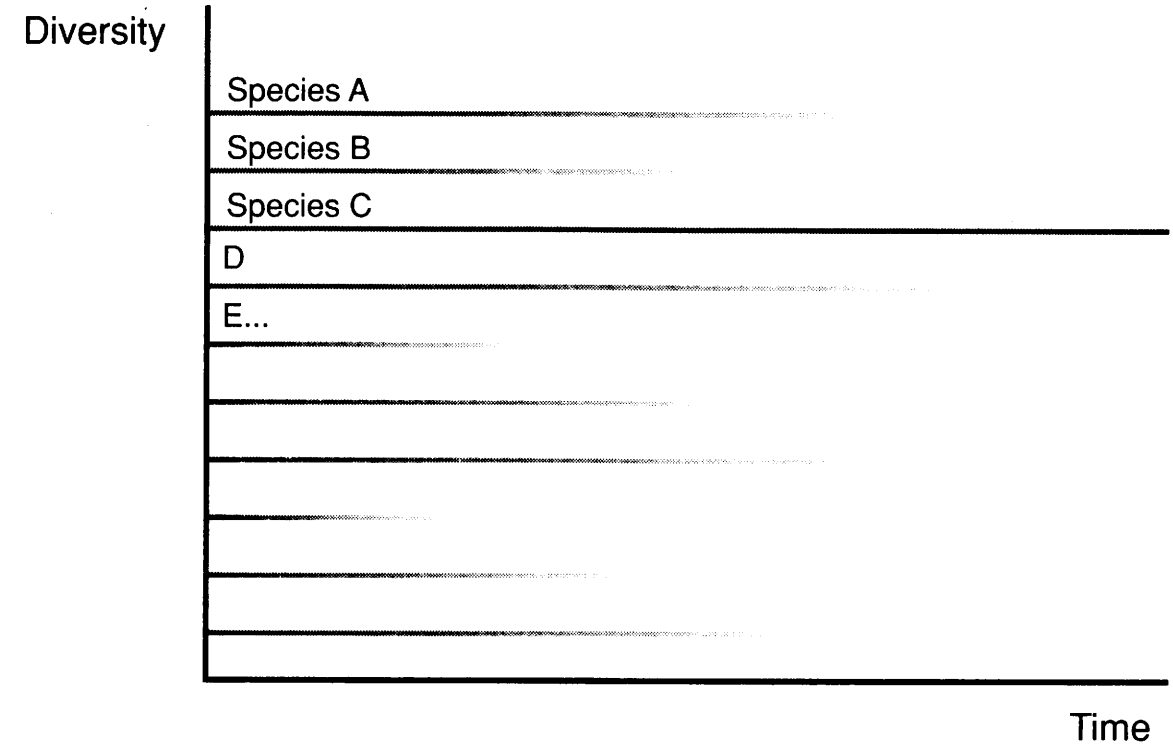


Figure 1: *Evolution — a simplistic view*

Comparisons of standards for the UNIX operating system with those in other markets are interesting. Firstly, such comparisons emphasise that the market for information technology obeys the same rules as other markets. Secondly, it underlines the fact that technical excellence does not drive markets, but politicking does: VHS is technically inferior to Betamax, but JVC's greater willingness to sublicence the technology resulted in more VHS hardware and software appearing in the shops, with the result that VHS conquered the evolutionary niche under your television set.

Pinning some more names on Figure 1, perhaps species A might be the MS-DOS operating system, species B BSD UNIX, C UNIX System V, D Xenix, and so on. As time goes on, MS-DOS slowly dies out — possibly because it has a tendency to choke on the large applications which UNIX can get its jaws around. UNIX System V reigns supreme, having shown BSD and Xenix the door.

Or has it? Life is not actually like that. As we know, although the features and name of UNIX System V are in the ascendent in the commercial market, it has not supplanted either BSD or Xenix. Instead, thanks to pressure from the POSIX standardisation effort, and to help from Sun Microsystems, it has absorbed features from BSD. Similarly, bowing to Microsoft's persuasive argument that it is worth taking notice of anything that can sell 400,000 copies, AT&T has incorporated Xenix functionality into System V.

Figure 2 shows a more correct view of the evolutionary situation: species which are related closely enough to interbreed do so, producing an offspring which is more fit to survive. Of course, it may take some time to overcome prejudices about interbreeding, but rationalisation wins out in the end. Perhaps it's something to do with hotel rooms.

Taking another example from the entertainment industry, back in 1948, RCA and Columbia were at each other's throats, promoting seven inch 45 RPM and twelve inch 331/3 RPM microgroove records respectively. Both were aiming to displace the 78 RPM disk — and each other. But eventually, the realisation dawned that the purchasing public would not put up with two incompatible formats, and that there was enough room in the market for both — provided that reproduction equipment could handle both, rather than just one or the other, ads had been the case when the formats were introduced. The two companies took only a year to bury their differences and a common international standard eventually resulted†. The 78 RPM disk duly faded away. It only took about fifteen years for it to disappear entirely.

† IEC 98, should you be interested.

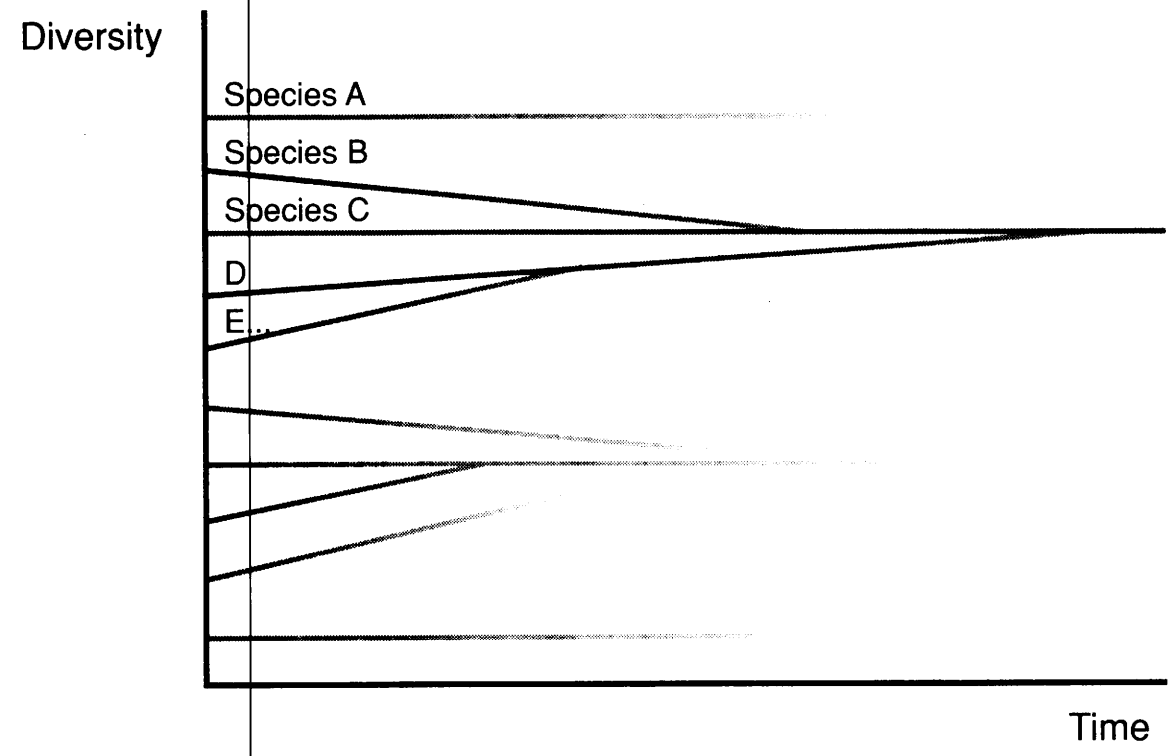


Figure 2: *Evolution — a more correct view*

This brings us to another problem with Figure 2. If species A is MS–DOS, it is clear that it is not going to fade away quickly. Looking back at the micro-computer market, it is clear that MS–DOS displaced CP/M as the operating system of choice for personal computers used in business. Yet CP/M is still around, in low-cost dedicated word-processors and in the hobbyist market. Having been displaced from its original habitat by a descendent species which proved more fit, CP/M refused to roll over and die. Instead, it sought out a new niche.

This happens a lot. Philips developed the Compact Cassette for low-quality dictation, an application for which it is little used today, as micro-cassettes give equally low quality, but are smaller. The Compact Cassette, meanwhile, has migrated to the much richer habitat of the entertainment industry, and is currently the most successful species among carriers of pre-recorded music. Clearly, any picture of the way that technology evolves must accommodate this sort of migration.

Figure 3 shows this awful reality. Mapping computer operating systems onto the picture, BSD UNIX and Xenix still merge with System V (species B, D and C respectively), but a new species (J on the diagram), Mach, breaks away from BSD shortly before the merger. (Mach is likely to rejoin the main line by the end of 1990.) MS–DOS (species A) makes a dive to the low end of the market, but spawns a child (species K) that can hold its own against many applications of the UNIX operating system. In fact, even OS/2 may join the main line, since Microsoft has announced its intention to graft a POSIX interface onto its operating system. Even as this happens, species I — call it the Macintosh operating system — pops into the picture from nowhere, and stays aloof, doing nothing more than picking up a few genes from ProDOS for the Apple][, a distant relative.

As usual, there is nothing special about the computer industry. In the world of video, VHS has spawned super VHS, Sony has come up with Video 8, and Philips has launched laser discs several times. In each case, technological advances are at the root of the changes: Super VHS gives higher recording quality by using better tape than the original VHS; Video 8 uses the same tape to achieve VHS-quality recordings while using less tape. Laser discs use a completely different technology from anything that went before.

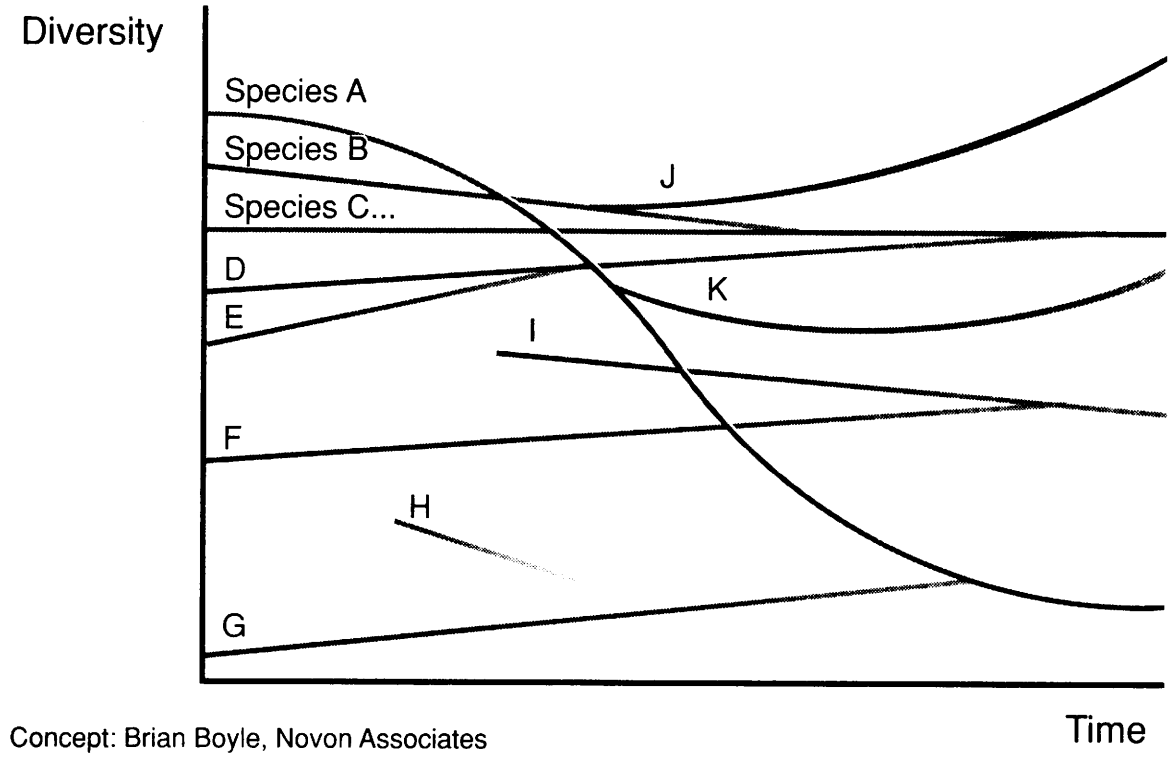


Figure 3: *Evolution — the awful reality*

There is one difference between the entertainment industry and the computer industry. Experience suggests that a successful consumer entertainment technology can expect a life of thirty years or more before its popularity begins to wane. The industry's thoughts are only now turning to high-definition television‡, after 525- and 625-line systems have been in use since the middle of the century. Vinyl disks are only just succumbing to the threat presented by the Compact Disc. The Compact Cassette is safe for the moment, although it remains to be seen how long music industry politics can put a brake on progress, with every consumer electronics company that can muster the research funds working on recordable CD media.

In the computer industry, progress is faster. Thirty years ago is pre-history. Nothing lasts that long. Or does it? What about FORTRAN, born 1954, and far from dead yet [Bas86a]? What about COBOL, vintage 1960? IBM mainframe operating systems, with roots in the early sixties? And what about UNIX, now twenty years old, and, despite signs of middle-age spread†, never more successful. The fact is that the rate of technological progress in computer software is similar to the rate of progress in recorded entertainment, and perhaps for similar reasons: computer users want to be able to run their old programs, just as consumers want to play their old recordings. It takes something pretty special to make either group rush out and rebuild their existing collections of data. By this token, CDs are something special, and the movement towards open systems may just be an analogue in the computer industry.

The thirty-year lifetime, then, appears to apply to computer programs, and to the data that they manipulate. It clearly does not apply to computer hardware, or even to the media on which data is stored. Where now are the eight inch diskettes of yesteryear? It is interesting that there is a long history of technology migration from the consumer electronics industry to high-density media for computer data storage: Compact Cassettes have been used — without, it must be said, great success — as a back-up medium, or even as a random-access device; Video 8 and Digital Audio Tape are seeing increasing use as data storage media which make the capacity of industry-standard half-inch tape look pretty silly; and Compact Disks

‡ HD TV, if and when it arrives, will benefit computer purchasers by slashing the cost of high-quality display devices. But, guess what. There's a standards war between the U.S.A., Japan, and Europe holding up progress. Plus ça change...

† SPREAD was also the name of the IBM task-force which laid the foundations for the 360 mainframe computer range...

provide so much read-only storage capacity in a mass-producible form that most people are still working out applications for them.

Of course, this intrusion into the computer world of technology developed for mass entertainment would not be possible without the adoption by the consumer electronics industry of digital techniques. If the trend continues, the next big leap in media capacity for computers will come shortly after the development of a domestic digital recording technique for video signals!

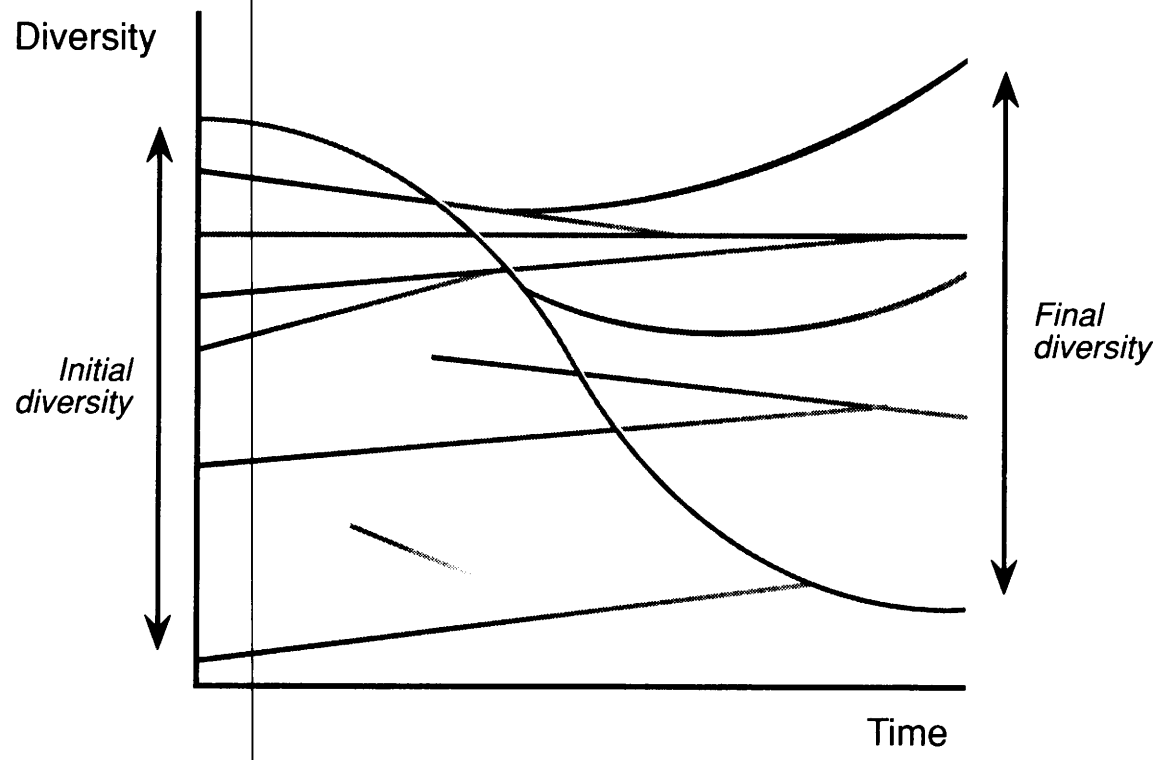


Figure 4: *Evolution brings little change in diversity*

An important aspect of Figure 3 is that the diversity of species at following the process is almost the same as it was at the start: the number of niches open for exploitation by successful species is little changed. Figure 4 illustrates this.

Looked at from the point of view of standardisers this is either good news or bad news. Taking the bad news first, standardisation is bound to be unsuccessful: as soon as one issue is nailed down, another springs up to vex those who seek uniformity in all things. But the good news is that standardisers are never going to be short of work!

The situation is also good news and bad news from the point of view of those who abhor standards as taking the fun out of everything. Standards seem to be a necessary part of the evolutionary process, eliminating needless inconsistencies — or perhaps, small but interesting differences, depending upon your point of view — between species which are trying to occupy the same niche. Certainly, a successful standard will fight off subsequent attacks on the niche it occupies — who remembers three-inch diskettes or Sony's Elcassette, for example — but it should not preclude development in other fields.

In fact, a successful standard actually aids progress and innovation in other fields. As part of M.I.T.'s Project Athena [Tre88a], the UNIX operating system nurtured the birth of the X Window system [Sch87a]. With this successful childhood behind it, X is now strong enough to see off competitors of its own. Similarly, even before the POSIX working group enshrined the operating system in legalistic language, UNIX gave the Corporation for Open Systems (COS-) a stable base for the testing of Open Systems Interconnect (OSI) protocols Q despite the UNIX community's continuing love affair with TCP/IP, and despite OSI's applicability to many other operating environments. The very existence of a standard environment allowed COS- to get on with the work in hand, rather than worrying about the structure needed

to support it.

Showing that cross-fertilisation in the both directions is possible, emerging standards for OSI network management have proved unexpectedly useful in the development of techniques for the administration of POSIX-compliant systems, and the POSIX distribution services (that is, networking) group has found itself spawning efforts to define standard programming interfaces to OSI protocol stack layers. The IEEE 1201 project, recently started to standardise aspects of the X Window System, also has strong links with POSIX.

Standardisation, then, seems to be a necessary — if not sufficient — part of progress in information technology. It also seems to be at least as efficient as the "market forces" about which some of us have heard so much in recent years. That is to say, a standards effort may sometimes do more than is strictly necessary, just as markets may, on occasion, provide too many goods. An example here is the IEEE 1003.2 working group's standardisation of the UNIX shell command language and utilities, a gargantuan task which has absorbed much time from some of the industry's most capable minds. But is it really needed? Is it a winner worth betting on? Who can tell. Only the wisdom of hindsight can yield the correct answer, just as it took commentators several years to realise the Compact Disc had settled in for a long stay. Standards are like any other sort of product: you can do the market research to establish what is needed, you can work out the time-window for delivery, and you can develop and ship the goods. But, in the end, as with markets, and as with evolution, it is large forces beyond the control of any single group which determine whether the ultimate standard is accepted or consigned to oblivion.

References

[USR84a] /usr/group, */usr/group Standard*, 2901 Tasman Drive, #201, Santa Clara, CA 95054, U.S.A, 1984.

[SVI86a] AT&T, *System V Interface Definition, Issue 2*, 1-3, 1986.

[Bas86a] Basche *et al.*, *IBMs Early Computers*, M.I.T. Press, 1986.

[ANS89a] American National Standards Institute, *ANSI X3.159-1989, Programming Language C*, 1-7, Prentice-Hall, 1989.

[IEE88a] *IEEE/ANSI Standard 1003.1-1988: Portable Operating System Interface for Computer Environments (POSIX)*, The Institute of Electrical & Electronic Engineers, 1988.

[Sch87a] R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, April 1987.

[Tre88a] G. Winfield Treese, "Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3 BSD," *Proceedings of the USENIX Winter Conference*, USENIX Association, Berkeley, CA, 1988.

Engineering a (Multiprocessor) UNIX Kernel

Michael H. Kelley
Andrew R. Huber

Data General Corporation
Research Triangle Park,
North Carolina 27709
USA
kelleymh@dg-rtp.dg.com
huber@dg-rtp.dg.com

ABSTRACT

This paper describes the software engineering aspects of the DG/UX™ kernel, a symmetric multiprocessor UNIX kernel that runs on the Data General AViiON™ and ECLIPSE™ MV families of computers. The DG/UX kernel is a completely new implementation (not based on AT&T or Berkeley source code) that has applied modern software engineering techniques to improve the structure and modularity of the code. The result is a kernel that is more reliable, easier to maintain, and easier to enhance.

The paper discusses four major themes that have guided the engineering of the kernel to achieve the reliability, maintainability, and enhanceability goal. It describes how the themes of *hierarchy*, *information hiding*, *conventions*, and *tools* have been applied throughout the kernel and implemented using the standard C language and UNIX development environment. Finally some practical experiences in using this method to produce the finished DG/UX product are described.

1. Goals

Work on the new DG/UX kernel began in late 1984 with two main goals. The first goal was to run on multiprocessor hardware in a fully symmetric fashion with multiple processors executing in the kernel simultaneously. The second goal was for much better reliability, maintainability, and extensibility compared with traditional UNIX kernels that are derived from AT&T or Berkeley source code.

At the time the project began, Data General had an existing UNIX kernel that became the first release of DG/UX on the ECLIPSE/MV family of 32-bit minicomputers in early 1985. This kernel was based on AT&T System V source code with the Berkeley 4.2 file system grafted in. The device drivers were heavily modified versions of drivers from AOS/RT32 (a Data General real-time operating system) and were written in assembly language. The virtual memory implementation was newly written for DG/UX. Later versions of this kernel included the Network File System (NFS™) from Sun Microsystems.

The uniprocessor nature of the existing kernel suggested that a complete rewrite would be necessary to achieve the desired goals. None of the code had been written with multiprocessors in mind. While others have added locks to traditional UNIX kernels to make them run on multiprocessor hardwarel [Bac84a, Ham88a], we felt that such a kernel could not meet our goals of a high degree of concurrency in kernel execution and effective use of more than just 2 or 3 processors.

The existing source code also presented significant problems in producing a kernel that was reliable, maintainable, and extensible. The traditional source code from AT&T or Berkeley is not known for being clear and well organized [Hen84a]. Changing one part of the code can have subtle effects on other, seemingly unrelated, parts of the code. These complex and obscure relationships make it difficult to find and fix bugs or make enhancements without creating other problems. Including code with other origins creates additional awkward relationships because such code doesn't always fit well together. Resolving these problems also suggested that a complete rewrite of the kernel would be desirable.

This paper will concentrate on the software engineering used to meet the reliability, maintainability, and extensibility goal. Achieving the multiprocessor goal has been discussed in an earlier paper [Kel89a] and will not be covered further here except when it motivates a specific software engineering technique.

2. Major themes

Four major themes have guided the engineering of the new DG/UX kernel:

- hierarchy
- information hiding
- conventions
- tools

These themes are not new and have been widely discussed in the literature. Their application, though, is relatively less common, particularly in the UNIX environment. One of the few examples is TUNIS, a UNIX kernel written with many of the same goals that we had [Hol83a].

The underlying motivation of these themes is to organize a complex piece of software (a multiprocessor UNIX kernel, in particular) in a way that the whole or any useful subset of the whole can be clearly understood. Such understanding is the prerequisite for reasoning that it is correct, for fixing a problem without introducing a new problem, or for adding an enhancement.

Each of these themes will be introduced below. Later sections of the paper will give details of how they have been applied to produce the new DG/UX kernel.

2.1. Hierarchy

The hierarchy theme suggests that a large software entity should be divided into a collection of smaller entities. Each member of the resulting collection can itself be divided into still smaller entities. This process of dividing can be applied recursively as many times as appropriate. The resulting hierarchy describes the structure of the original large software entity. At each level of the hierarchy, the division into smaller entities is made by grouping related sections together. Each of these smaller entities is easier to understand and work with because it is smaller than the whole and because it contains only closely related sections.

In the DG/UX kernel, the kernel as a whole is the root of the hierarchy. The kernel is divided into about 50 *subsystems*, each of which constitutes a major functional area of the kernel. A subsystem is further divided into *modules*, each of which is a C language source file. A module is divided into functions and type definitions. These four levels are the major structure for organizing the kernel and for applying the other software engineering themes.

2.2. Information hiding

The information hiding theme implies that a software entity should be separated into its *interface* and its *implementation*. The interface describes the function that is provided, including any assumptions or constraints, but contains no information about how the function is provided [Par72a, Par72b]. The implementation, which consists of the algorithms and data structures needed to support the interface, is hidden from its user.

Hiding the implementation from its users has several benefits. First, the implementation can be changed without affecting the user, if the interface is kept the same. Second, the implementation code and the using code can be worked on independently, with the interface forming a contract between the two. Also, just requiring that the interface be explicitly stated may bring hidden assumptions and requirements into the open where they can be considered and either made part of the interface or explicitly disallowed.

The information hiding idea is a key component of the object-oriented approach to programming. An object has an interface describing the operations that other objects may invoke, and it has an implementation consisting of code and data that is hidden from other objects. This same notion is present in languages such as ADA which incorporate the interface/implementation paradigm directly into the language.

In the DG/UX kernel, the information hiding theme is applied at each level of the kernel hierarchy. For the whole kernel, for a subsystem, for a module, or for a function or type definition, there is an interface and an implementation. For example, each subsystem has an interface containing functions and type definitions that it presents for use by other subsystems. The data structures and functions used for the implementation of a subsystem are hidden within that subsystem and may not be accessed by other subsystems.

2.3. Conventions

Conventions are used in a software entity to increase the uniformity of the resulting product by encouraging developers to do the same thing the same way. This uniformity produces several advantages. First, time and effort are saved because there are prescribed patterns for doing things; developers don't "reinvent the wheel". One developer can look at another's work and feel comfortable with it because it follows the same conventions. Second, conventions promote sharing and reusing code by reducing incompatibilities caused by two developers doing the same thing in slightly different ways. Third, since the prescribed patterns have been carefully checked, errors are also reduced. Fourth, conventions facilitate the use of tools and other automation in the software development process.

Several kinds of conventions are used in the DG/UX kernel. Naming conventions govern how subsystems, modules within a subsystem, and functions and type definitions within a module are named. Structural conventions govern how subsystems are related and divided into modules. Coding style conventions govern how C language constructs are used and not used. Documentation conventions specify documentation that exists at the subsystem, module, and function levels.

2.4. Tools

The tools theme encourages the use of software tools and hardware processing power to enable developers to be more productive. By making a fixed investment in tool development and in hardware to run the tools, developer time and elapsed development time can be reduced. The idea is to spend processor cycles, not people.

A number of tools and techniques were created during the DG/UX development, including: error checking tools that look at source modules to detect type mismatches, violations of conventions, or dependency loops between subsystems; and text processing tools that extract documentation from source modules to produce design documents at the subsystem or module level. The kernel "build mechanism" automates the determination of dependencies between subsystems and modules and ensures that the proper modules get recompiled when a data structure changes.

3. Applying the Hierarchy Theme

The DG/UX kernel is divided into a hierarchy with four levels: 1) the kernel as a whole, 2) subsystems, 3) modules, and 4) individual functions and type definitions. This division is summarized below.

3.1. Structure of the kernel

The kernel as a whole is divided into some 50 subsystems, each constituting a major area of kernel functionality. Each subsystem implements a related set of functions that operate on one or more related complex data types. A subsystem contains functionality that can be understood and mastered by one developer and that is a natural unit on which to do development work. Some subsystems providing relatively simple utility functions consist of only 3 or 4 source files. Other subsystems supply major kernel functionality and have 60 or more source files.

For example, all of the code for managing file system directories is in the Directory Manager (DM) subsystem. DM includes routines to add an entry to a directory, to delete an entry from a directory, to lookup an entry in a directory, and to list the contents of a directory. When a subsystem such as the Pathname Manager (PN) needs to resolve a pathname, it does so by repeatedly calling a lookup function in DM to locate each pathname component in its containing directory.

Using a subsystem as a unit for development has several aspects. Design and code reviews are typically done on a subsystem because it should be understandable by a single developer. When major functional or performance enhancements are to be made, a subsystem can be modified and tested in parallel, and then substituted as a replacement when it is deemed ready for real use. A subsystem is also a natural unit of substitution for different kinds of computing environments. For example, the Medium Term Scheduler (MTS) subsystem in the DG/UX kernel encapsulates the scheduling heuristics. It is likely that time-sharing and real-time versions of the kernel would have different medium term schedulers, so different versions of the MTS can be used for the different environments.

At the subsystem level of the hierarchy, subsystems are layered in the traditional sense to provide the kernel functionality. Subsystems in lower layers provide services that are used by subsystems in higher layers. The higher layers, in turn, provide the system call interface of the kernel that is used by application programs. Note that this functional layering is independent of the kernel/subsystem/module/function hierarchy that guides the software engineering of the kernel.

3.2. Structure of a subsystem

Just as the kernel consists of subsystems, a subsystem consists of multiple modules (C language source files). The division of a subsystem into modules is guided by two goals. First, functionally related areas of a subsystem are grouped together. A subsystem that implements several related abstractions has several groups of modules, each of which implements one abstraction. Second, the different components of an abstraction are put in different modules to support the use of the C compiler and allow flexible storage allocation in the kernel image. Three categories of modules exist; no source file contains code from more than one category:

- definitions — C language typedefs and #defines of constant values
- data — variable declarations that cause allocation of storage in the kernel executable
- functions — functions and #defines of code fragments

Because header (.h) files contain only definitions, they never cause allocation of storage and hence can be included in several places without causing errors due to multiple allocation of the same storage item.

Data items fall into one of four sub-categories: 1) global data, 2) per-process data, 3) per-processor data, and 4) message data. Data items in each category are put into separate C source files that contain only data items of that type. Global data is globally addressable within the kernel and is grouped together so it can be placed in the global read/write portion of the kernel address space. Per-process data is unique for every process in the system, but addresses are allocated in the kernel space for only one process's per-process data. Using the virtual memory mechanism, this "window" of kernel addresses is remapped whenever a new process is run so that a running process will always find its per-process data in the window. In order for the remapping to work, per-process data from different subsystems must be allocated together. Per-processor data is like per-process data except that different mappings exist for each processor instead of for each process. Per-processor data from different subsystems must be allocated together in the per-processor area of the kernel address space. Message data contains text strings that may be written to the operator's console; it is grouped together so it may be easily translated to other languages and so it may be placed in a read only portion of the kernel address space.

Functions and #defines of code fragments are placed in C modules grouped by the abstractions they implement. Modules containing functions are the only modules that generate executable instructions, and they are placed in an execute only portion of the kernel address space. Write protecting the executable code makes it easier to isolate bugs that overwrite random pieces of kernel memory.

3.3. Structure of a module

A module is a sequence of definitions, variables, or functions, depending on the type of module. It also contains C include statements and a variety of sentinel comments that are interpreted by special tools. At the module level in the hierarchy, little additional structure can be imposed. The ordering of definitions, variables, or functions is constrained primarily by the C language definition.

Since the themes discussed in this paper are ultimately implemented as some method of handling the C source files, many of the details of a source file are best left to the sections that discuss the other themes. Include files are discussed under information hiding because of the key role that they play in implementing information hiding. The sentinel comments and their interpretation are discussed as their specific application is discussed.

4. Applying the Information Hiding Theme

In the DG/UX kernel, information hiding is applied at all levels of the hierarchy. Clear distinction is made between the interface and the implementation of the kernel, of individual subsystems, and of modules within a subsystem. Information hiding is achieved by using this distinction to control the knowledge that different parts of the kernel have about other parts.

The information hiding theme is implemented in the DG/UX kernel using the standard C language with a small increment of additional tools and conventions. While programming languages such as C++ or ADA provide various kinds of support for information hiding, compilers for these languages were not widely enough available at the time the new DG/UX kernel was begun to make their use a viable option. The small set of additional tools and conventions that we used enabled us to achieve almost all of the benefits of information hiding while still using a very widely available and standard programming language.

4.1. Information hiding in the kernel as a whole

Since a traditional UNIX kernel has a well-specified set of definitions and system calls that applications may use, the information hiding theme is evident when considering such a kernel as a whole. These definitions and system calls define the external kernel interface, while the rest of the kernel is private and may not be depended upon by applications. However, two major kinds of violations occur that allow applications to depend upon the kernel implementation. First, the system call interface is incompletely specified, allowing applications to become dependent on unspecified quirks of the implementation. Second, the absence of system calls to support certain important operations has led to applications that read kernel data structures directly out of kernel memory via a "backdoor" mechanism called */dev/kmem*. The new DG/UX kernel has addressed these problems as described below.

The problem of the incompletely specified kernel interface has been addressed in DG/UX by doing extensive work to improve the specification of the system call interface. When the specification failed to state the results of a particular combination of arguments to a system call, DG/UX documentation has added an explicit statement on what the kernel does, based on the AT&T and Berkeley kernel code. Implicit requirements on the kernel implementation heavily depended upon by applications were documented and made an explicit part of the interface (for example, the algorithm for assigning file descriptor numbers to open files). In other cases where applications were not dependent upon an implementation algorithm, cautionary notes were added to the specification stating that an application must not depend on a particular feature of the implementation. This work on improving the specification of the kernel interface has been aided by industry-wide standards development efforts such as POSIX, but these standards do not address all of the system calls so some work has remained.

The problem of applications accessing kernel data structures directly through */dev/kmem* has been addressed in DG/UX by adding some new system calls. Most of the applications that have traditionally used */dev/kmem* are utilities that report kernel status or performance information. An application, such as *ps*, that shows information about all of the processes running in the system would simply read the kernel process table directly from kernel memory and format the data into a conveniently displayable form. The *ps* command must know the layout of the kernel process table, and a new version of *ps* must be created if fields are added or removed from the process table. The new DG/UX kernel has added several new system calls with well specified interfaces that report the kind of information that *ps* and other related commands require. The applications have been rewritten to use these new system calls so that they are now independent of the layout of the internal kernel data structures. The kernel data structures are now truly private to the kernel and can change without affecting any applications.

4.2. Information hiding in a subsystem

A subsystem has an interface and an implementation. The interface is the data structure definitions (e.g., C language typedefs) and functions that are explicitly stated in the subsystem to be available for use by other subsystems. Such definitions and functions are said to be *exported* by the subsystem. The implementation is the set of definitions and functions that are internal to the subsystem and may not be used by other subsystems. Such definitions and functions are said to be *private* to the subsystem. If the interface of a subsystem is not modified, changes may be made to the implementation without any effect on the rest of the kernel.

The interface to a subsystem is realized as a single C language header (.h) file that contains the data structure definitions and external function declarations for everything that is exported by the subsystem. This header file is the union of the interfaces of individual modules in the subsystem that are declared to be part of the subsystem interface. Everything that other parts of the kernel may know about a subsystem is part of that subsystem's interface header file. A subsystem's interface header file is created automatically as described later in this section.

4.2.1. Avoiding global data

By convention, all data in the kernel is private to the subsystem of which it is a part. Instead of exporting a data item, a subsystem exports procedures that define the allowed set of operations on the data item. Exporting routines instead of the data item itself allows the representation of the data to be changed without having to recompile or change the code in all the places in the kernel that perform operations on it. Having every data item "owned" by a subsystem effectively eliminates global data from the kernel.

Having clear ownership of every data item in the kernel is extremely important in achieving multiprocessor operation. In a multiprocessor kernel data structures must be locked before being accessed in order to ensure that the contents are consistent. Ownership of a data item restricts access to the few owning functions so that the lock and unlock operations can be clearly and consistently applied.

A significant improvement in the new DG/UX kernel has been the elimination of global data. In traditional UNIX operating systems two globally accessible data structures, the *proc table* and the *u-area*, are particularly bad in this regard. For both of these data structures neither the overall structure nor the individual fields have clear ownership because all of the fields are accessible to all kernel code. Fields are modified directly from whatever code needs to make some change. While the rewritten DG/UX kernel doesn't have the same proc table and u-area, it does need something logically equivalent to each that meets the goals of subsystem-based design. These goals are met differently for the proc table and u-area as described below.

4.2.1.1. Proc table

The proc table is an array of structures containing various per-process fields, with one entry in the array for each active process. In DG/UX the fields of the proc table have been distributed into several smaller process tables, each of which is owned by a particular subsystem. Each smaller process table is an array of structures containing only fields for which the owning subsystem is responsible. Each active process in the system is assigned a process index that it can use to index into any of the smaller process tables associated with a particular subsystem. The combination of the fields from all the smaller process tables entries with the same process index is equivalent to the old proc table. See Figure 1. This rearrangement of the proc table is possible because there is no reason the fields associated with a particular process must be allocated contiguously.

With the rearrangement of the process table, a subsystem can add, delete, or change a field without affecting the rest of the kernel. In particular, no recompile of the rest of the kernel is necessary when such a change is made. A new and optional subsystem, such as for a networking protocol, can have its own process table fields without creating a dependency on the rest of the kernel. The rearrangement will likely produce better locality of reference when searching through the table for a particular entry or when following links because the individual array entries are smaller. Multiprocessor access is facilitated because each smaller process table can be individually locked by the code in the subsystem that owns it.

4.2.1.2. U-area

The u-area presents the same ownership and global data problems as the proc table. Different from the proc table, though, is the requirement that even unrelated fields associated with a process must be allocated contiguously to permit the remapping operation that makes a process's u-area appear at the right logical address.

The DG/UX kernel uses per-process data as the equivalent to the traditional u-area. Each field of the u-area is declared as a separate per-process variable that is owned by a particular subsystem. Since per-process variables are put in source files that contain no other kinds of variables or code, the object files containing per-process data can be grouped together at link time to ensure that the per-process variables are contiguous in the kernel address space. The resulting contiguous area is effectively equivalent to the u-area. Because the order of fields within the u-area is not important, it is not necessary to control the order in which the linker allocates space for these per-process variables.

As with the proc table, distributing the u-area provides great flexibility in being able to change fields without forcing a recompile of the rest of the kernel. New or optional subsystems can create, modify, or delete per-process variables without creating dependencies on the rest of the kernel.

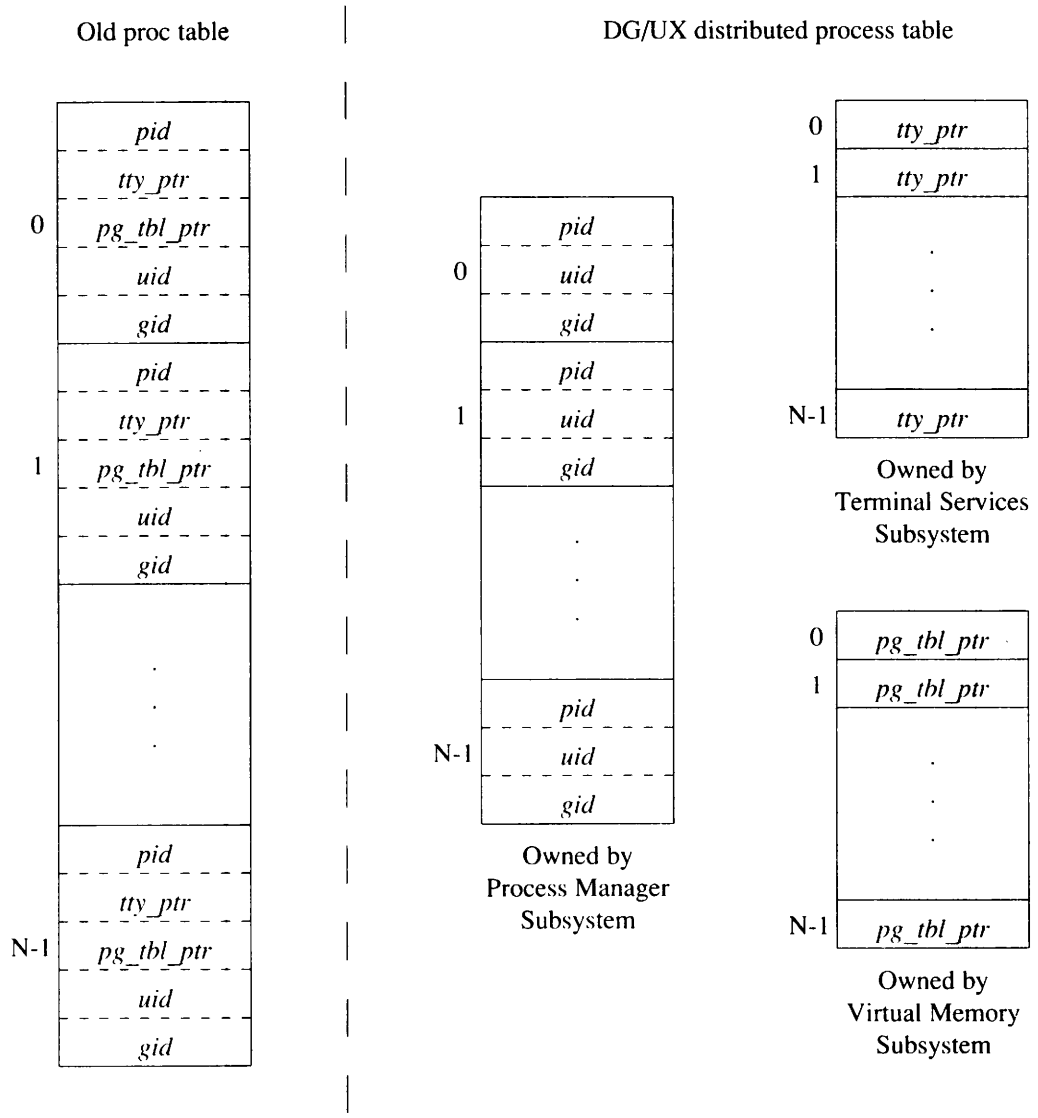


Figure 1: *Distributing the process table*

4.3. Information hiding in a module

As with a subsystem as a whole, each module has an interface and an implementation. Since modules are realized as C language source files, the interface and implementation at the module level are defined in terms of C language constructs. The interfaces, in conjunction with the C language include statements in the source file, realize the information hiding scheme of the DG/UX kernel.

4.3.1. Module interface and implementation

For a header file, which contains only definitions, the interface and the implementation are equivalent. Everything in the source file is part of the interface and part of the implementation. For a source file containing data, everything in the source file except for initialization values are part of the interface. Initialization values are part of the implementation because they are not needed in order to compile other source files that reference the data item. For a source file containing functions and #defines of code fragments, the names of the functions, the types of their return values, and the number and types of their arguments are part of the interface. All #defines are part of the interface. The implementation of the functions, which is usually most of the code in a source file, is not part of the interface.

Prescribed comment blocks are placed in source files around the important C language syntactic elements so a special utility program, called the *extractor*, can recognize and extract the interface of the source file. As part of a kernel build, the utility program is run over every .c source file and produces as output a file with the same name but with .d† instead of .c as the suffix. The extractor is not run over .h files because the interface and implementation of a .h file are equivalent; hence .h files do not have corresponding .d files.

The .d file contains C language external declarations for every variable or function found in the .c file. The keyword *extern* is inserted and the types of the variables and function return values are reproduced as appropriate for the .d file to be included by another source file that wishes to reference the items it contains. Portions of the .c file that are not part of the interface are not reproduced in the .d file.

The .d file could be produced manually using an editor, but whenever editing the .c file changed its interface, the corresponding .d file would have to be manually updated as well. Manually updating both invites divergence between them, as well as being more work. Using the extractor establishes one source file, the .c, with the .d simply being a product of building the kernel.

4.3.2. Include files

C language include files are the main mechanism for achieving the information hiding theme in the DG/UX kernel. Several rules govern the use of include files. First, every source file must be self-contained in that it either defines internally or includes a definition for every identifier it references. A source file containing functions must include a declaration of every variable or outside function that it calls; function return types and variables are not allowed to default to *int*. Declarations for functions implemented by the same subsystem are obtained by including the .d file corresponding to the .c file that implements the function. Declarations for functions implemented by other subsystems are obtained by including the subsystem interface file for the implementing subsystem. Hence a .c file may include .h and .d files from its own subsystem, or the subsystem interface file from other subsystems.

A header file that defines a new structure must define within itself or include a definition for the types of the fields of the new structure. As with function declarations, type definitions that come from the same subsystem may be obtained by simply including the .h file that defines the type. If the type definition comes from another subsystem, the subsystem interface file for the other subsystem is included. Hence a .h file may include other subsystems. It must not (and should not need to) include a .d file from its own subsystem.

Because each source file includes everything it needs, the source file (or, for .c source files, its derivative .d file) can itself be included by a higher level module without requiring the higher level module to have other include statements. Because there can be many levels, large include file hierarchies can result, and a particular include file may be included several times at different points in the hierarchy. To solve this problem, each include file contains a #ifdef and a #define statement that cause the file to be skipped if it has already been included once.

The files included by a particular source file must be grouped into two categories: interface includes and implements includes. Interface includes are the include files that are needed to define the types and constants that are part of the interface of the source file. Implements includes are the additional include files that are needed for the implementation of the source file. A specific include file is included only once — as an interface include if necessary, otherwise as an implements include. For a .h file, all includes are interface includes because everything in the source file is part of the interface. For a .c file, the includes that define the types of variables, functions, and function arguments are interface includes and all other includes are implements includes. The distinction between interface and implements includes is used when .d files are created. Include statements are placed in the .d file for all of the interface includes, but not for the implements includes. This step ensures that the .d file is self-contained; i.e., it includes definitions for all of the types it uses.

† The .d suffix has no particular significance. The letter d was chosen simply to avoid conflict with existing commonly used suffixes.

4.3.3. Achieving information hiding

Information hiding is achieved by the include structure described above. First, since a module only includes .h and .d files from its own subsystem, it only includes the interfaces of other modules, never their implementations. Hence one module cannot get implementation information about another module. Second, a module may only include the subsystem interface file from other subsystems; it may not include .h or .d files from another subsystem. Third, a subsystem has explicit control over which of its .h and .d files are part of its subsystem interface file. These latter two constraints ensure that a subsystem has control over what it exports to other subsystems and that other subsystems cannot access non-exported parts.

5. Applying the Conventions Theme

Conventions are used heavily throughout the DG/UX kernel in order to increase the uniformity of the resulting product even when many developers have worked on it. These conventions also allow the tools to work properly.

The exact convention used is not the important point. Many arguments can be had over the merits of one convention versus a competing convention covering the same area. The important point is that *some* convention be used so that there is uniformity in the results. Also, the conventions are intended to be just that: conventions. They are not mandatory and are not a substitute for the use of good judgement by the developer, though the developer who violates the conventions should be prepared to defend his choice at a design or code review.

5.1. Naming conventions

In order to ensure consistency from subsystem to subsystem, a number of naming conventions are used. These include: using conventional prefixes and suffixes on C source file names to denote various attributes (such as the source's subsystem, the category of data items it contains, or whether the source contains external kernel interfaces or subsystem interfaces); similar conventions on identifier names (indicating the defining subsystem, whether the identifier is a pointer, etc.); and always naming functions in the form verb-object (for instance, *dm_create_link* instead of *dm_link_create*). Such conventions are quite useful in reading and debugging code, especially in unfamiliar areas of the kernel.

5.2. Coding conventions

The DG/UX kernel has many conventions on the use of the C language to obtain consistency across a large number of developers and to avoid common errors. For example, the indentation and placement of braces and parentheses is spelled out so that it is the same in all kernel modules, regardless of who did the actual work. Other examples include: avoiding error-prone or non-portable C constructs (because of the confusion between "=" and "==", assignment within conditionals is prohibited, for example); declaring everything in the DG/UX kernel explicitly (even when allowing the C compiler to chose the default would produce the correct result); and not using the compiler built-in types such as *int* and *short* directly within the kernel because the realization of the types can vary on different machine architectures and because some of the types may produce inefficient code (instead kernel basic types are used that are defined in terms of the compiler built-in types).

5.3. Documentation

Each source file contains comments according to prescribed conventions. A module header gives the name of the module, lists the identifiers that are defined in the module, and gives a high-level English language description of the logical entity the module represents. It describes how the contents of the module are related and what the typical use is. The module header also contains a revision history in which changes to the module are noted.

Each definition, variable, or function in the module also has a header that describes the item. For type definitions, the header describes the purposes of the data structure, its possible relationships to other data structures, and similar details on each field if present. For constant definitions, the header describes the purpose of the value and how the particular constant was arrived at (e.g., mandated by a standards organization, determined by the hardware, or experimentally as the value that makes things work "well"). For variables, the header describes the range of legal values, when the variable is modified, and when it is initialized. For functions, the header describes what the function does, while being careful not to give details on how it does it. Function headers also state assumptions about locks that are held or other conditions that must be met before the function is called.

Within a function, comment blocks describe the implementation details. These comments constitute the detailed design of the function, which may be produced and reviewed before the code is actually written. They are not just an English language transliteration of code, but explain the overall algorithm and control flow used. They also point out subtle areas of the code and explain why the code is the way it is so that a future maintainer of the code will be careful when making changes.

6. Applying the Tools Theme

In addition to the standard C compiler, the tools used to create the DG/UX kernel include lint, a proprietary type checking program called ccheck, and a variety of shell scripts and makefiles. The particular use of these tools is summarized below.

6.1. Ccheck and lint

Ccheck is a proprietary utility that provides a stronger measure of type checking than lint or the C compiler. It is routinely used as part of the kernel compilation process in order to make the use of C conform to strongly typed rules. The overall reliability and enhanceability goals of the project make it important to flag type-mismatch errors early in the development process. Using special comments generated during the extraction of .d files, ccheck compares actual arguments against the corresponding formal arguments for type correctness. This checking is similar to the checking that is possible using function prototypes as specified by ANSI C, and could be done by the compiler when ANSI compilers are widely available. Ccheck also checks for missing declarations, mixing of pointer types and mixing of signed and unsigned arithmetic operands. Lint is used as well because of the heuristics checks it makes for unused variables, variables that are set but not used, or variables that are used without being set.

6.2. Build mechanism

The key feature of the build mechanism is that it detects what source files have changed and recompiles only those portions of the kernel that are affected by the changed source file. It consists of a collection of shell scripts, make files, and utility programs that are run each night to produce object files that reflect the state of all changes made during the day.

The first phase of the build mechanism constructs the dependency tree of the source files in the form of a makefile. When a source file changes the list of files that it includes, sed and awk scripts that scan each source file looking for #include statements automatically detect the change in the dependency tree and do the appropriate recompilations.

The second phase builds the interface of each subsystem by extracting the interfaces of source modules that are part of the subsystem interface and concatenating them to form the subsystem interface include file. The newly created subsystem interface include file is compared with the previous version, if any. If both versions are identical, then the subsystem interface hasn't changed and the old subsystem interface include file, with its old modification time, is preserved.

The third phase compiles each subsystem. Within each subsystem, interfaces are extracted and compared with the old version to determine if the interface has changed, and include files may be touched (changing the modification time) to reflect that an include file it depends upon has changed. Code modules are compiled and the resulting object files are archived into a library that contains all the code from the subsystem. Though the work within a subsystem must be done according to the hierarchy within the subsystem, subsystems as a whole can be processed in any order during the third phase of the build. This fact accrues from the requirement that a subsystem cannot have dependencies on the internals of other subsystems; it can only depend on other subsystems' interface include file. Since these were all created in the second phase of the build, subsystems in the third phase can be done in any order. The build mechanism takes advantage of this by doing multiple subsystems in parallel. On a multiprocessor machine, this parallelism makes more effective use of the processors and allows the build to run faster.

The build mechanism also supports check-in and check-out operations on source code. Whole subsystems must be checked out, and only one developer may have a subsystem checked out at a time. Not being able to have different developers check out different source files in a subsystem has proven somewhat restrictive, though not unbearably so. The restriction simply reflects that more work is needed in the tools and is not inherent in the design. Source code revision control is provided using SCCS as part of the check-in and check-out shell scripts.

6.3. Source level debugger

Use of a source level debugger has been an important productivity enhancement in the development of the kernel. The debugger can be used on all parts of the kernel. It allows walking back the kernel stack of a particular process, displaying the value of local and global variables, and setting breakpoints at source line numbers. Structure variables and arrays are displayed showing the individual field names or indices along with its associated value.

The debugger also forms the basis for a crash analysis tool which operates exactly like the debugger, except that it accesses a main memory image dump file created after a system panic. All of the debugger features for walking back stacks, display variables, etc., are available except for setting and taking breakpoints. This method of examining crashes has proved extremely fruitful in finding and fixing bugs from remote sites.

7. Experience and Conclusions

The application of these engineering techniques to the DG/UX kernel began with the project in late 1984, and have continued in use through its first release and up to the present. On the whole the results have been very successful.

7.1. Design and implementation implications

The decomposition of the kernel into subsystems and enforcement of the distinction between interfaces and implementation has been an excellent framework for motivating good design. Issues in the design were forced into the open early when interfaces between subsystems didn't mesh properly and when there was difficultly in maintaining the interface/implementation paradigm. These issues forced the designers to re-examine the partitioning of work between subsystems, resulting in a better understanding of the interactions between different parts of the project and hence a better design.

The emphasis on information hiding and encapsulation made it possible during the development to replace whole subsystems with new versions. In particular, the Medium Term Scheduler (MTS) subsystem was initially coded without much sophistication because the full kernel was not operational and writing a good scheduler requires a working system on which experiments can be run and measurements made. The interfaces were designed to encapsulate the scheduling knowledge, however, such that it could be replaced later without much trouble. Much later, after the system was working with the rudimentary MTS, a new MTS was developed in parallel that was eventually substituted for the old one. The new MTS did require some additional information from the Virtual Memory (VM) subsystem, so some functions were added to the VM interface that were not used by the old MTS. After testing was completed, the new MTS was inserted without disturbing the rest of the system.

At another level, information hiding allowed the representation of an eventcounter† to be changed from 64 bits to 32 bits in order to increase performance. The modules defining the eventcounter type and implementing the operations on eventcounters were modified to reflect the change and inserted into the main code tree. After a night of recompilation everything worked fine — except for the one place in the code where an eventcounter had been cast as a 64 bit integer in order to perform a non-standard operation on it. This sort of cheating cropped up occasionally, though in most cases it was spotted during a code review and corrected before it caused a problem.

The emphasis on strong type checking provided two advantages. First, strong type checking forces designers to be sure they understand how the various parts of their subsystem fit together. If the types are not working out correctly, or many casts are required in the code, it may be an indication that the design needs to be rethought. Second, many simple coding errors were eliminated at compile time instead of at debug time. Errors such as passing an *int* instead of an *int* * as an actual argument are detected when ccheck is run, and can be corrected before going to the trouble to set up a debugging environment.

The subsystem organization also allowed partial kernels to be built and tested well before the full kernel was ready for active use. The first test contained just a few of the lowest level subsystems and tested the ability of the Virtual Processor (VP) subsystem to swap virtual processors among the pool of physical processors. This test was run on dual-processor hardware long before most of the rest of the kernel was ready. Later, the Medium Term Scheduler (MTS) test checked out a larger portion of the system. Also, a Logical Disk Manager (LDM) test was built that just tested the ability of the system to read and write disks,

† An eventcounter is a data structure used for synchronization in the DG/UX kernel. The details are not relevant here, but see [Kel89a].

including the process multiplexing required to take and handle interrupts.

7.2. Performance implications

Two kinds of performance implications occur because of the emphasis on good software engineering. First, a significant amount of production machine time is required to perform the extract, compilation, ccheck, and kernel build phases. The large number of separate modules in the kernel and the include file hierarchy increase the compile time. These production machines resources, however, are being used to find bugs that would otherwise be found on standalone machines during kernel testing. A conscious decision was made to obtain the necessary production machine resources because bugs were found earlier, reducing the level of standalone machine resources and debug time required.

The check in and check out procedures associated with the subsystem paradigm are still too slow and cumbersome to be completely satisfactory. The procedures were developed on an ad hoc basis, and could benefit from some serious work by someone experienced in producing good tools. The slowness problems may be largely resolved by the introduction of the next generation of faster processors.

The second and more important performance implication is in the runtime performance of the kernel. Use of information hiding and encapsulation is often criticized because it generally introduces a larger number of function calls in a given code path than would be there if code paths directly modified all data structures. For example, it is not uncommon for a system call to produce 30 frames on the kernel stack at its deepest point. In most cases, however, the DG/UX kernel is willing to pay the additional cost of the function calls either because the additional cost is small relative to the total cost of the system call, or because the code path that incurs a significant penalty is rare during typical operation of a system. Hence the overhead of function calls usually has a negligible effect on overall system performance while providing significant benefits in the maintainability and enhanceability of the software. In the few cases where function call overhead is significant, code has been changed in ways that might be less than ideal from a software engineering standpoint (by coding in assembler, for example) in order to meet performance goals. In these cases, the software engineering has allowed the decision to be postponed until the latter stages of the project so that the important places could be fixed and premature optimization is avoided. Also, because processor performance is generally increasing at a faster rate than memory or I/O performance, the relative importance of processor overhead in making function calls is decreasing over time. The DG/UX kernel was designed with this trend in mind, and with the belief that trading a small amount of performance for increased reliability, maintainability, and extensibility is a worthwhile trade.

8. Acknowledgements

We would like to acknowledge the enthusiasm and professionalism of the entire DG/UX kernel development team in producing the DG/UX kernel. Special thanks is due to the original members of the group: Katie Algeo, Philip Christopher, Bob Goudreau, Jerry Pendergrass, Jeff Kimmel, Earle MacHardy, Joe Pittman, Steve Stukenborg, Tom Wood, and Hilary Zaloom. We would also like to thank and acknowledge our management for providing critical support: Dennis Balch, Jim Hebert, and Lee Schiller.

Trademarks

References

[Bac84a] M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Operating Systems," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1733–1749, October 1984.

[Ham88a] Graham Hamilton and Daniel S. Conde, "An Experimental Symmetric Multiprocessor Ultrix Kernel," *Proceedings of the Winter 1988 USENIX Conference.ds [I The USENIX Association*, pp. 283–290, Berkeley, CA, 1988.

[Hen84a] Sallie Henry and Dennis Kafura, "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics," *Software — Practice and Experience*, vol. 14, no. 6, pp. 561–573, June 1984.

[Hol83a] R. C. Holt, M. P. Mendel, and S. G. Perelgut, "TUNIS: A Portable, UNIX Compatible Kernel Written in Concurrent Euclid," *Proceedings of the Summer 1983 USENIX Conference*, pp. 61–74, The USENIX Association, Berkeley, CA, 1983.

[Kel89a] Michael H. Kelley, "Multiprocessor Aspects of the DG/UX Kernel," *Proceedings of the Winter 1989 USENIX Conference*, pp. 85–99, The USENIX Association, Berkeley, CA, 1989.

[Par72a] D. L. Parnas, "A Technique for Software Module Specification with Examples," *Communications of the ACM*, vol. 15, no. 5, pp. 330–336, May 1972.

[Par72b] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, December 1972.

An implementation of STREAMS for a Symmetric Multiprocessor UNIX Kernel

Philippe Bernadat

O.S.F. Research Institute
c/o Bull-Imag†
2 Avenue Vignate, Z.I. Mayencin
38610 Gieres
France
bernadat@gu.bull.fr

ABSTRACT

This paper describes a solution to running UNIX Streams in a multiprocessor environment. As for other layers of the UNIX kernel, every part of the streams code and every stream module can be executed on any CPU. The design is adaptable to various kernel implementations and only relies on the streams message mechanism.

1. Introduction

The Streams mechanism was designed and implemented for monoprocessor machines. Kernel programmers develop their own stream modules or drivers and then link them into the basic kernel. No assumption is made about the kernel implementation; the programmer only knows the interface which is described in "The Streams Programmer's Guide" [ATT87a]. As a consequence no multiprocessor synchronisation is added to modules and drivers. We describe an implementation of streams in the kernel where synchronisation is done outside the modules. The algorithm takes advantage of parallelism.

The kernel implementation is based on R3.1 & R2.4 versions from AT&T with additional functionalities such as the FFS (fast file system), socket interface from BSD 4.3 and SUN 3.2 NFS. This implementation of streams is adaptable to all symmetric multiprocessor kernels based on common memory architectures.

2. Machine Description

This multiprocessor kernel has been developed on a prototype model of the BULL DPX2000 family. This architecture includes local memory for each processor and also a common memory.

2.1. Global Bus

The hardware architecture is built on a proprietary global bus, to which up to 8 CPUs, 16 IOPs and Common Memory boards are connectable. The CPUs have local memories plugged on a local bus, invisible to the other CPUs. All the CPUs can address the IOPs and Common Memory boards on the global bus with the same priority. IOPs are slaves, which means they cannot access the global bus.

2.2. CPUs

All CPUs are identical, with the following components:

- MC68030 microprocessor.
- MC68881/2 floating point unit.
- 16 KB of physical cache memory.
- One asynchronous line.
- Timers.

† The author was employed within BULL S.A. (Echirolles, France) at the time the paper was written.

2.3. IOPs

The main types of IOPs are:

- SCSI board: connectable to disks, cartridge tapes, floppies and magnetic tapes.
- Ethernet board.
- Asynchronous lines boards.
- Synchronous board.
- Line printer board.
- Graphic stations board.
- GPIB/DR11 instrumentation boards.

Each IOP has its own microprocessor and memory.

2.4. Multiprocessor features

Each CPU can raise an interrupt to wake up all the CPUs. There is a flip-flop used in conjunction with the TAS instruction to prevent a cpu waiting for a resource from looping with the TAS instruction on the global bus.

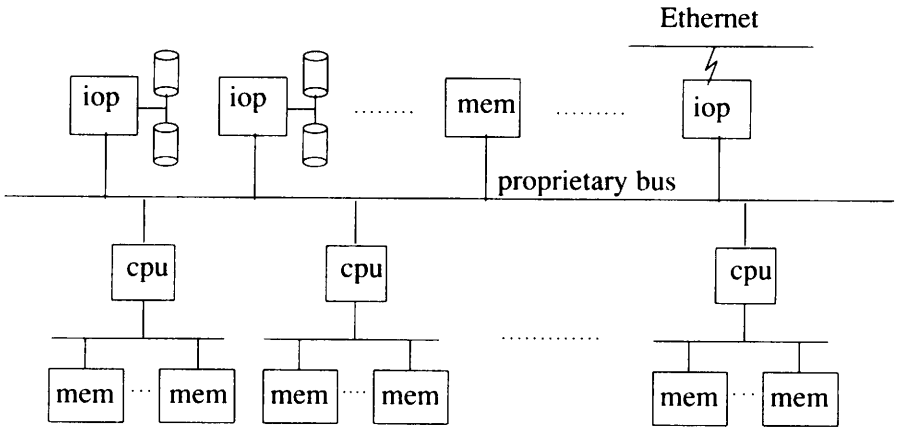


Figure 1: *DPX 2000 architecture description*

3. Kernel Description

The kernel is based on AT&T's R3.1 version, including FSS (File System Switch), Streams, shared libraries and System V IPC. The FSS is used to add both BSD 4.3 FFS (Fast File System) and SUN 3.2 NFS (Network File System). Sockets are implemented on top of the streams. The multi-processor adaptation is based on AT&T's R2.4 version, except for FSS, FFS, NFS and Streams which are not part of R2.4.

UNIX code is duplicated in each local memory, with most of the common kernel data loaded in common memory. The user segments and the *User area block* are loaded in local memory with shared memory segments loaded in common memory.

3.1. Multiprocessor protection

Multiprocessor protection is implemented with **semaphores** and **locks** [Bac84a]. Locks are used for "busy wait" and semaphores allow queuing of processes waiting for a resource or an event. A semaphore itself is protected with a lock. The following operations on semaphores are used:

P wait/grant operation on a semaphore.

V signal/free operation on a semaphore.

CP Conditional grant on a semaphore with no wait, a return status indicates if semaphore was free.

Depending on its initial value a semaphore is used in two different ways:

0 The semaphore is used for event synchronisation, for example waiting for a file system buffer.

1 The semaphore is used to ensure exclusive access to kernel text or data.

3.2. File Systems

We designed a synchronisation which allows foreign file systems to be added without modifying their code. An optional external semaphore is taken with a P operation when entering the FSS and released with a V operation when leaving it. This semaphore is also released each time a process inside this file system goes to sleep, or if calling another file system, or a driver. The first implementation of NFS has been tested using this external synchronisation. Later on, locks and semaphores were added to the NFS code (just as for FFS) to allow for more parallelism.

3.3. Drivers

Here the goal (just as it was for FSS) was to link-edit drivers without modifying their code with multiprocessor synchronisation. For each driver, an optional structure is used, containing the following items:

- A semaphore.
- A linked list of pending interrupts.
- A lock to protect this list.

This structure is pointed to by the *bdevsw* and/or *cdevsw* structures. When calling a driver through the *bdevsw* or *cdevsw* a P operation is done on the semaphore if it exists. This is transparent from the programmer's point of view. A V operation is done when coming back.

When an interrupt occurs, the semaphore state is checked, using a conditional CP operation. If it is free, just process the interrupt and then release the semaphore. If it is not free, the interrupt is acknowledged, linked on to the list but not processed. The process owning the semaphore will process this interrupt when releasing the semaphore. As a consequence, each time a process or an interrupt routine leaves a driver, it must run the pending interrupts linked inside the list.

When a process goes to sleep inside a driver, the semaphore must also be released and taken back when the process is awakened.

4. Streams Mechanism

A stream consists of connected modules, with data flows in both directions: from stream heads to drivers and vice versa. Each module consists of two queues, one in each direction. The communication between the modules is done exclusively by sending messages to the upper or lower module [Rit84a].

There is 5 types of interface between stream modules or between stream modules and other parts of the kernel:

1. **System call** interface with stream heads.
2. **"Put" procedures**, used to pass a message to the queue of the upper or lower module of a stream. The only parameters for this procedure are a message and a pointer to the next queue, there is no return code and this is the only interface used between two modules.
3. **Service routines** used to delay module processing. These routines are called once more time-sensitive kernel activities have been performed.
4. **Recovery from message block allocation**. If a message block allocation fails, a module can ask to be called with a private function as soon as new message blocks are available.
5. Hardware **interrupts**.

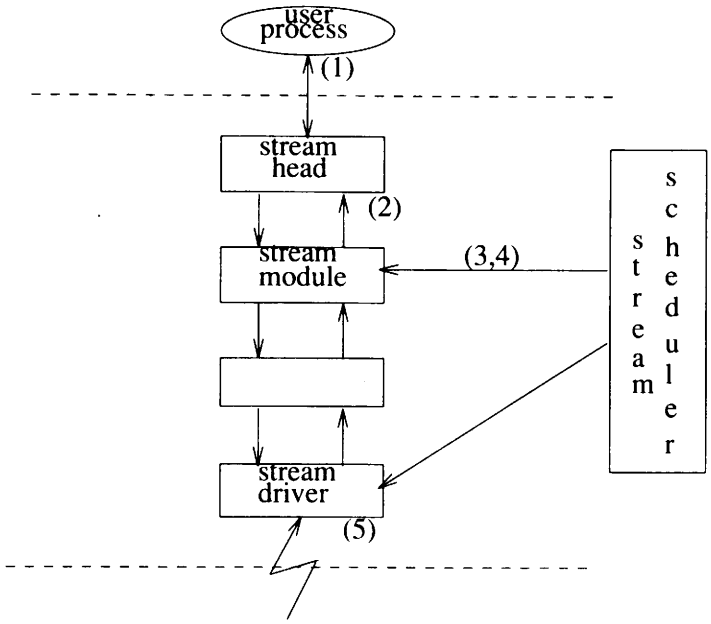


Figure 2: *Stream basic architecture*

5. Multiprocessor Streams

Besides the stream modules, other parts of the stream code are protected with the usual semaphores and locks. This includes:

- Message block allocation.
- Queue allocation.
- Stream head allocation.
- Stream scheduler.
- Clone driver.

When accessed from above, a stream head is considered as a regular module, so that it may be protected in the same way as any other module.

5.1. Module protection

To maintain consistency without modifying the module's code, at most one processor at a time is allowed to run a module. The goal is to protect the module code in an efficient way: "busy wait" must be avoided before entering modules, and taking advantage of parallelism would enhance performance.

For each module there is defined an optional structure containing the following items:

- A semaphore.
- A linked list of messages
- A linked list of interrupts.
- A lock to protect the two lists.

5.1.1. System calls

When using streams via system calls, a process enters a stream head module. The stream head module is locked using a P operation on the module semaphore. It is released with a V operation when returning from the system call. If the process goes to sleep, the semaphore is released, and taken back when awakened.

5.1.2. Put procedures

Put procedures are called in both directions: this means that the kernel may be in interrupt mode and that it cannot use P operations (otherwise the system could sleep whilst in interrupt state). Using busy waits is not performed. So the conditional CP operation is used and 3 situations may occur:

1. The module of the next queue is free. Just process the put procedure, without releasing the previous module semaphore. When coming back, this called module is released, and the previous one is not "relocked" as it was not released.
2. The next module is not free but the current task already owns it. The put routine is processed without locking or releasing any module. (A task stands here for a process in kernel mode or a cpu handling an interrupt.)
3. The next module is not free and is not owned by the current task. The message is added to the linked list of messages defined above. The queue identification is also recorded in the message: indeed there are two queues per module and a module may be pushed on more than one stream at a time. From the caller's point of view, this linking of the message is completely invisible, there is no synchronisation when calling "put" procedures, the caller ignores what will be performed on the next queue, and potential replies are sent using a "put" routine (qreply) in the reverse direction.

This solution implies that when releasing a module, the linked list of messages must be scanned to perform the put operations on the related queues.

If the notion of owner of a semaphore does not exist, or if it is not possible to know the owner, case 2 and 3 are merged: if a module is locked by the current task, the message is linked and will be processed when the task itself releases it. It is just a matter of performance.

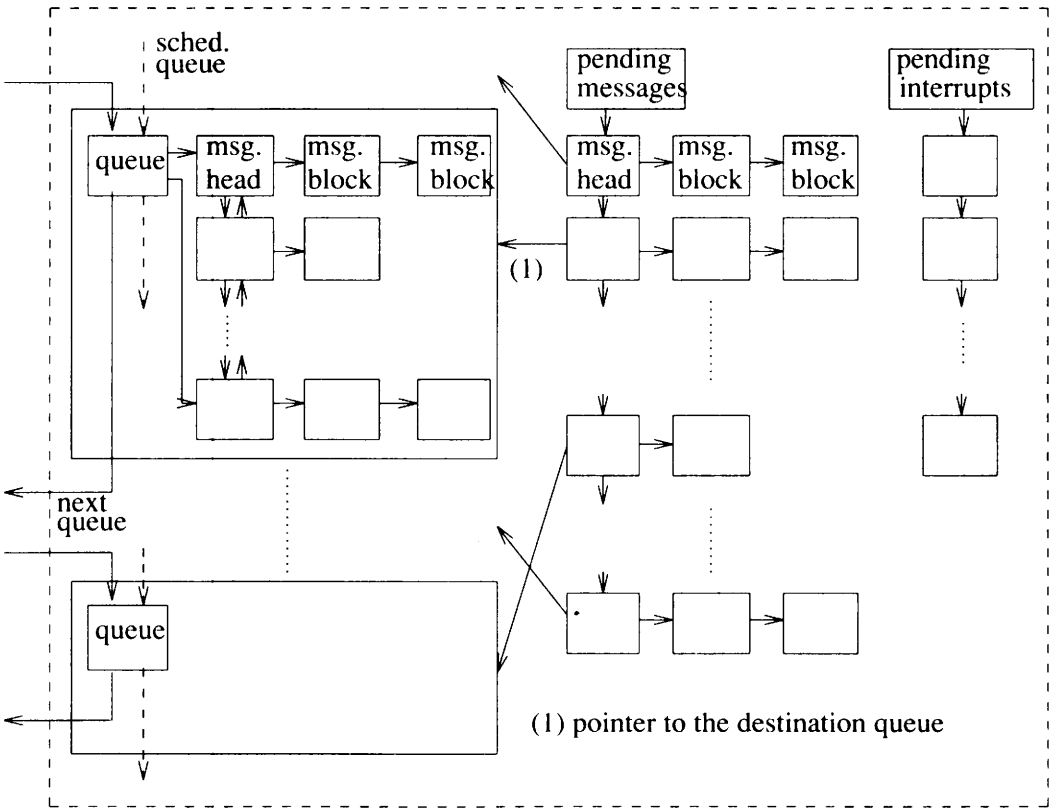


Figure 3: *Module organization*

5.1.3. Service routines and recovery from allocation failure

These two interfaces are called by the stream scheduler. The stream scheduler is activated when returning from interrupt mode to user process and before the system enters the idle state and in fact once more time sensitive activities are performed. The scheduler is not a process, it has no context and cannot sleep. When activated, it scans several lists:

- A list of queues to activate via the service routines.
- For each message class, a list of modules waiting for message blocks. If message blocks are free, a specific function of the module is called.

For every member of each list, the scheduler tries to lock the corresponding module semaphore with a CP operation; if it is free, just perform the work. Otherwise, the queue or module activation is delayed until the next scheduler activation. All the CPUs can activate the scheduler, and more than one CPU at a time can run the scheduler. The lists of queues to activate are protected from concurrent accesses but are shared. This allows parallelism.

5.1.4. Interrupts

Interrupts are performed exactly as they are for a regular driver, using a linked list of pending interrupts in case the module is not free. Additionally the interrupt task must also process pending messages before releasing the module.

5.1.5. Linked list processing

The above algorithms imply that before releasing a module's semaphore, a task (process or interrupt task) must perform the following operations:

1. Process the pending interrupts.
2. Call the put procedures for the pending messages.

5.1.6. Message priorities

To keep a priority order between data messages and control messages, the control messages can be linked at the head of the linked list and data messages at the tail. Pending interrupts are always processed before pending messages.

5.1.7. Module functional organization

At init time a module contains text and private data. When pushed on a stream, queues are allocated to the module. Messages are then sent by the upper and lower queues of other modules. Additional needs for the multiprocessor synchronisation are one linked list of pending messages and one linked list of interrupts (Figure 3). A semaphore controls the access to the module and a lock protects the two linked lists.

5.2. Parallelism

Figure 4 illustrates the type of parallelism that can happen for a single stream with this implementation. The different modules of the stream can be run by different CPUs at the same time.

There is also parallelism between different streams as long as they are not processing the same module, however, there is no parallelism between different streams inside the same module.

6. Application to sockets and NFS

NFS, and TCP/IP applications (remote commands, file transfer protocols ...) are implemented using sockets. Sockets are themselves implemented over streams. The interface between a user process and the kernel is the socket's one. The socket layer consists of specific stream heads invisible from the user's side, accessed only by the kernel.

Figure 5 illustrates this stream implementation: each solid line boxes represents either a stream head, a simple module, a multiplexer or a driver module. Each one is protected with an external semaphore.

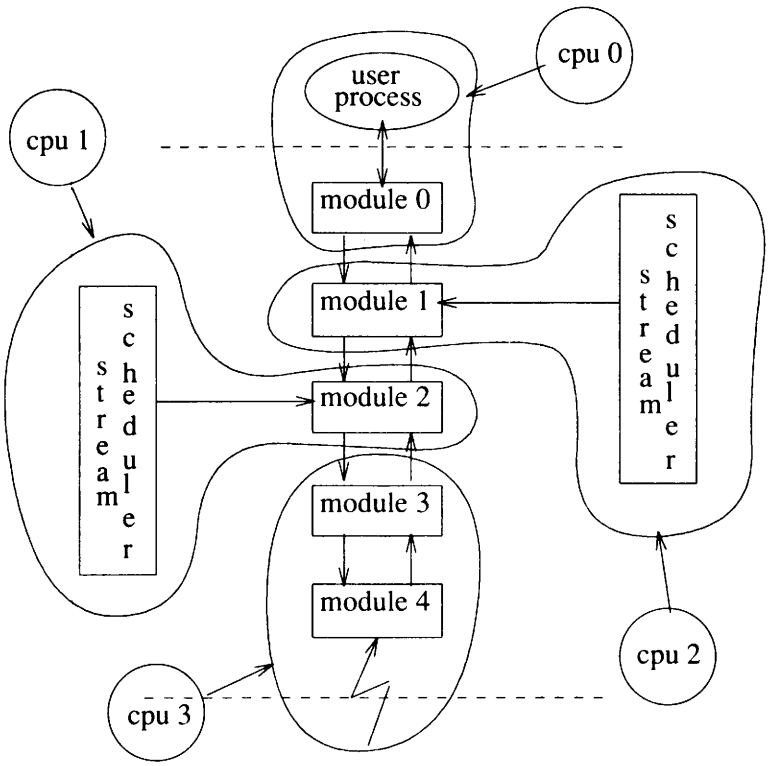


Figure 4: *Multiprocessor stream execution example*

7. Performance

This algorithm has been developed on a prototype machine. When running with a single processor, only local memory is used, whereas when running with more than one CPU, most of the system tables are in global memory which is much slower than local memory. Indeed, global memory is organized with 16-bit words and is not cached.

However, to measure the impact of parallelism, the prototype is sufficient. We have done measurements with two types of commands:

rcp running over TCP/IP.

cp across **NFS** running over UDP/IP.

First each of the two commands is executed from 1 to 4 times in parallel on a single CPU. Then it is executed from 1 to 4 times in parallel on 4 CPUs. The same transfer is done with each of the two commands.

Most of the elapsed time is system time inside the stream's code or idle time. When running with only 1 command, the system idles 30% of the time, which means that the global throughput is limited by the Ethernet board throughput and cannot be increased with additional CPUs. Looking at the figures for 1 command the impact of parallelism for TCP can be observed. System time on the CPU where the command runs is 13.34 for 1 cpu and 5.40 for 4 cpus. **60%** of the system time (i.e. stream procedures) is achieved by other cpus. TCP intensively uses service procedures whenever possible as recommended in the streams programmers guide, whereas UDP does not.

Better performance could probably be obtained if some modules were rewritten in a multi-processor environment. In our example, the IP layer is a multiplexed module and might be a bottleneck.

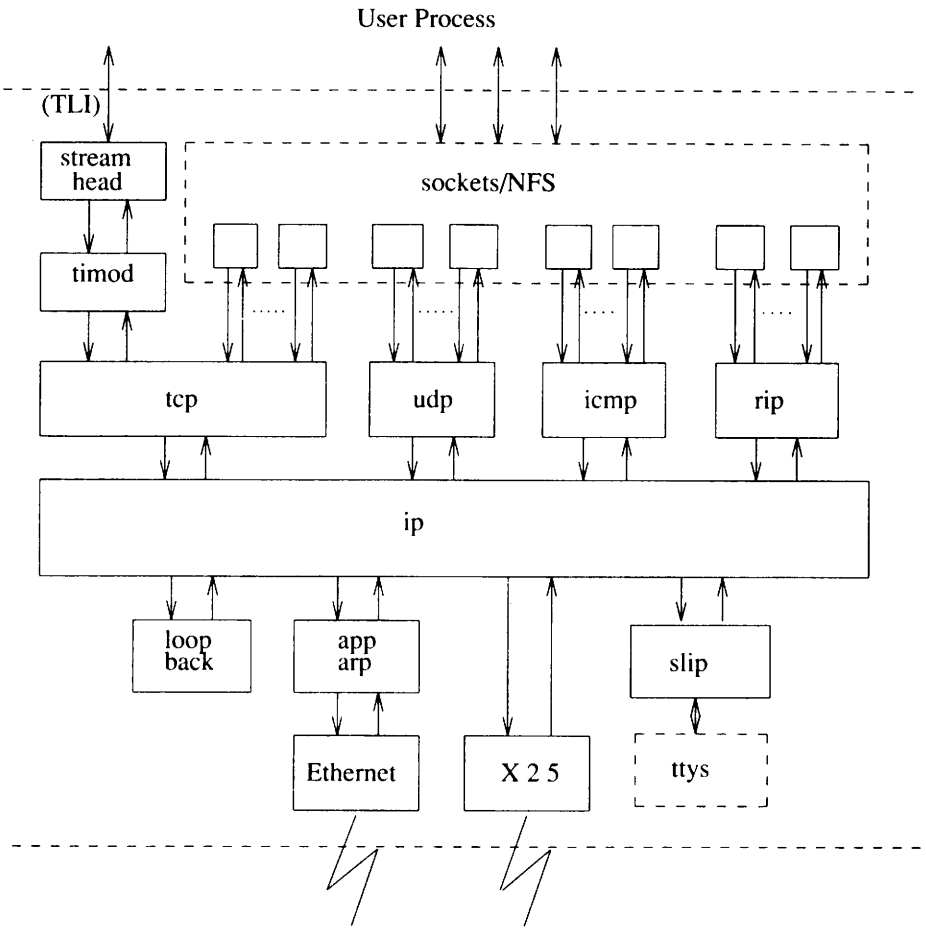


Figure 5: *Network's stream implementation*

8. Conclusion

The algorithm described here is quite natural, it relies on the Streams basic idea: a unique way to communicate between layers (modules) with messages. Besides the advantage of not having to change module's code, performance is enhanced if procedure services are widely used. Future enhancements concern a larger parallelism within multiplexed modules.

9. Acknowledgments

I would like to thank the following people who worked with me on this implementation: Eddine Walehiane who introduced me to the streams, Philippe Durieux who helped me in designing and porting algorithms for the sockets and Jean Pierre Joannin for his useful comments.

References

[ATT87a] AT&T, *UNIX System V Release 3 STREAMS Programmer's Guide*, 1987.

[Bac84a] M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Operating Systems," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, October 1984.

[Rit84a] D. M. Ritchie, "A stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, October 1984.

	Command				
	rcp		**cp across NFS**		
Number of commands run in parallel	1 CPU	4 CPUs	1 CPU	4 CPUs	time (seconds)
1	18.22 0.14 13.34	15.92 0.14 5.40	11.86 0.08 2.60	10.00 0.10 2.58	real user system
2	30.38 0.30 25.90	21.38 0.34 24.14	22.62 0.12 4.16	19.58 0.10 5.86	real user system
3	43.86 0.42 38.70	32.84 0.44 33.82	34.14 0.12 6.00	28.86 0.12 8.80	real user system
4	56.96 0.56 51.02	38.64 0.58 55.20	46.06 0.18 7.72	37.76 0.22 11.68	real user system

Figure 6: *Compared performances of mono- and multi-processor execution*

Developing Writing Tools for UNIX Workstations

Martin D. Beer, Steven M. George and Roy Rada

Department of Computer Science,
University of Liverpool,
P. O. Box 147,
LIVERPOOL
L69 3BX.
UK
mdb@mva.cs.liv.ac.uk

ABSTRACT

The availability of networks of UNIX-based graphical workstations has stimulated new developments in authoring software. This paper presents our experiences along several fronts. First, we discuss the lessons learnt from developing a simple authoring tool to run on the Atari-ST, using the GEM operating system. This was always intended to be used by a single author and was not tied to expensive computer networks. With the arrival of a large network of powerful graphical workstations in our department, developments have recently transferred to them. We discuss the development of software using 1) the X11 toolkit and one of the readily available widget sets, 2) a configurable editor (GNU Emacs) to develop prototype applications, and 3) the Andrew toolkit to re-implement the original Atari authoring system, but this time providing a tool that will allow several authors to collaborate closely with each other. The practicalities of these approaches are discussed with reference to our own experiences.

1. Introduction

The recent development of high-powered, relatively cheap, graphical workstations has opened many opportunities for investigating tools to assist in the writing process. Since these workstations are based on a full, multi-user operating system (UNIX), which support not only the execution of many processes on different network clients, but also a single integrated, network-wide filestore, it is much easier to share information between users than on conventional, PC-based networks of word-processors. This has led to the investigation of existing collaborative writing tools, and their methodologies, in the same framework.

Before the advent of more sophisticated equipment in our department, attempts to develop writing tools that provide a comfortable user-interface were based on the Atari-ST microcomputer. These generated user interest, but were limited in the functionality provided as all the features had to be developed from scratch. A number of valuable lessons were however learnt at this stage, and these will be discussed next.

2. The Atari Experience

A usable prototype authoring system, called ATARI-ST-TROFF, was developed on the Atari, which provided an acceptable interface for the user who did not wish to learn the complexities of the standard UNIX documentation tools:

- the *troff* phototypesetting program,
- the *ms* macro set,
- the *tbl* table preprocessor, and
- the *eqn* equation preprocessor [Ker84a].

Both Academic staff and research students spend a considerable proportion of their time preparing documents. It was therefore relatively easy to assess the main requirements of the intended system. These included:

- avoiding the need to edit and preview documents on the larger UNIX machines, which was often not convenient, particularly when members of staff needed to work at home,
- not having to learn another collection of commands. Discussions with a number of active users indicated particularly difficult tasks, such as:
  - remembering the order in which certain macros should appear, particularly in the document heading (i.e. the order of .RP, .TL, .AU, .AI etc.), and
  - ensuring that bracketing macros matched (ensuring that for example there was a .KE for every .KS).
- obtaining draft output whenever required.
- editing documents in their final form on-screen. People did not like modifying text files that bear no relation to the final output.

Further investigation showed that proofs are used mainly for correcting the content of documents and their overall look, rather than to sort out particular layout problems. It is therefore unnecessary for proofs on dot matrix printers and the basic editing screen to show exactly what will appear on the final printed page, so long as the essential features are present. In the pilot program for instance, no attempt was made to place page boundaries on the editing screen, since these may appear in different positions in the final document. This corresponds closely with the intermediate editing representation used by GRIF [Fur88a].

ATARI-ST-TROFF allowed the user to enter titles and section headings by means of dialog boxes which control the levels. Information which does not vary often, such as the author's name and address, was collected from a simple database, so that it did not have to be entered for every paper. Once headings had been defined, text was entered from the keyboard, in the same way as with a normal word processor. Paragraph starts were selected from pull-down menus.

Analysis of intended usage showed the need for a system that was portable in that drafts for documents could be downloaded onto a floppy disk that was compatible with a home computer. The document was saved to disk as a straightforward text file that, after transferring to the server by means of the Kermit [Cru84a] file transfer protocol, could be entered directly into the phototypesetting program. Facilities are also provided for outlining and dynamic reorganisation of the document on the workstation screen.

Our ATARI-ST-TROFF system has been used to good effect by several of our colleagues. However, major deficiencies were soon identified relative to the following needs:

- to configure functionality, so that the editor could be easily tailored to individuals' requirements and writing styles,
- to manage large bodies of text (typically books and large reports consisting of hundreds of pages in several chapters),
- to collect text from several sources,
- to mark-up and index, which users considered to be a chore that could be considerably eased by partial automation.

It was also found that the requirement for a transportable text entry system (hence the use of the Atari) was not as important as originally envisaged, as a much higher proportion of the actual typing was performed in the department, than was originally expected. This was fortunate because it was clear that the additional functionality that our colleagues were requesting, could not be accommodated within the Atari, which was already fully stretched running the initial prototype.

3. Converting to a Workstation Environment

The delivery of a large departmental network of Hewlett-Packard 300 series workstations allowed a new approach. Every member of staff and postgraduate student has ready access to a workstation at their place of work. The adoption of the X-Windows user interface has allowed much of the ATARI-ST-TROFF design work to be continued with little modification, particularly as the Mouse-Window-Menu structure could be retained. In fact, the conversion has been highly advantageous in several respects, as limitations imposed artificially by GEM on the Atari's have been removed.

The conversion to a system with a large shared filing system, as is the case with our workstation network, has allowed the text storage and retrieval mechanisms to be improved considerably. This supports collaborative writing and elementary version control. Since the basic editor does not now provide a fully transportable working environment, tools have been provided that allow documents generated on other systems to be entered into the system. It was also necessary to develop data structures to manipulate large

items of text as collections of independent objects, that could be linked into a coherent whole. This led to the adoption of a hypertext model.

To try the various strategies as quickly as possible and to allow user reactions to be determined at an early stage, rapid prototyping tools have been used. This paper will discuss the advantages and disadvantages of three rapid development methods:

- expanding the functionality of a powerful text editor (GNU Emacs) by writing programs in its own programming language (Mock Lisp)
- developing programs based on the X-Toolkit widget libraries (principly the HP widget set, with the user interface built around the text editing widget)
- making use of specialised toolkits (in our case the Andrew toolkit for building multimedia editors) [Mor86a].

All three approaches have allowed us to investigate methodologies for developing writing tools in the workstation environment.

3.1. Using Widget Libraries

The first requirement was to develop a program which would provide equivalent facilities to those available using the Atari program. This required the development environment to be as close as possible to that available under GEM on the Atari, (or on the Apple Macintosh, which was the other windowing system which had been used extensively for development work within the department). The version of X Windows that was delivered with the workstations was only X10.4, and proved to be a major disappointment, as the programming facilities provided by the *Xlib* library are at a very low level. Fortunately we were soon able to obtain a copy of MIT Release 3 of X11. This proved much more satisfactory, as it provided us with not only the *XtLib* toolkit library, but also a variety of widget sets as well.

After some investigation, it was decided to try to implement the document editor, using the HP widget set. This choice was made primarily because full documentation was available [HPW88a]. This proved remarkably easy, so long as the functions required were available as widgets. The code proved to be much more compact than the Modula-2 code for the Atari. (It is not possible to give reliable statistics at this stage as the Atari version includes considerably more functionality.) Considering the traumas of installing and working on a totally new system, implementation work started very swiftly.

It was decided to base the main editing functions on the *Text Editing* widget, which provided much of the front-end functionality that was required. This widget is extremely complicated, and proved to be the most difficult part of the system to program. As provided, it is really only suitable for handling small amounts of text. It does however, allow the programmer to replace its editing functions with his own. This was essential in our case anyway, as we did not want to store the document in exactly the same form that it appeared on the screen. So far the program has been developed with the default editing facilities, so that the basic user interface could be defined. This proved relatively easy, except in the case of the menus. The documentation describes the use of pull-down menus in some detail, but only widgets for pop-up menus are provided. It took much experimentation to redesign the menu tree so that it would look right as pop-up menus.

3.2. Expanding on Emacs

On delivery of Hewlett Packard workstations, several new editing tools soon became available. At an early stage it became apparent that one of the editors, GNU Emacs, could be customised and extended with relative ease.

3.2.1. Characteristics of Emacs

GNU Emacs is a freely available member of the Emacs [Sta81a] editor family. It has a large set of primitive functions and a programming language called Mock Lisp as an integral part of the release. Mock Lisp contains many of the functions of the Lisp language. The language is particularly strong in functions (including all those of the editor itself) which may easily be used for the development of document processing programs. In the best traditions of Lisp programming, one can build new, more complex, routines from those already existing. These new functions become part of the language by loading them at the start of each session.

Emacs also has a program running environment. The program can be edited within Emacs itself, and when complete and having been saved to a normal text file, it can then be *loaded*, which comprises its compilation before being successfully incorporated as an extension to Mock Lisp. Once successful loading has taken place, the function is available either for use in new programs, or from an interactive mode, in which the user can type in Mock Lisp expressions and see the results immediately on screen.

GNU Emacs offers a multiple window interface to the user, giving the ability to look either at different documents within buffers of the editor, or the user can see multiple views on the same document (or even a mixture of both). On the Hewlett Packard workstations, the Emacs editor can run under the X Windows interface, and thus several, simultaneous copies of the Emacs editor can be used on one workstation concurrently. Each of these incantations can have access to the same files, and the editor will provide elementary messages to bring the users attention to the fact that certain versions of a document may have changed during the current working session. Although this may seem obvious when the different copies of Emacs are on the same workstation, this principle is very important when considering collaborative work where the simultaneous copies of the editor are on distributed workstations.

3.2.2. Authoring Method

Initially a structure for representing documents within the prototype was agreed. This consisted of mainly three parts the first of which was a directory (called the paragraph directory) which contained many small files each of which represented a basic building block (either paragraphs or figures) of the document. Since we were looking at this prototype from the point of view of a Hypertext approach, we were investigating the methodology of splitting texts into smaller units and creating documents from a concatenation of these, controlled by a traversal of the second part of the prototype, the semantic net.

The semantic net was also stored in the form of many small files each of which was called a frame. Each frame represented the information about a single node in the semantic net, showing the name of the node, the links to other nodes in the semantic net and also which paragraphs should be associated with the node. The semantic net was built up in parallel with the collection of paragraphs, and by traversing this network we hoped to produce a coherent document representing the authors original intentions.

Now that we had a prototypical document representation and authoring method, we looked at traversal techniques for the semantic net and tried to compare the document created by this method with that originally in the mind of the author. We looked at different orderings of the paragraphs produced by different traversal algorithms, trying to assess the most useful and make improvements where necessary. The different traversals have also given rise to an investigation into browsing a semantic net in the form that we have developed, allowing users to move around the document via the semantic net and in the case of an author, be able to edit the document using this navigation method.

It can be seen that even with this crude, early attempt at an authoring system, there is favourable scope for collaborative authoring. However, we see that there are several problems which need to be overcome, such as version management for the semantic net files and the paragraph files. We also need to support communication among collaborating authors. This has led us to look at discussion techniques and consider how comments could be incorporated into the system.

Although easy to customise (given knowledge of Lisp) and very powerful, Emacs does have some serious drawbacks. It is very slow indeed. The production of a 60-page, linearised document could take to 20 minutes. We certainly do not have a user friendly interface as the production of a document requires knowledge of the Emacs editor, which is not easy to learn. The structure under which the document is stored is unsophisticated and in the future would be replaced by a relational database. Speed has been increased by using the C programming language in conjunction with Emacs.

In conclusion then, although Emacs has provided a quick and easy route to implementing a rough and ready prototypical authoring method, we see that for an advanced system, we would have to look towards other methods of programming.

3.3. Using Andrew

Another approach to re-implementing the Atari document processor was to use a specialised document editor toolkit. The most widely distributed example of this is the Andrew toolkit†. This toolkit was designed so that a multi-media editor could be written which allows text, tables, drawings, animations and

† We originally obtained a copy on the X11r3 release tape, but later obtained a version customised for the HP 300 series workstations from Jean Gascon of HPlabs, Palo Alto, to whom we are most grateful

so on to be combined in a single document. This has been achieved very successfully, and a multi-media editor, an associated mail system and an extremely comprehensive help system are provided as example applications. In many cases sites will mount the example Andrew programs on their own system and use them without modification. This is not our intention, as we wish to develop our document model.

Further study of the example document editor (*EZ*) showed that it could be used as a basis for developing our work:

- Documents are held in a special internal format, which is converted into raw *ditroff* before printing (a previewer is available, which displays the text on the screen in exactly the same form as it is printed). This internal format could be modified very easily. We would need to write a new document translation program, which would generate text to be processed either with *LaTeX*, or with *troff* plus the *ms* macros.
- There was a need to allow documents to be made from the contents of several files (in the same way as the document browser implemented in *Emacs*, described in the last section). This would require modification of the present editor to handle the file requirements.
- Andrew allows document formatting to be modified by *Style Sheets*, which are held as part of the document. This means that it is difficult to change the form of a document, once it has been entered. The advantage is that the style sheet is always available, anywhere on the network, including when messages are sent by electronic mail [Joh88a]. We would like to be able to determine the usage of the document, and then select the appropriate Style Sheet from a database.

This is a formidable list of objectives, and obviously could not be tackled at once. One advantage is that there is no immediate need for the picture drawing capabilities of the multi-media editor, as the software necessary for printing them is not currently available at Liverpool. This simplifies the document translator considerably. It was therefore decided to tackle the printer translator first.

The internal representation of different text objects within Andrew, is a bracketed structure, that allows these objects to be nested. This is a somewhat simpler structure than our own representation, which is based on the *ms* macro calls. This had been adopted originally to avoid the need for complicated translation programs, which could pose considerable difficulties on small computers, such as the Atari. Since Andrew required this translation anyway, it was decided to use the Andrew representation.

Menus similar to the Atari program were defined, and included within *EZ*. The only problem that has so far been identified, is that it is impossible to automatically number sections (those introduced by the .NH macro) when displayed on the screen. This would require a considerable reorganisation of the editor, and has been avoided for the moment.

Andrew makes use of an object-oriented class system, similar in many respects to C++. All applications are written in this way. This is a particularly easy environment in which to work, and progress has been quite rapid so far. Simple objects are defined from scratch, and then combined into more complex objects. These can in turn be used on their own, or combined with further complex objects to form even more complex applications.

We have been fortunate in being able to follow the form of the usual document translation program, *ezprint*. This is supplied as part of the standard Andrew release. Our program, *msprint*, follows the form of this program very closely. In practice, our program is considerably simpler, as in most cases all that is necessary, is for the *ms* macros to be added before and after the text.

4. Conclusions

So far, all but one of the systems described here have assumed what is essentially a hierarchical document model. Work elsewhere in the department [Ben89a], shows that this is in many respects inadequate, and proposes a model based on directed graphs. This approach has been taken further where a generalised model is developed that can support multi-threaded [Bar89a] documents. This approach shows considerable promise in supporting our current research into support tools for collaborative authoring.

Problems are caused by the desires of users to make use of the facilities offered at the earliest possible stage. They then pressurise the developer to make early versions of their software available, which may be incomplete or inadequately tested. This either leads to dissatisfaction with the system, which then goes out of use, or users make copies of development versions that were never intended for general distribution, and problems with these come back to the development team long after they have been solved in the main development process.

Particular difficulties are caused by the nature of the distributed filestore, which requires many systems files to be copied to each file server. This problem has been avoided to a great extent by linking local directories to a single support directory, on a user disk. It is then possible to apparently update the software on all workstations simultaneously. This avoids problem caused by forgetting to update individual servers, at the same time as the rest. Once installed, these links have proved extremely successful, not only for the above reasons, but also because any software installation can be generally performed from within the development username, and not by *root*.

Care needs to be taken that released software has only the execution traps set in favour of the users. On the rare occasions that this has not been done, copies of that version have multiplied extremely rapidly, as users took copies 'just in case'. Unfortunately these were usually copies of extremely early releases of programs that were soon superseded by significantly better versions. When problems that we had thought were long sorted out reappeared, we soon learnt our lesson!

It is now clear that our original aim of providing a single support tool, which could be configured to accept individual document models, be they:

- reports,
- books,
- documentation,
- letters, memos, or
- source code

in such a way that any item of data need only be entered once, is too ambitious for current methodologies, at least within a development timeframe that we can handle. There is still the need to investigate the means by which information can be managed away from the main network.

This includes both:

- version control as the document is developed separately on two different systems, and
- authorisation control when the new text is made available again to the main system.

We have proposals in both these areas [Bee89a].

References

[Mor86a] J. H. Morris *et al.*, "Andrew: A Distributed Personal Computing Environment," *CACM*, vol. 29, pp. 184–201, 1986.

[HPW88a] Hewlett-Packard Company, *Programming With the HP X Widgets*, 1988. HP Part Number 5959-6155

[Bar89a] J. Barlow, M. Beer, T. Bench-Capon, D. Diaper, P. E. S. Dunne, and R. Rada, *Expertext: Hypertext-Expert System Theory, Synergy and Potential Applications*, London, England, September 1989. To be presented at Expert Systems '89

[Bee89a] M. D. Beer, *The Use of an Office Information Server in an Environment based on High-Performance Workstations*, Cologne, West Germany, September 1989. To be presented at EUROMICRO '89

[Ben89a] T. J. M. Bench-Capon and P. E. Dunne, "Consistent Graph Modification Systems for Classes of Electronic Document," Report CS/CSCW/1/89, Computer Science Department, University of Liverpool, Liverpool, 1989.

[Cru84a] F. da Cruz and W. Catchings, "Kermit: A File Transfer Protocol for Universities," *Byte*, June, July 1984.

[Fur88a] R. Furuta, V. Quint, and J. Andre, "Interactively Editing Structured Documents," *Electronic Publishing*, vol. 1, pp. 19–44, 1988.

[Joh88a] J. Johnson and R. J. Beach, "Styles in Document Editing Systems," *IEEE Computer*, pp. 32–43, January 1988.

[Ker84a] B. W. Kernighan, "The UNIX Document Preparation Tools – A Retrospective," *Conference Proceedings of PROTEXT I*, Dublin, 1984.

[Sta81a] R. M. Stallman, "EMACS: The extensible, customisable self-documenting display editor," *Proceedings ACM SIGPLAN/SIGOA Conference on Text Manipulation*, pp. 147–156, 1981.

Implementation of a Window Manager under X11R3

Hans-Joachim Brede
Nicolai Josuttis
Achim Lörke

BREDEX GmbH
D-3300 Braunschweig
nico@bredex.uucp

ABSTRACT

A window manager in the X11 environment is responsible for the appearance of all X applications. It does the placement (moving, resizing, stacking etc.) of windows and can add some "decoration" (titlebars) to the top-level X windows. Additionally, it will enforce some policy on iconifying windows.

After a brief overview over the duty of a window manager and the responsibility of conforming clients, we will describe the design and implementation of a specific window manager. The implementation follows the guidelines given in the "Inter-Client Communication Conventions Manual" [ICCCM].

Special effort will be given to some of the less trivial aspects of a window manager. This includes discussion of colormap policy, icon concepts and some special X requests (synthetic events, save-set handling and additional properties).

The whole window manager is part of a bigger system that emulates an existing window system for SINIX† called COLLAGE‡ This system was realized using X toolkit widgets, and therefore the window manager is integrated in the toolkit environment. Problems that arose from this context are those with the reparenting of windows and the handling of events regarding the root window.

1. Introduction

A project we haved worked on was to port an existing window system for SINIX called COLLAGE into the X environment. COLLAGE is a proprietary system that runs on a variety of Siemens hardware. It includes an application monitor (APM), a build in window manager and an application programming interface (API) called Window Access Method (WAM). The goal of our XCOLLAGE project was to build an environment that only requires relinking of already existing applications to make them work with our implementation and let them coincident with other X Window applications. This leads to an architecture with an XCOLLAGE server interpreting WAM messages, translating them into X calls and vice versa. An environment variable, similar to the DISPLAY variable in X, tells a COLLAGE application were to look for its server.

XCOLLAGE is carried out by using the X toolkit with new widgets. For more details on XCOLLAGE refer to [Lor89a].

In this paper we want to concentrate on the window manager part. COLLAGE has a build in window manager that provides window decoration like titlebars etc. This means all windows in the COLLAGE desk top window have to have this COLLAGE look and feel. All other X windows, including the XCOLLAGE desk top window, have to be maintained with a normal X window manager. In general it is not helpful for the end user to work with two different window managers. Therefore it was obvious to extend COLLAGES window manager capabilities to replace the X window manager. The actual implementation of XCOLLAGE allows the user to choose any existing X window manager or to use the build in window manager called **CWM.**

† SINIX is a trademark of Siemens AG, München.

‡ COLLAGE is a trademark of Siemens AG, München.

While we were working on the window manager part of the project, the Inter-Client Communication Conventions Manual [Ros89a] was sent out for public review. Because the X Window Systems only provides mechanisms and no policy, these conventions try to formulate rules for how X applications should interact with each other. Because a window manager is a "normal" X application we have decided to follow this document.

In this paper we do not discuss all the details of interaction between clients and window managers or even all aspects of the ICCCM, like selections, cut-buffers and session managers.

2. Some aspects on implementing a window manager following the ICCCM

2.1. Basic tasks, start, save set

One important component of the X Window System is the window manager. Windows in the X Window System are organized in a tree hierarchy. At the top is the "root window". Window managers only control direct child windows of the root window called "top-level windows". Because the window manager is a client process and not built into the server, the user is free to use a window manager of his choice or even not to use any window manager. But usually a window manger will be used.

> [ICCCM]: "It is a principle of these conventions (in the ICCCM) that a general client should neither know nor care which window manager is running, or indeed if one is running at all.
>
> "A goal of the conventions is to make it possible to kill and restart window managers without loss of functionality."

Not all conventions are supported by direct Xlib calls or specified in the X protocol so they have to be carried out by a series of Xlib calls. (ie. colormap or icon handling).

Because the user is free to start a window manager of his choice, he might try to start a second one at the same time. This is not useful and might result in conflicts between those different window managers. Therefore it is necessary to make sure that only one window manager is running. This has to be done by registering interest in the **SubstructureRedirect** Event on the root window. Because only one X client at a time is allowed to select this event, an X error is generated if a second client tries to select this event type. To handle this error an error handler should be set up.

```
static int XErrorOtherWM ( display, error_event )
Display *display;
XErrorEvent *error_event;
{
   fprintf ( stderr,
      "It seems, you are running another window manager\n");
   exit(1);
}

void InitWM ()
{
   ...
   /* Tell server, CWM does Layout policy.
    * Exit if another window manager is running.
    */
   XSetErrorHandler ( XErrorOtherWM );
   XSelectInput ( display, root_window,
                  SubstructureRedirectMask |
                  SubstructureNotifyMask );
   XSync ( display, False );
   InitNormalErrorHandling ();
   ...
}
```

Because "it is possible to kill and restart window managers without loss of functionality", a window manager not only has to control windows that have been created after his startup. It is also necessary to manage all previously existing windows.

```
...
if ( XQueryTree ( display, root_window,
                  &root_return,
                  &parent_return,
                  &children_return,
                  &nchildren_return ) ) {

        for ( i=0; i < nchildren_return, i++ ) {
                ...
                CreateWindow ( children_return[i] );
        }
}
...
```

The CWM is a so called reparenting window manager. This means it modifies the window hierarchy and puts a container or decoration window in between the root window and the application window. Because of the window hierarchy, all child windows of a window are lost or destroyed if the parent window is destroyed. So this change in the hierarchy would delete all application windows if the window manager dies unexpectedly, and therefore his windows (and child windows) are deleted by the X server. X supports a save-set, that is a safety net for windows that have been reparented by the window manager. The windows in this set will remain alive, because they are automatically reparented to their closest living ancestor, whenever the window manager dies. This ancestor is the root window.

```
void CreateWindow ( window )
Window window;
{
   ...
   XAddToSaveSet ( display, window );
   ...
}
```

For space reasons, other standard actions of a window manger, like setting up the window title or looking into standard window manager properties, cannot be discussed in this paper.

2.2. Colormaps

An interesting topic is the usage of color. The X Window System allows virtual colormaps. Each window might have its own colormap. These colormaps have to be installed into one or more physical color lookup tables. One subsection in the ICCCM is dedicated to this topic. The key sentence (it differs from previous drafts) is that

> Clients must not use *InstallColormap* or *UninstallColormap*.

This task has to be done by the window manager. Many of todays X window managers and X applications will not fulfill the ICCCM requirements for handling colormaps correctly.

CWM is one of the first managers, that follows the correct ICCCM colormap conventions. This new policy also might require rework of client programs, that use private colormaps. If applications (un-)install colormaps directly, not expecting having a window manager performing this task unpredictable behaviour might occur.

The ICCCM introduces a new property called WM_COLORMAP_WINDOWS. Clients might set up this property if they want to specify a priority list for colormaps. This property should contain window-id's. The window manager will install as many of the attached colormaps as possible. If no WM_COLORMAP_WINDOWS property is set, the colormap, being the top-level window's attribute, is installed. Note, that if the top-level window does not appear in the list it will be assumed to be higher priority than any other window in the list.

CWM has the following colormap focus policy build in. When a client's top-level window or icon gets activated, that means it gets the input focus, it also get the colormap focus. Windows that have the colormap focus will be displayed in correct colors, while other windows might have false colors. CWM will ensure that for the activated client's top-level window the M most recently installed Colormaps are guaranteed to be installed, where M is the min-installed-maps field of the screen in the connection setup.

Because in X11R3 there is no Atom WM_COLORMAP_WINDOWS predefined, this Atom has to be defined implicitly by "XInternAtom".

An example of colormap handling is given in listing 1.

The window manager also has to react on changes to colormaps of top-level windows. CWM does this by handling the ColormapNotify event if there is only one colormap installed. If there is a colormap priority list, it will handle the PropertyNotify event.

2.3. Window State Policy

Client top-level windows are in one of three different states:

- NormalState. The client's top-level window is visible.
- IconicState. The client's top-level window is iconic, whatever that means for this window manager. The client can assume that its icon_window (if any) will be visible, and failing that its icon_pixmap (if any), or its WM_ICON_NAME will be visible.
- WithdrawnState. Neither the client's top-level window nor its icon are visible.

Newly created top-level windows are in Withdrawn state. Once it has left Withdrawn state, the client will know that the window is in Normal state if it is mapped, and that the window is in Iconic state if it is not mapped.

The client is free to change the state. The ICCCM describes conventions for a change from any state to any state. As an example, if a client application wants to iconify its top-level window, it has to send a special client message event with the atom WM_CHANGE_STATE to the root window.
The CWM will handle that event like this:

```
void ProcessClientMessage ( wop, ep )
WindowObj *wop;  /* Pointer to structure with several window
                    informations */
XClientMessageEvent *message_event;
{
        if ( message_event->message_type == XA_WM_CHANGE_STATE
             && message_event->data.l[0] == IconicState
             && (wop->window_state & cwmNormalState) != 0 ) {

                IconifyWindow ( wop );
        }
}
```

Because WM_CHANGE_STATE is an atom, that is not predefined in X11R3, we have to define it with XInternAtom:

```
#ifndef XA_WM_CHANGE_STATE
#define XA_WM_CHANGE_STATE \
        (XInternAtom(display,"WM_CHANGE_STATE",False))
#endif
```

As a second example, if the client wants to change the window state to Withdrawn state, in addition to unmapping the window itself he must send a synthetic UnmapNotify event. The reason for doing is, to ensure that the window manager gets some notification of the desire to change state, even though the window may already be unmapped (being in Iconic state for example).

```
/* tell window manager window changed to withdrawn state
 */
SynthUnmap (window)
Window window;     /* window getting unmapped */
{
   XEvent event;

   event.type = UnmapNotify;
   event.xunmap.display = display;
   event.xunmap.event = root;
   event.xunmap.window = window;
   event.xunmap.from_configure = False;

   XSendEvent (display, root, False,
               SubstructureRedirectMask|SubstructureNotifyMask,
               &event);
```

If CWM gets such an UnmapNotify event, it changes the window state to Withdrawn, destroying the window the client's window was reparented to.

3. Window Manager Functionality in XCOLLAGE

XCOLLAGE is a typical X toolkit program. There is a "main loop" which looks into the event queue and then dispatches all the toolkit events. To perform window manager operations, the CWM part of XCOLLAGE has interest in events that happen on the root window, that have no widget. Therefore we use a slightly modified main loop. The event is taken with *XtAppNextEvent*(). If the returned event has happened on the root window, the CWM dispatcher is called, otherwise the call is passed to *XtDispatchEvent*().

```
XtAppNextEvent(CollageContext,&event);
if ( event.xany.window == root ) {
      cwmDispatch(&event);
}
else {
      XtDispatchEvent(&event);
}
```

CWM was realized by designing and implementing a set of widgets for different purposes. For every top-level X window there is a special class "HolderWidget". It is a subclass of the standard XCOLLAGE "DataWidget" and makes it possible to interact with the X window as a normal COLLAGE window. Special functions are enforced with special callbacks. The realize method is shown in listing 2.

COLLAGE provides an icon button and a close button in its title bar. If the user clicks into one of these buttons, a specific event is generated and sent to the COLLAGE application. It is the responsibility of that program to perform the appropriate action. This differs from X.

4. Conclusion

In this project we have implemented the functionality of a proper window manger and made its look and feel visible to other X clients. These applications will get the same window decorations as normal COLLAGE applications. The internal interface follows the ICCCM.

In the above paper we have discussed some aspects like colormaps and window state policy. There are many more aspects we have to regard. See [Nye88a] for some of these aspects.

All told it is possible to integrate the functionality of a window manager into the emulation of another window system. It works even with very good performance in spite of a very high and complex programming level.

References

[Lor89a] Achim Lörke, "XCollage: Portierung eines Windowsystemes auf X11," *Proceeding at 12th DECUS Symposium*, München, 1989.

[Nye88a] Adrian Nye, *Xlib Programming Manual for Version 11*, O'Reilly & Associates Inc., Newton, Massachusetts, 1988.

[Ros89a] David S. H. Rosenthal, *X Window System, Version 11, Inter-Client Communication Conventions Manual*, 1989. Public Review Draft

Listings

```
#ifndef XA_WM_COLORMAP_WINDOWS
#define XA_WM_COLORMAP_WINDOWS \
        (XInternAtom(display,"WM_COLORMAP_WINDOWS",False))
#endif

static int NumCmaps;  /* number of installed Colormaps,
                       * if window gets colormap focus
                       * (getting activated)
                       */

/* InitCmap
 *
 * Initialize NumCmaps.
 */
void InitCmap()
```

```
{
    cwmNumCmaps = MinCmapsOfScreen(ScreenOfDisplay(display,screen));
}

/* InstallCmaps
 *
 * Install the colormaps for window.
 */
void InstallCmaps ( window )
Window window;
{
    Atom actual_type;
    int actual_format;
    unsigned long nitems;
    unsigned long bytes_after;
    unsigned char *prop;
    XWindowAttributes winattr;
    Window *cwp;                /* Colormap Window pointer */
    int i;
    int WindowInPropertyList;

    /* read WM_COLORMAP_WINDOW property
     */
    XGetWindowProperty ( display, window, XA_WM_COLORMAP_WINDOWS,
                         0, 100, False, XA_WINDOW,
                         &actual_type, &actual_format,
                         &nitems, &bytes_after, &prop );

    /* wrong property type
     */
    if ( actual_type != None && actual_type != XA_WINDOW ) {
          WarningMsg ( "wrong WM_COLORMAP_WINDOWS property type" );
          /* install window's colormap */
          actual_type = None;
    }

    /* get window's colormap
     */
    XGetWindowAttributes ( display, window, &winattr );

    /* IF   no property set, install window's colormap
     */
    if ( actual_type == None && !winattr.map_installed ) {
          XInstallColormap ( Display, winattr.colormap );
          return;
          /*NOTREACHED*/
    }

    /* ELSE process property:
     *
     * window in property list ?
     */
    WindowInPropertyList = False;
    for ( i-=(int)nitems-1; i >= 0; i-- ) {
          if ( winattr.colormap ==
                (Window *) (prop+(i*actual_format/8)) ) {
                 WindowInPropertyList = True;
                 break;
           }
    }

    /* install colormaps */
    i = MIN ( (int)nitems, cwmNumCmaps );
    for ( i-=1; i >= 0; i-- ) {
          /* get windowID for colormap
           */
          cwp = (Window *) (prop+(i*actual_format/8));
          /* install colormap
           */
          XGetWindowAttributes ( display, *cwp, &winattr );
          XInstallColormap ( display, winattr.colormap );
    }

    /* If window does not appear in list it has highest
       priority */
    if (! WindowInPropertyList) {
          XInstallColormap ( Display, winattr.colormap );
    }
```

```
    XFree ( (caddr_t) prop );
}
```

Listing 1: *An example of colormap handling*

```
/* realize DataWidget for WAM or X application
 */
static void Realize(widget, value_mask, attributes)
Widget widget;
XtValueMask* value_mask;
XSetWindowAttributes* attributes;
{
    if ( (DataWidget)widget->data.original_x_window == (Window)0 ) {
        /* WAM application */
        ...
        /* create WAM data window */
        XtCreateWindow (widget, (unsigned)InputOutput,
                        (Visual*)CopyFromParent,
                        *value_mask, attributes);
    }
    else {
        /* X application */
        Arg args[10];
        Cardinal n = 0;
        Widget holder;

        /* create data window for X application */
        XtCreateWindow (widget, (unsigned)InputOutput,
                        (Visual*)CopyFromParent,
                        *value_mask, attributes);

        /* create special subwidget for the X application */
        XtSetArg(args[n],XtNholderWindow,
                 (DataWidget)widget->data.original_x_window);
        n++;
        holder = XtCreateManagedWidget("holder",holderWidgetClass,
                                       widget,args,n);

        /* data window contains x window */
        XReparentWindow(XtDisplay(widget),
                        (DataWidget)widget->data.original_x_window,
                        XtWindow(widget),
                        0, 0);

        /* special event handling */
        XtAddEventHandler
            ( widget,
              SubstructureNotifyMask|SubstructureRedirectMask,
              False, ClientHandler, (caddr_t)NULL );
        XtAddEventHandler
            ( holder,
              PropertyChangeMask|ColormapChangeMask,
              True, ClientHolderHandler, (caddr_t)NULL );
}
```

Listing 2: *Realizing a dataWidget*

Teaching a Spreadsheet how to Access Big Databases

Michael Haberler
Martin Ertl

Hewlett-Packard Austria, Vienna
Technical University Vienna
mah@hpuviea.at
martin@hpuviea.at

ABSTRACT

While PCs are becoming commodity items, few attempts have been made to tap their processing power for *cooperative programs* in typical business applications. We view a cooperative program as set of distributed communicating processes with specialized tasks. In the context of a typical data entry/lookup application, the PC "process" might provide the user interface while a process running on a mainframe is responsible for database access. This might result in a fast, responsive user interface while lowering the mainframe and communications requirements. To explore the feasibility of this approach, we have built a sample cooperative program: a PC spreadsheet which can access a relational database running on a UNIX host. We describe different approaches to problem partitioning as well as our experience with using Sun Remote Procedure Call and BSD sockets as a programming interface for cooperative programs.

1. Introduction

Both spreadsheets and relational databases operate on tabular data. Assume you would like to analyze the result of a mainframe database query with your favorite PC spreadsheet. You probably would logon to the host using a terminal emulator, run your query, transfer the resulting file to the PC, convert into a format digestible by your spreadsheet, and run the spreadsheet on the result. Or one might have the database reside on the PC, or use a spreadsheet on a mainframe, provided there is one.

These usage patterns seem fairly widespread: PCs are mostly used "standalone", maybe with device- and file-sharing, or as dumb terminals, having limitations from both worlds. An alternative approach might be to build distributed applications where tasks are more sensibly assigned to machines, like having application and user interface code run on the front-end, while the database runs on a remote server. In our example, one could teach the spreadsheet program to directly access a mainframe database. One might provide functions to connect to a remote database, to specify a query and to insert the resulting table into the worksheet. This is the approach we have taken.

Our goals were twofold: first, to gain experience with distributed programs for typical business applications, and second, to explore the feasibility of peer-to-peer-communication between a a low-end computer running the application-specific code and a larger server. We did not intend to implement yet another spreadsheet, or replace commercially available products. Despite the narrow focus of our example, we believe that our approach might be applicable to a wide range of typical business applications, resulting in programs with better "look and feel".

2. Design Considerations

When designing a distributed application the tasks must be divided among the cooperating processes. Usually there are several possible choices about where to cut. To the user, the whole application appears as a single program; she should never be aware of the fact that more than one process is involved. The underlying structure of our spreadsheet example might be as follows:

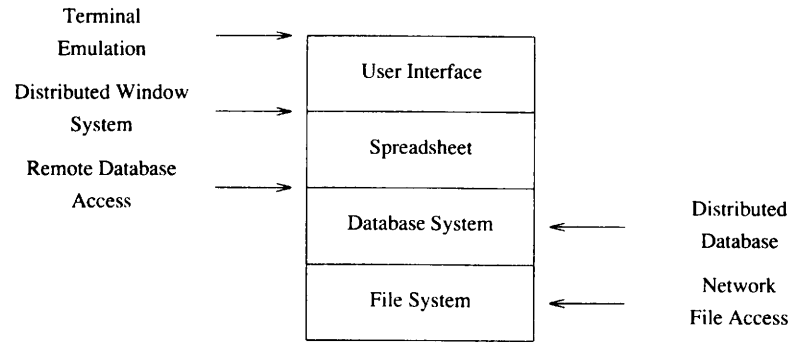


Figure 1: *Possible Interface Choices*

Terminal Emulation leaves all processing and functionality to the host side, resulting in host and network load and usually a poor user interface. The front-end remains essentially unused. *Distributed Window Systems* like the X Window system offload some user interface tasks from the host and may provide a high quality user interface. However, network traffic and processing requirements on the host are still high if only the X server rubs on the front-end. *Remote Database Access* leaves all spreadsheet and user interface functions to the front-end and accesses the host just for database operations through some kind of protocol. *Distributed Database* access might leave a part of the database on the front-end while accessing a host for data not available locally. This requires substantial resources on the front-end while not providing convincing advantages over the previous variant. *Network File Access* would leave the complete database system on the front-end and use the host only for file access on the file system level. Resource requirements are similar to the previous variant. The synchronization of several databases and data transfer volume might pose problems.

There are additional possibilites to divide the application; e.g. one might leave user interface and spreadsheet command parsing to the front-end while interpretation and database access is left to the host, or partition the database system. However, these interfaces are fairly "wide" and inelegant in the software engineering sense. For the particular example, one might also consider using the external representation of a worksheet. Besides from being specific to spreadsheets, this method also suffers from the plethora of different external formats currently in use (for example [Wal86a] lists five different spreadsheet programs, each having its own format for storing worksheets).

We have decided for the remote database access interface for the following reasons:

- There are good reasons for running a database system on a central host as opposed to distributing it over several small computers. Amongst them are access control, consistency, economies of scale and very likely a better-maintained environment. No distributed database is required just for remote access, with its associated problems of higher cost and likely version mixes due to different machine architectures.
- The interface for remote database access can be closely modeled after its programming interface.
- The traffic over an applications interface to a database system is considerably less than the database I/O traffic.
- Leaving the user interface and non-database operations in the front-end provides short, constant response time. The user interface can be made as fancy as the front-end is able to support.
- The front-end part of applications partitioned along these lines usually fits within the limits of low-end PCs without requiring expensive hardware upgrade, which would render this approach incompetitive compared to terminal-based approach.

For our example, a few prerequisites are necessary:

- A spreadsheet program as a basis for remote access extensions. In our case, we used a public-domain spreadsheet program available in C source code from the *comp.sources.unix* Usenet bulletin board. It runs under UNIX as well as MS-DOS.

- The database system must support formulation of queries at runtime when accessed from via the programming interface *(dynamic queries)*. Many database systems support this functionality, e.g. ORACLE, INGRES and Hewlett-Packard's Allbase. If all data types are known at compile time (like for a typical data entry/lookup application, as opposed to ad-hoc queries), this feature is not needed.
- Networking functions to establish program-to-program communication. We use the TCP/IP protocol suite as a foundation. This comes standard with many brands of minicomputers and workstations, and is also available for mainframes and PCs. On the PC side, we used the library available from Network Research Corporation [NRC88a], and in a second implementation, Sun's PC-NFS Programmer's Toolkit [Sun87a]. Both libraries provide the Berkeley *socket* interface; Sun's product also provides the RPC/XDR Remote Procedure Call Interface.
- Conventions for exchanging data between host and front-end to cope with different data representation like byte ordering, word size and floating point format.
- Hardware to support the networking functions. In our case, we used an Ethernet card for the PC connected to a LAN.

3. Implementation

The implementation involved three steps: a set of remote database access primitives, a server to provide them, and extensions to the spreadsheet to use them.

3.1. Remote Database Interface

We did two implementations, the first one being based on a TCP transport, the second implementation is based on Sun RPC on top of TCP. We defined a set of data types and database operations which was to be general enough to allow opertion with at least Oracle and HP's Allbase (these we had at hand). Both employ a database-cursor-based programming interface, which we used as a foundation. Figure 2 shows a subset of the *rpcgen* specification.

```
program REMSQL {
        version REMSQLVERSION {
                rsqlca      CONNECT(conn_info) = 1;
                rsqlca      PREPARE(string) = 2;
                description DESCRIBE(desc_info) = 3;
                rsqlca      OPEN_CURSOR(void) = 4;
                row         FETCH(void) = 6;
                rsqlca      CLOSE_CURSOR(void) = 7;
                rsqlca      EXECUTE(void) = 8;
                rsqlca      COMMIT(void) = 9;
                rsqlca      ROLLBACK(void) = 10;
                void        EXIT(int) = 12;
        } = 1;
} = 200001;
```

Figure 2: *Remote Database Interface Description*

Connect() conveys authorization information. *Prepare()* submits a SQL statement to the SQL parser. *Execute()* executes a parsed statement. *Describe()* returns the data types for a row after executing a query. *Open_Cursor()* is used to define a database cursor, which is essentially a pointer to the "current row" in a relation used in insert/delete, iterator and *fetch()* operations.

3.2. Database Server

For the TCP-based version, we used a demon process started by inetd(8). This demon connects to the database based on the authorization information passed with *connect()*. The actual database interface was implemented using the standard SQL preprocessor for C. The protocol handler is a state machine driven by input events and database state. Protocol requests and replies are encoded as ASCII encoded messages which takes care of architectural differences. This demon actually does not know about being run over a network connection and thus can easily be tested interactively.

The RPC-based version was implemented by mapping the remote operations onto a rpcgen specification, and the resulting XDR data conversion functions provide machine independence. The server process is started by inetd(8) with help from portmap(8).

3.3. Spreadsheet Extensions

The front-end part involved an extension of the spreadsheet with commands to connect a remote database, edit and execute a query and specify the coordinates for the resulting table within the worksheet. This proved easy due to the use of the *yacc* parser generator.

The whole package was developed on a UNIX machine and after testing the front-end was moved to a PC. The PC and UNIX versions share the same code, as the networking interface provided by both DOS libraries is identical to the one found on UNIX versions derived from Berkeley UNIX. Implementation effort was about 5 person weeks by the second author who knew the C language but had no previous experience with networking and *yacc*.

4. Results

The resulting program is simple to use. The fact that actually more than one machine is involved is transparent except for the hostname which is parameter for the *connect* command. A query may be started by using the *sql* command. A default editor is used to create or modify the query; it can be replaced to match the user's taste. The location of the result table is determined by the *destination* command.

Performance of the first implementation was poor. A query resulting in a 14-row by 8-column table took about five seconds. We traced this to inefficient use of the TCP stream through single-character I/O. Instead of improving this version we moved on to the RPC-based approach. This improved performance by an order of magnitude. Using UDP instead of TCP as a transport showed only insignificant improvements; however, tranferring the result of a query as an XDR array brought another 30% improvement.

ASCII Stream	**RPC** one call/row	**RPC** XDR array
316	33	22

Figure 3: *Time to transfer a 490-row table with 8 columns (seconds)*

These figures are medians of five runs and were measured between HP9000/350 computers running HP-UX Revision 6.5.

There is still room for improvement. A promising approach is the use of *batching* [Sun88a]. Batching introduces asynchrony by sending multiple requests without waiting for replies until after the last request. This approach also has been taken in the X Window Protocol for performance reasons. Some timings we have done show that by batching a speedup of at least half an order of magnitude can be achieved for long batches. In our case, this could speed up the iteration over the result table, which is the primary bottleneck.

5. Lessons learned

When embarking on a related project, these are points to watch for:

- There is no equivalent to *stdio(3)* for database access yet. Even with de-facto standards like the SQL preprocessor interface, annoying differences remain. For example, SQL statements passed as strings must be delimited by a semicolon with one system and may not with another. Although it might be feasible to define a class which could then be mapped onto several database systems, the sequence of calls through this interface varies, and thus either must be hidden under the interface layer or exported to applications.
- The idea of stateless servers [Sun88b] does not work too well for remote database access, at least with current programming interfaces. Most database systems follow a model similar to *login*, and keep a substantial amount of state information which is hardly accessible, e.g. the database cursor. Besides, if the database system is fragile, a stateless server will hardly make it robust.

- RPC by default provides access to a **single** server process from several clients. This is different from the inetd(8) model where each connection request is usually handled by a different process. In our spreadsheet example, this implies that the first client's database access authorization might be used or changed by another client if credentials tied to processes on the server. On the other hand this is an elegant scheme for multi-access applications.
- The cost of remote procedure calls is substantially higher than local calls. Iterators might have to be redesigned, batching requests if possible. Unfortunately, Sun RPC batching is currently only implemented in the client-to-server direction, although a suitable mechanism for the other direction exists with the broadcast RPC interface.
- RPC might not be the right paradigm; asynchronous message passing based on strongly typed streams (maybe using XDR without RPC) might be an alternative.
- Use all the type checking you can get, especially when running the code on different architectures.
- RPC requires compile-time binding of argument and result types; a dependency which must be managed. In many cases it is possible to generate RPC data type definitions by tranforming a data dictionary entry.
- Even if you dont use RPC, rpcgen is a sane way to create XDR routines automatically.

References

[Sun87a] Sun Microsystems, Inc., *PC-NFS Programmer's Toolkit User Manual*, 1987.

[NRC88a] Network Research Corporation, *Fusion Network Software – Programmer's Reference Manual*, Oxnard, CA, 1988.

[Sun88a] Sun Microsystems, Inc., *ONC/NFS Protocol Specifications and Services Manual*, August, 1988. Part No. 800-3084-10

[Sun88b] Sun Microsystems, Inc., "ONC-Open Network Computing," *Tutorial given at the 1988 Winter USENIX conference*, Dallas, TX, 1988.

[Wal86a] Jeff Walden, *File Formats for Popular PC Software*, John Wiley & Sons, New York, 1986.

System Security – Administration Through Automation

Dale A. Moir

Lachman Associates, Inc.
1901 N. Naper Boulevard
Naperville, IL 60540
USA
dale@laidbak.uucp

ABSTRACT

A suggested approach to implementing an automated computer security policy is presented. Policy considerations ranging from physical security to file permissions are discussed in detail. In each case, methods of integrating the security policy considerations into an automated procedure are described. The cost effectiveness of an automated policy is defined in terms of user education, administrator training, and relative advantages over alternative methods. A brief section on disaster recovery is also included, as the same approach to automation may be applied in this area as well. Finally, the resultant computing environment, with full security measures in place, is described from the user and administrator perspectives.

1. Introduction

It has long been recognized that *information* can be a very valuable resource. This is true for governments, corporations, and individuals. In the age of electronic data processing, the task of protecting the information resource has evolved along with the technologies used to process and store the information. Most of the traditional resource protection schemes consist primarily of a set of policy directives, defining how the resource is to be managed and protected. In the case of information, a security policy should encompass both the physical and virtual representation of the resource. More specifically, this amounts to a computer security policy that addresses both hardware and software security issues.

Designing and implementing a computer security policy can seem like a monumental task. However, given that most of our information processing tasks have been automated, it follows that the information security tasks can be treated in a similar manner. Continuing with this line of reasoning, we might guess that a small amount of up-front effort (developing proper tools), will result in a considerable reduction of effort in the long term.

This paper considers the task of formulating a reasonable security policy for a research and development environment. At the same time, in an effort to adhere to the methodology described above, each phase of the policy will be designed for automation (where possible). The resulting set of tools will be designed to 1) implement the policy, 2) insure its execution, and 3) educate the users in the process.

An automated approach to computer security can ease the difficulty of policy implementation, *and* reduce the effort required for user education. Note that the concepts discussed in this paper can also be applied to computing environments where more stringent security requirements exist.

2. Designing a Security Policy

Our goal is to provide a security policy for a research and development (or similar) environment. We will begin by defining those areas that our security policy should address. These are:

Physical security – site and computer facilities

Access security – access to computer resources, either remotely or locally

Resource security – protecting the resources that people can access

Recovery mechanisms – correcting problems, restoring functionality

In the following sections, we will outline a computer security policy that addresses each of the above issues. In doing so, we will consider ways to automate the task of security administration.

2.1. Physical Security

Physical security consists of three major categories:

1) Building Access – the ability to enter the office complex
2) Computer Room Access – the ability to access the computing hardware
3) User Access – the ability to utilize the computing facilities

We will consider each aspect (or "layer") of physical security in detail.

2.1.1. Building Access

For most companies involved in research and development, some form of controlled entry system is required. The selected method of control may range from a single entry point monitored by a receptionist to a system of employee identification with verification checkpoints.

The security policy essentially dictates that all persons entering the building must be authorized employees. The approach used to enforce this policy will depend on the size of the installation in question, and on the perceived security requirements. A common approach is to issue all employees some form of identification – a photo identification badge, for instance. Then, upon entering the premises, each employee displays their identification as a means of authorization.

This approach works well, but is somewhat labor intensive. In the interest of automating *both* the entrance and authorization steps, one might issue electronic key-cards to their employees instead. A key-card electronically stores an employee's identification and building access privileges. In many cases, use of such a device is a sufficient mechanism for providing building access security. The key-card notion can be combined with a photo identification badge as well, by means of a magnetic strip or other electronically readable media.

The advantage of a key-card or similar system is that it automates the verification process. In addition, employee access privileges can be limited to certain access points, and certain time periods. For example, certain employees may only require access through the main entrance, during the hours of 7:00 a.m. to 8:00 p.m. These access limitations can be specified in most key-card access systems. Such a system may also provide access logging, and similar advantages not available in mechanical key systems. Regardless of how building access is controlled, some thought should be given as to whether or not the process can reasonably be automated, or otherwise integrated into our overall security policy.

2.1.2. Computer Room Access

In addition to requiring basic building access control, many companies also require access control for "sensitive" areas within the building. Again, the implementation of this policy decision may take a variety of forms. If a key-card system is used, it becomes a simple matter to limit physical access to selected resources via the key-cards. The computer room is a good example. In most facilities, the computing hardware is maintained in a temperature-controlled environment. And, since the physical hardware embodies the information processing capability, it represents a physical security concern. Thus, the computer room is a logical candidate for key-card or similar access protection.

2.1.3. User Access

The remaining aspect of physical security that we wish to address is user access. User access can be defined as the ability to access the computing equipment via log-in or other user interface. In general, user access requires physical access to a terminal or other I/O device. Thus, the first line of defense against unauthorized user access is to prevent unauthorized access to computer terminals.

If user terminals are grouped in separate rooms, then key-card access can be applied to the terminal rooms. More commonly, terminals are distributed among employee offices, or otherwise openly available. In either case, the risk associated with terminal access should be minimized. We will consider the login/password authorization sequence in a later section devoted entirely to software access control mechanisms. In this section, the issue of easily accessible logged-in terminals must be addressed.

Efforts to control physical access to computer terminals may or may not be effective. An additional security policy consideration should address the exposure created by unattended, logged-in user terminals. This is perhaps the most common source of risk for unauthorized user access. The policy statement should basically specify that logged-in terminals are not to be left unattended. Efforts to automate the enforcement of this policy may include an idle terminal log-out program, or software utilities for terminal "locking". If an appropriate mail message or other electronic warning is issued prior to log-out, then users will begin to understand that unattended, logged-in terminals are a considerable security risk.

2.2. Access Security

Access security requires that only authorized personnel can access the computer resources. Of course, physical security is a substantial part of the access security problem. Assuming that physical access has been adequately addressed, then the task becomes one of maintaining a policy for access to on-line resources. In many systems, some level of access control is provided by the operating system. Since we are addressing a research-type environment, we will use the UNIX operating system in our examples. UNIX is a reasonable choice because it exemplifies the "open" nature of research and development computing environments, and still offers a fairly robust set of security features. In less "open" environments, the same examples will apply, but the implementations will vary somewhat.

One facet of access security is insuring that legitimate users do not allow others to easily gain access to the system. The first line of defense against unauthorized access in UNIX is the login/password verification sequence. Thus, a good example of access vulnerability is an easily guessed password. A good security policy will include some provisions for password management.

2.2.1. Password Management

Basically, our policy should be designed to insure that the login/password sequence is utilized to its fullest advantage. For the most part, this consists of insuring that users select and maintain "good" passwords. A "good" password will be defined as one that is reasonably complex, difficult to guess, and yet still easy to remember. Since password selection is a distributed responsibility, there are several considerations to be addressed by our security policy.

The obvious first step in addressing the password issue is user education. Barring any form of password assignment, users will be responsible for selecting their own passwords. Thus, each user should be aware of the significant role that passwords play in controlling computer access. Also, some minimum requirements for password complexity should be defined and explained. Our policy may dictate that users should be familiar with the password management guidelines, and at the same time specify what the guidelines will be. Advising against password sharing and similar procedural errors can be included in our policy statement. In addition, we can attempt to include automated measures designed to enforce our password policy.

Given that a minimum complexity criteria has been defined for user passwords, it is possible to include a software check that determines if a user's new password meets the criteria. For example, if we require that passwords be a minimum of six (6) characters, and include at least one (1) non-alphabetic character, then these criteria can be checked in software when a user selects a new password. In fact, many operating systems provide password complexity checking as a standard feature. It would be reasonable to suggest that employees refer to the password policy guidelines within the error message that rejects a "bad" password.

Password complexity checking ensures that users' passwords contain a sufficient number and mix of characters to thwart a naive attempt to gain unauthorized login access. Still, the use of a password like "wizard1" will meet our complexity criteria, but would not necessarily be difficult to guess. If passwords are to meet all of our suggested criteria, then an additional, automated approach may be in order. For example, we might have a program designed to "guess" users' passwords during idle machine cycles, or during off-hours. Such a program could rely on a variety of information, and might send mail to users whose passwords are discovered. Again, it would be appropriate for the mail message to direct the user to the password management guidelines.

The final consideration to be applied to user passwords is the length of time permitted between updates. Since password effectiveness tends to erode over time, it is reasonable to require a periodic password change as a part of our security policy. Many systems support password aging, and so again we have available an automated method of ensuring that our policy directives are followed.

Note that the access privilege itself may also be allowed to expire – much like a password. For example, users that do not log onto the system for some specified amount of time might be considered "expired". Allowing unused logins to remain on a system allows intruders a stable environment for subversion. Our policy should also include a provision for forfeiting access privileges by default – if events indicate that access is not really required. The automation of this policy requires a tool that examines user logins for recent activity, and flags those users whose privileges have expired.

2.2.2. Dial-up Access Control

Assuming that we have done our best to insure that users select "good" passwords, and that our facilities are not easily roamed by intruders – or if so, that idle terminals and other forms of machine access are limited, we are left with one more line of attack. This is the modem connection, or dial-up access. This can include user dial-up for login, or network connections that utilize the telephone network. An example in UNIX, our reference system, might be the uucp networking facility.

Dial-up connections can be addressed in a variety of ways:

Access Logging – each user's dial-up session is logged according to date and time of access. One can then check for unusual login patterns.

Failed Access Hang-up – if it is possible to determine in software that a dial-up line is being used, then the login program can terminate after some number of failed attempts. This will discourage a "brute force" attack on your password space, by forcing a would-be intruder to reconnect after just a few login attempts.

Dial-up Password – of course, if a dial-up terminal is easily detected at login, then a prompt for a dial-up password can also be installed. This provides an extra layer of access security. Note that dial-up passwords that are *shared* among many users are limited in their effectiveness.

Access Limiting – it may be reasonable to limit the hours that dial-up access is available. In a research environment, however, it is unlikely that limited access will be desired.

Dial-back Software – on systems with dial-out modems, the task of adding a dial-back feature to the login program is relatively straightforward. A dial-back feature terminates the incoming call, and calls the user *back* at a predefined number.

Dial-back Modems – dial-back modems are also available, and provide a dial-back capability *without* necessarily invoking the login program. These are an attractive alternative for administrators that do not wish to modify their system software.

It may be reasonable to assume that your password management scheme will adequately address dial-up access control. However, if any of the above-mentioned mechanisms are available, then they should certainly be considered. Again, it is a policy decision to determine what constitutes reasonable dial-up access capabilities. This policy decision should be reinforced with automated mechanisms that enforce the policy directives.

2.2.3. Network Access Control

The final issue to address under dial-up access is networking utilities and protocols. In many cases, system network activity is on an auto-timed, poll-and-respond type basis. That is, the network activity takes place without administrative intervention, responding to requests by users. The potential access control issue stems from the fact that networking software is often riddled with security holes. In our example environment, where UNIX is the target system, the most common network connection is via "uucp". The uucp facility consists of software utilities, and so-called "configuration files" that determine how the utilities can be used. For example, the configuration files determine what other systems can dial in, what they have access to, what commands they can run, etc.

From a security standpoint, networking software is a virtual nightmare. However, as the marketplace has illustrated, connectivity and communication are essential to productivity. For most sites, the limited amount of risk incurred by *having* network connectivity will be offset by the increase in productivity.

The security policy on network connections is simple. Network utilities should perform their function in as limited a manner as possible. Given the capabilities of the network software, the security policy should dictate *exactly* what the network capabilities should be. This done, automated methods of verifying that the policy directives are followed can be designed.

In the case of uucp, the network capabilities are defined by the software specification and uucp configuration files. Based on this information, we can write policy directives that specify exactly how information is to flow through the network, and what tasks the network software can "legally" perform. This done, we are left with the task of automating the network evaluation task. The goal is to write a software utility that interprets the configuration files, and based on the implicit capabilities of the networking software, outputs a list of security problems, or a description of network capabilities. It then becomes a simple matter for the administrator to determine if the network configuration conforms to the security policy.

The added benefit of *automating* the network configuration process is that future administrators will not have to be experts on how the network software works. By using the tools, they can easily determine what effect their configuration file changes will have on the network. Thus, in addition to saving the administrator's day-to-day time, you also have a tool that allows novice administrators to perform adequately, and at the same time *trains* them in network administration. Again, with a small investment up-front, your long term gains are substantial.

The access control problem has been addressed at the physical security level, at the system log-on level, and at the dial-up level. Additional concerns arise once a user is logged into the system which may legitimately be called access control issues. However, once a user is logged on to the system, the problem is really one of resource control – with resource *access* being one part of the overall issue. Resource security is covered in the following section.

2.3. Resource Security

Resources in a computer system can be categorized in a variety of ways. When designing a security policy, the following resources should be considered:

Computer time – execution time is a resource.

Storage space – primarily disk storage, also memory.

Ownership domain – users on a system have resources that they "own", such as files and directories. This defines an ownership domain for each user, and a "system" domain as well.

Execution domain – users' execution privileges on a system also define a domain – and other users may attempt to usurp those privileges. The same is true for "system" or administrator privileges.

In essence, we can view ourselves as benevolent caretakers of a microcosm. Our users have rights, and privileges, and commodities, and we must insure that these are protected. Now, some responsibility is left to the individual users, but we can assist them in a wide variety of cases. Also, we must take care that no one should usurp *our* privilege – the administrator privilege – for this privilege overrides all others. The following sections examine security for the above-mentioned resources in detail.

2.3.1. Computer Time

Computer time is perhaps the most difficult resource to protect. On most systems, there is nothing to prevent each user from creating a CPU-intensive job, starting it, and collectively bringing the machine to its knees. This issue is addressed only through access control in most cases – if a user has access, it is assumed that his or her job is legitimate. More sophisticated systems might try to limit or recover from CPU overload, but such a feature requires that some "reasonable" limit be placed on CPU execution time. Assuming that this is possible, perhaps even on a per-user basis, then a simple tool that periodically checks the cumulative CPU utilization of each active process is called for. This tool might also be given the power to terminate any processes that have exceeded their CPU execution-time limit.

2.3.2. Storage Space

Most computer storage is in the form of disk space – the so-called file system. While some systems employ a more sophisticated storage hierarchy, this discussion will be limited to user files and directories. In reality, the storage space for each user might be considered a part of the ownership domain – space that the user "owns". We will make the distinction between things that are owned (files) and places to put them (space). Here we will consider only space.

In our example system (UNIX) users have what is known as a login directory – a directory in the file system that is their current directory at login time, and under which they are free to create their own files and subdirectories. All of the users whose login directories share a common file system will collectively share that disk space. This sharing occurs on a first-come first-served basis. In order for a user from another file system to "steal" disk space from these users, a file or directory that is writable must be located. If such a file or directory exists, the malicious user may overwrite the file with new information, or add his own files to the directory. The same considerations apply to files and directories "owned" by the system – although some file space is *supposed* to be shared.

Our security policy should dictate that user file space be clearly partitioned. Files belonging to one user should not be openly accessible to other users. At a minimum, write permission should always be partitioned on a per-user basis. If file space is to be shared, then a predefined mechanism for file sharing must be provided.

In the interest of automating our policy directives, several checks can be performed. In addition to the resource issue, a more security-minded concern is vulnerability to attack via trojan horse, virus, or spoof – any of which can be installed in a writable login directory. If we devise a tool that checks for files owned by "others" beneath a user's home directory, then files installed by other users can be detected. Additionally, a search for writable executable files will turn up any tools that can be easily overwritten with a spoof, virus, or trojan horse. Finally, a search for writable files and directories might turn up instances where storage space can be "stolen" by other users. This final check might not be necessary in a research or similar environment, unless disk space is specifically charged for.

An additional check, perhaps more important than the above, can be developed for systems such as UNIX. Users on a UNIX system control an environment variable called "umask", which can be assigned some value. The value of this variable determines the default access permissions assigned to each file that the user creates. If this variable is not set, then any files that the user creates may be writable by default. Thus, an additional check that our administration tool might perform would be to verify that users' default permissions meet some minimum security criteria.

In short, our storage space policy basically states that users will not leave their own storage hierarchy vulnerable to attack, or to unauthorized use by others. Once again, our automated tools provide a means for the administrator to insure that the users are protecting themselves. These same tools can be used to insure that system-owned files and directories are likewise protected. Note that the tools serve to implement the policy, and the design of the tools incorporates the reasoning behind the policy. If warnings of policy violations are tempered with informational messages, then the tools serve to educate the users and administrators, as well as automate the task of ensuring that the policy is followed.

2.3.3. Ownership Domain

The idea of ownership is now limited to *items* that may be owned, as storage space for these items is covered above. The difference can be likened to someone stealing your car, versus someone stealing your parking space. The act of "stealing" a file can take several forms. First, a user can change the ownership on a file so that the original owner no longer controls it. A user might also *copy* a file to another location, even though he or she is not authorized to do so. The act of overwriting or removing a file is covered in the above section, as the resource "taken" is really the file space – even though the file *contents* may also be lost.

In most systems, each user is free to "give away" files by changing their ownership to someone else. The ability to "take" files from other users is limited to the administrator only. In cases where ownership modification is an issue, some form of monitoring is appropriate. That is, a periodic check should be run to insure that critical files do not have their ownership modified. Security policy should define the *correct* ownership and permissions on all critical system files, and our security tools should verify that the current state of the system matches the expected state.

Unauthorized copying of files can be prevented in much the same manner. Those files that are not to be copied (such as games) should be subjected to an occasional global search. Should an unauthorized copy turn up, the owner of the file should be reminded of the policy, and asked to remove the unauthorized file.

2.3.4. Execution Domain

Execution domain defines the set of programs that a given user is authorized to execute. Under this category, a variety of issues needs to be addressed. These include:

System Integrity – the executable files provided by the system must be monitored for integrity. Any modification of system utilities, either by virus or unauthorized user, must be detected.

Trusted Paths – users should be protected or protect themselves against illegitimate command substitution. In UNIX, this must include verification of utilities (as above) *and* PATH variable verification.

File Permissions – execution permissions on system files must be set up such that unauthorized users are denied access. This requires a periodic check of system file permissions.

Domain Variance – some operating systems, such as UNIX, allow a program to vary its permissions upon execution. An example is the use of setuid or setgid programs. These programs assume the privileges of the *owner* of the file upon execution. As such, all instances of such files must be specifically authorized, and their contents monitored.

File Naming – in addition to trusted path verification, a security policy must dictate that command names be *unique*. This helps to insure that users do not execute spoofs or trojan horses instead of the intended utility. Again, periodic checks of the file system should insure that system program names are unique within the file system.

Log File Monitoring – many utilities keep a log of their activity for billing or security purposes. These logs can be automatically checked for evidence of security breaches.

Known Bugs – well-known software bugs may exist that allow a breach of security.

2.3.5. System Integrity

If security policy dictates that system file integrity is to be maintained, then some method of verifying integrity must be devised. In the case of utility programs, the most effective method is the use of checksums. One approach is to design a system utility which records the checksum of each critical system file, and periodically verifies that the current checksum matches the expected value. This amounts to a periodic check of the *contents* of these files.

2.3.6. Trusted Paths

Trusted path verification is subject to interpretation on the system in question. In our example, the UNIX environment variable PATH would be subject to verification for each user. The PATH variable defines a list of directories to be searched (in sequence) to locate command names. Such a PATH would have to be checked for legitimate system directories, along with an integrity check of any directory found in the PATH itself.

2.3.7. File Permissions

The execution permissions on the utility programs in the system implicitly define the execution domain of each user. This should be compared against the *expected* execution domain. Again, an automated utility program which defines the "correct" set of system file permissions, and periodically checks the actual permissions is needed. Directories, a special type of "file", can also be protected in this manner.

2.3.8. Domain Variance

The ability to vary one's execution or access permissions can be a powerful tool if properly used. It can be even more powerful if subverted for use by an intruder or other malicious user. In UNIX, the setuid/setgid feature allows a system designer to permit limited, distributed access to privileged system files. However, it also creates a significant level of vulnerability. For example, a setuid copy of the command interpreter (shell), owned by userid 0 (root), would allow a user to invoke a command interpreter such that all commands submitted may be executed with administrator-level permissions. As such, the use of this feature must be carefully controlled. To automate this policy, a list of "authorized" setuid/setgid files might be compiled, which are then compared against the set of *existing* setuid/setgid files. Needless to say, these files' contents and permissions must also be carefully monitored.

2.3.9. File Naming

As we have noted, command execution in UNIX depends on the command name matching a file name that appears in a PATH directory. As such, the substitution of a spoof or trojan horse for a legitimate command, using the same command name, can be very effective. Assuring that users set up their PATH variables appropriately is the first line of defense. An additional policy measure is to require that command names be *unique* in the file system. This policy disallows executable files whose names match those of legitimate commands. Also, a check for illegitimate file names is easily devised, and an automated periodic check can be performed.

Again, UNIX supplies a special case that must be considered. Users may have the ability to "alias" command names. This feature can be used to cause the command interpreter to substitute some *other* command name for the one that the user typed. This feature can be subverted if a user's alias list can be modified, or if the files associated with the alias list are themselves vulnerable. This implies that a tool to check users' alias values might be useful.

2.3.10. Log File Monitoring

Some system utilities keep a log of their activity so that illicit use can be detected, or so that billing can be performed or verified. If security can be enhanced with a periodic check of these log files, then this task can also be automated. In the UNIX operating system, several of the networking facilities keep an activity log, the activity of user terminals is logged, and use of the "su" program (which grants super-user privileges) is also logged. In each case, a periodic check of the log files to insure that only authorized, reasonable use of these facilities is in order. An example can be drawn from the physical security discussion of idle user terminals. If the log files indicate that some users tend to leave their terminals idle for long periods of time, then these users might be reminded of the policy regarding unattended terminals.

2.3.11. Known Bugs

In many systems, and UNIX is no exception, there exists an ever-changing set of known software bugs, some of which can potentially be used to breach system security. In addition to remaining abreast of these issues, it may be wise to devise an automated test for each known bug as soon as it is discovered. By doing so, it is possible to re-check your system for known security risks after a new operating system release is installed. Also, if your company uses several implementations of the same operating system, supplied by different vendors, it becomes a simple task to check *all* of your systems in one simple operation.

To summarize, the use of the execution domain must be clearly spelled out in our security policy. This done, an automated method of verifying the integrity of the execution domain can be devised. This allows the administrator to verify that policy is maintained, and provides a working implementation of the policy itself.

The final area that we wish to discuss is recovery mechanisms. The discussion up until now has focused on preventive measures.

2.4. Recovery Mechanisms

Our computer security policy has thus far been concerned with protecting our computer resources. Implicit in our discussion, however, is that some method exists for *correcting* any problems that might occur. In most cases, the correction scheme is relatively straightforward. For example, a poorly selected password need only be replaced with one that is more sophisticated. A system file with incorrect permissions need only have its permissions corrected. However, what does one do when a system file becomes corrupted? We have assumed, of course, that a back-up copy, or an uncorrupted version, can somehow be obtained.

The final addition to our security policy is the mechanism required to *recover* from security breaches. Note that the security policy can be considered a component of an overall disaster prevention and recovery scheme. The recovery mechanisms for some security "disasters" are the same as those for some physical disasters. A case in point is file system back-up tapes. Back-ups provide a means to restore disk files. Whether the disk files are destroyed by a head crash, a fire, or a disk virus is inconsequential in terms of recovery. In each case, a well maintained back-up library provides the recovery mechanism.

By considering our security policy as a natural extension of an overall disaster recovery plan, we can extend our previous line of reasoning to include recovery tools. That is, a tape back-up system, or utility, can be integrated into an overall recovery mechanism. If the back-up function is automated, then there is additional assurance that the task will be performed on a timely and regular basis.

Thus our first line of defense, and the most cost-effective, is a comprehensive set of preventive measures designed to avoid disasters – security or otherwise. In the event that those measures are not sufficient, recovery mechanisms must be put in place so that a total loss is avoided. For mechanisms like a tape back-up library, issues such as the time between back-ups, and the storage location of the tapes themselves (off-site?) must be considered.

3. The Resulting Environment

We have now described a variety of computer security concerns, and the ways in which they can be addressed. Our goal was to devise a comprehensive security policy at the outset, and build a set of automated tools to implement that policy. Once this mechanism is in place, a variety of benefits can be realized.

First of all, the policy itself exists in theory *and* in practice. That is, new users can be made aware of the policy as part of their education, but the task of implementing the policy is not entirely left to the distributed user community. Instead, the policy is implicit in the set of utilities that implement it. This way, should a user or administrator stray from the accepted policy, the tools will notify them of the security breach, and remind them of their responsibility.

Second, the task of security administration is largely automated. This simplifies the job of the system administrator, and ensures that security concerns are continually addressed. Even if the administrator is not knowledgeable in security issues, an adequate level of security can still be maintained. And, like any other skilled profession, continued use of the tools provides a continuing education in security concerns.

Finally, by formalizing the security policy through implementation, the task of *changing* the policy is reduced to software maintenance. The re-education effort is not necessarily reduced, but the effort required to verify that policy is adhered to is reduced significantly.

In the course of this discussion, we have described a long list of security related tools and practices. For the most part, employing these tools and practices would not detract from normal computer operations, or normal employee activity. True, each user may be required to think about security a little more seriously, such as when choosing a new password, but most day-to-day activities would remain unchanged.

Assuming that administrators are responsible for maintaining computer security, and that the tools described are made available, their overall workload should actually be reduced. The savings are derived from our automated methodology – most of the security checking is done in software, rather than through human effort. This leaves the administrator free to perform more important tasks – like solving problems and helping users.

Note that in no case have we assumed that the operating system itself must be modified, or that additional layers must be added to the user interface. That is, users would still see the same UNIX system that they are familiar with, even though we maintain a reasonable level of security. Thus, our security policy implementation does not adversely affect productivity, or require additional equipment or staff.

4. Conclusion

What I have tried to illustrate in this paper is the application of a well-known principle in computer science. That is, if you spend a little time up-front developing proper tools, your long-term costs in time and development will be greatly reduced. In the computer security arena, this amounts to a set of administrative tools that implement a corporate computer security policy. In the case of UNIX and other well known operating systems, these tools are readily available on the commercial market – and in fact form the basis for this discussion. The use of a commercial tool provides several additional advantages for the security practitioner:

1) You can be relatively certain that most major issues have been addressed, without developing the expertise yourself.

2) On-going support and development can often be left up to the vendor, through some sort of support agreement.

3) The cost of acquiring a tools package is fixed, which simplifies the task of incorporating computer security plans into the budget.

In summary, a good security policy can be made much more effective if it is automated. By automating the procedures for maintaining security, the task itself is simplified, security is much more likely to be maintained, and the educational curve can be reduced. Finally, commercial automation packages eliminate the need for development and maintenance, and at the same time provide a comprehensive administration policy that might not otherwise be developed. After all, policy without practice is no policy at all.

5. Bibliography

John Cray, *An Automated Approach to UNIX System Security*, in Communixations, Vol. VII, No. 5. /usr/group, Santa Clara, California, September/October 1987.

F. T. Grampp and R. H. Morris, *UNIX Operating System Security*, in AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2. AT&T, New York, New York, October 1984.

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Patrick H. Wood and Stephen G. Kochan, *UNIX System Security*. Hayden Books, Indianapolis, Indiana, 1987.

Lettermatrix for the selection of passwords through the user

Ernst Piller

BULL Austria
Linke Wienzeile 192
A-1150 Vienna, Austria
piller@atbull.at

ABSTRACT

In UNIX a user authenticates himself by entering a secret password. Most users choose simple passwords (short words and/or words with a special meaning), which are changed frequently. The result is a lack of security. In this paper, a new method for the selection of passwords by the user is beeing introduced. The method is based on a usual (traditional) secret password and on a lettermatrix, which is displayed on the screen of the user. The user has to find his way through the lettermatrix, based on his usual secret password, and he will find during and at the end of these steps, his actual password. These passwords constantly change (one-time passwords) and stand out because of their security.

1. Introduction

The UNIX login program asks the user to type his name and password, which consists of a character string. The password can be observed by an observer (for instance "over the shoulder observer") and detected by a hacker (someone who attempts access by the "trial and error method"). Passwords should for security reasons be:

- be long (as many characters as possible)
- have no meaning (no well known names of persons, countries, firms etc., first names, birthdates, house numbers, street names, city names, words as "test", "demo" etc. and also these words spelled backward)
- be changed regularly (for instance once a month; but not too often, since the password can be easily forgotten or written down or because of the frequent change, become too easy.
- not be written down anywhere and only stored in the user's brain and under protection in the UNIX system
- be entered into the terminal (keyboard) quickly

Many people do not accept these security regulations, because of lazyness or lack knowledge. They prefer short passwords (e.g. four characters) and passwords which have special meanings, which in addition are changed only rarely. Short passwords, but especially passwords with special meanings, can be found out by trial and error method or by looking over someones shoulder while he is putting the password into the terminal (keyboard). One way out of this dilemma is the use of special algorithms. It permits the use of a new password at each new "login". It is simpler and more secure, to remember a "password algorithm" with, for instance a simple secret word, than to try to remember a long password without any meaning and to change this regularly. The result of the use of password algorithms is either one-time passwords or very long "relative static" passwords, which the user remembers through a special technique (for instance reconstructed passwords or word combinations with veiling). Such algorithms are presented in [Has84a] and [Pil86a].

In this paper a new algorithm is introduced with which passwords can be secured and which makes it practically impossible for hackers, observers and "eavesdroppers" to break the password. With this algorithm, from now on called LEMA-method (lettermatrix algorithm), a new lettermatrix will be displayed on the display of the user at each login. The user has to find his way through the lettermatrix, based on his secret word (usual secret password), and he will find during and at the end of these steps, his actual password. Since the lettermatrix is changed at each login, the user constantly receives a new

password. An observer and an eavesdropper cannot find out the current secret word then, and a hacker also fails because of the many possibilities which exist, since every password represents a random character string.

2. Verbal description of the algorithm (LEMA-method)

The user takes note of the usual secret password $G = g_1 g_2 \ldots g_h$, and 3 numbers (n, r_1, r_2), which are described more fully below. The user only ever gives the UNIX system this secret word and the 3 numbers whenever a change is made. Every change should be especially protected (without observers and, if possible, encrypted).

When the password is requested (login phase), a lettermatrix with 24 rows and 24 columns will be displayed on the screen. Each of these 24 rows and columns contains all 24 letters of the alphabet from "A" to "X", i.e. individual letters appear only once in a row or column. The user now searches for his password by going through the matrix. This way has h positions (changes of direction) resulting from the letters g_1 to g_h. Two of these h positions are designated end positions, and the letters following these two positions give the password.

The user starts searching for his password by looking for the first letter of his secret word (g_1) in the first row of the matrix. The position of g_1 in the first row indicates the first position, especially the column, in which the second letter of the secret word (g_2) is to be found. Once the second position has been found, the third letter is searched for in the same row etc. until the last letter (g_h) in the lettermatrix has been found. The second end position is given by the position of the last letter (g_h). The first end position corresponds to the n^{th} position (based upon the letter g_n) with $1 \leq n < h$. The letters following both end positions make up the searched for password, whereby the user chooses himself how many letters, after the first and second end position, will be selected (variable r_1 and r_2). The sum of $r_1 + r_2$ is equal to the length of the password. The values of n, r_1 and r_2 are chosen loosely by the user. If the user, for example, chooses a four letter password in the form of two pairs of letters ($\rightarrow r_1 = 2$ and $r_2 = 2$), then the 2 letters immediately following the first and second end position create the password.

3. An Example

The user chooses as his secret word the letters "KRILIR" and the values $n = 5$, $r_1 = 2$ and $r_2 = 2$ and then gives this data to the UNIX system. This secret word and the values of n, r_1 and r_2 are valid until the next change is made. The UNIX system generates randomly a combination of letters to create a lettermatrix which is then displayed on the screen of the user. For example the matrix in Figure 1 results in the password "QSBD".

U	X	W	G	I	V	K	N	P	E	O	H	B	R	T	A	L	F	Q	M	S	C	D	J
I	L	K	S	U	J	W	B	D	Q	C	T	N	F	H	M	X	R	E	A	G	O	P	V
G	J	I	Q	S	H	U	X	B	O	A	R	L	D	F	K	V	P	C	W	E	M	N	T
W	B	A	I	K	X	M	P	R	G	Q	J	D	T	V	C	N	H	S	O	U	E	F	L
D	G	F	N	P	E	R	U	W	L	V	O	I	A	C	H	S	M	X	T	B	J	K	Q
A	D	C	K	M	B	O	R	T	I	S	L	F	V	X	E	P	J	U	Q	W	G	H	N
V	A	X	H	J	W	L	O	Q	F	P	I	C	S	U	B	M	G	R	N	T	D	E	K
T	W	V	F	H	U	J	M	O	D	N	G	A	Q	S	X	K	E	P	L	R	B	C	I
S	V	U	E	G	T	I	L	N	C	M	F	X	P	R	W	J	D	O	K	Q	A	B	H
C	F	E	M	O	D	Q	T	V	K	U	N	H	X	B	G	R	L	W	S	A	I	J	P
M	P	O	W	A	N	C	F	H	U	G	X	R	J	L	Q	D	V	I	E	K	S	T	B
R	U	T	D	F	S	H	K	M	B	L	E	W	O	Q	V	I	C	N	J	P	X	A	G
X	C	B	J	L	A	N	Q	S	H	R	K	E	U	W	D	O	I	T	P	V	F	G	M
F	I	H	P	R	G	T	W	A	N	X	Q	K	C	E	J	U	O	B	V	D	L	M	S
J	M	L	T	V	K	X	C	E	R	D	U	O	G	I	N	A	S	F	B	H	P	Q	W
O	R	Q	A	C	P	E	H	J	W	I	B	T	L	N	S	F	X	K	G	M	U	V	D
K	N	M	U	W	L	A	D	F	S	E	V	P	H	J	O	B	T	G	C	I	Q	R	X
H	K	J	R	T	I	V	A	C	P	B	S	M	E	G	L	W	Q	D	X	F	N	O	U
L	O	N	V	X	M	B	E	G	T	F	W	Q	I	K	P	C	U	H	D	J	R	S	A
P	S	R	B	D	Q	F	I	K	X	J	C	U	M	O	T	G	A	L	H	N	V	W	E
Q	T	S	C	E	R	G	J	L	A	K	D	V	N	P	U	H	B	M	I	O	W	X	F
N	Q	P	X	B	O	D	G	I	V	H	A	S	K	M	R	E	W	J	F	L	T	U	C
E	H	G	O	Q	F	S	V	X	M	W	P	J	B	D	I	T	N	A	U	C	K	L	R
B	E	D	L	N	C	P	S	U	J	T	M	G	W	A	F	Q	K	V	R	X	H	I	O

Figure 1: *Example lettermatrix*

The next time a password is requested, another lettermatrix is displayed on the screen and therefore by using the same secret word another password results.

4. Description of the algorithm

Lettermatrix K = $k_{1,1}$ .... $k_{24,24}$ with $k_{i,j} \in$ {"A"..."X"} and $1 \le i,j \le 24$

Secret word G = g_1 .... g_h with $g_i \in$ {"A"..."X"} and $1 \le i \le h$

Length of the password q = $r_1 + r_2$ with $q > 3$

First end position n with $1 \le n < h$

Password P = $p_1 p_2$ .... p_q = Algorithm1(G, K, n, r_1, r_2)

```
Algorithm1:
    i := 1; j := 1; v := 1;
    while v≤h do
    begin
      j := 1;
      while k_{i,j}≠g_v do j := j + 1;
      if v = n then
      begin
        s := i;
        t := j
      end;
      v := v + 1;
      if v ≤h then
      begin
        i := 1;
        while k_{i,j}≠g_v do i := i + 1;
        if v = n then
        begin
          s := i;
          t := j
        end
      end;
      v := v + 1
    end;
    for u := 1 to r_1 do p_u :=k_{s,1+(t+u-1 mod 24)};
    for u := 1 to r_2 do p_{u+r_1}:=k_{i,1+(j+u-1 mod 24)};
```

5. Notes

There are no letters "Y" or "Z" in the lettermatrix. Therefore these letters cannot appear in the secret word or else the result will be that all "Y" will be replaced by "I" and all "Z" by "C". Also, only capital letters appear in the matrix.

The screen must be capable of displaying 24 rows and 80 columns. The lettermatrix is made up of 24 rows of letters and 49 columns (24 letters and 25 vertical dividing lines). The dividing lines (e.g. | or blanks) serve to improve the ease of reading the matrix. No (horizontal) dividing line is necessary between the rows. Beside the matrix, 33 columns per row remain free for text such as prompting for a password, error reports, comments etc. The lettermatrix disappears from the screen as soon as the password is given (e.g. after the last letter of the password has been entered).

In a nonstandard version the lettermatrix can consist of more, or less than, 24 letters and also with numbers. A numerical matrix requires only a 10 row-screen and a single number as secret word and password. With a 10×10 numerical matrix, a "hacker" can choose from 10,000 different possibilities (with a 24×24 lettermatrix, there are more than 300,000 possibilities). The number of different secret words, paths through the matrix etc. is also less, in other words, the chances of success for a "hacker" is much higher than with a 24×24 lettermatrix.

6. Algorithms for calculating the lettermatrix

The lettermatrix can either be completely calculated in the UNIX system, or, a part of it in the UNIX system and the rest in the terminal. The former method has the disadvantage that when the lettermatrix and password string are transmitted along the line (between UNIX system and terminal) in a cleartext (decrypted) form, an electronic eavesdropper tapping the lines could eventually find out (calculate) the secret word from these two pieces of information. In order to do this, however, he would also require more logins (lettermatrices including password) of the same user and a lot of computing time and memory. A much safer method would be a twin or shared generation of the lettermatrix, i.e. the UNIX system generates one part and the terminal the rest. The password and that part of the matrix created by the UNIX system, can then be transmitted decrypted without an electronic eavesdropper being able to use the information. This kind of divided generation is only possible with an "intelligent" terminal.

6.1. Algorithm for the complete generation of the lettermatrix in the UNIX system

Starting with two vectors, each with 24 elements, the lettermatrix is made up from these two vectors through addition of their elements. A function called "random" serves as an aid in this case by producing random numbers from the range of the natural numbers {1,2,3,....}.

Algorithm2:

$letter_0$:= "A"; $letter_1$:= "B"; $letter_2$:= "C"; ..... $letter_{23}$:= "X";
for i := 0 to 23 do
begin
  row_i := i; $column_i$:= i
end;
for i := 0 to 23 do
begin
  j := random modulo 24;
  v := row_j ; row_j := row_i ; row_i := v;
  j := random modulo 24;
  v := $column_j$; $column_j$:= $column_i$; $column_i$:= v
end;
for i := 1 to 24 do
for j := 1 to 24 do
begin
  v := (row_{i-1}+$column_{j-1}$) modulo 24;
  $k_{i,j}$:= $letter_v$
end;

This algorithm allows to calculate 24!* 23! (i.e. appr. 10^{46}) different lettermatrices.

6.2. Algorithm for the two-part generation of the lettermatrix: UNIX system part

The UNIX system calculates a vector (called "comp") based on function random and sends this vector to the terminal. With this algorithm 24! (i.e. approximately 10^{23}) different vectors can be calculated.

Algorithm3:

for i := 0 to 23 do $comp_i$ = i;
for i := 0 to 23 do
begin
  j := random modulo 24;
  v := $comp_j$;
  $comp_j$:= $comp_i$;
  $comp_i$:= v
end;

6.3. Algorithm for the two-part generation of the lettermatrix: Terminal part

In each terminal there is a vector with 24 elements secretly stored, which in the following algorithm is called "term". In this vector, each element has a value between 0 and 23 and each value appears only once. This vector is only known to the terminal and the UNIX system and should be different in every terminal connected to the UNIX system. The lettermatrix is created by the addition of the vector (called "comp") transmitted from the UNIX system and the vector (called "term") stored in the terminal. In doing so, the basic rule that in every lettermatrix, each letter appears only once in each column and row, is adhered to automatically.

Algorithm4:

$letter_0$:= "A"; $letter_1$:= "B"; $letter_2$:= "C"; ..... $letter_{23}$:= "X";
for i := 1 to 24 do
for j := 1 to 24 do
begin
  v := ($comp_{i-1}$+$term_{j-1}$) modulo 24;
  $k_{i,j}$:=$letter_v$
end;

With this algorithm 24!* 23! (i.e. approximately 10 sup 46) different lettermatrices can be calculated.

7. Acceptance

The LEMA-method requires more work to be done at the login. Since any "unnecessary" extra effort is always regarded as being inconvenient, the user should be briefly informed as to the dangers of using traditional (usual secret) password systems and the advantages of the LEMA method. Additionally, if the user no longer needs to change his secret word (i.e. his traditional password) so often (e.g. annually instead of monthly) then, as a rule, no acceptability problems should arise. Some users have problems in using the LEMA method, in that they cannot always find their way through the matrix. You should avoid using two or more identical letters following one another in your secret word and choose $n=h-1$ (i.e. the two end positions immediately follow one another). Generally it can be taken for granted that a considerably higher degree of security should be more important for certain users than the fact that the login is somewhat more complicated.

Acknowledgments

The author thanks Mr. E. Eder and Mr. T. Gmach for several valuable suggestions.

References

[Has84a] J. A. Haskett, "Pass-Algorithms: A User-Validation Scheme Based on Knowledge of Secret Algorithms," *CACM*, vol. 27, pp. 777–781, 1984.

[Pil86a] E. Piller and A. Weissenbrunner, *Software-Schutz. Reihe Angewandte Informatik*, Springer Verlag, Wien, New York, 1986.

User Experience with Security in a Wide-area TCP/IP Environment

Peder Chr. Nørgaard

Computer Science Department,
Aarhus University,
Denmark
pcnorgaard@daimi.dk

ABSTRACT

EUUG is currently considering establishing a TCP/IP based network on leased lines between its European members, similar to the North American Internet.

We know from stories from the USA that the Internet is breeding ground for things like the Internet Worm of November 1988. This gives natural basis for concerns about the security problems in the wide-area TCP/IP environment.

In the Scandinavian countries the NORDUnet has been operational since January 1989. NORDUnet connects most Nordic universities offering several services, among those TCP/IP. The NORDUnet is connected to the John v. Neumann center in USA through a 64 Kb transatlantic line, effectively integrating it into the Internet.

I will here report on those considerations of security that we have been focusing on since it became evident that any computer science student of USA and Canada can connect to any TCP/IP-based service on all our machines.

1. Introduction

First, let me elaborate a bit on the title of the paper. The word "user" does not mean ordinary user. I am system manager at a Computer Science Department, and I suppose that most of the readers will have a similar interest in system managing. But on the other hand, I am not managing the wide-area network itself, and I am no security expert, so in that sense I am a "user".

Experience should perhaps have been Considerations. To my big disappointment the most alarming security experience I have had was a single student at the neighbouring university who used an account with no password to browse our system. So in this sense I have not experienced a lot of security problems.

Wide-area TCP/IP environment means exactly that: the computer on my desk sits directly on a class B TCP/IP network together with several hundred computers on the five Danish universities, and this class B network is connected via a gateway to the entire Internet.

We enjoy "being on the Internet". We pay a fixed yearly sum for which we get fast and verified e-mail with no amount charge. We can pick up all kind of interesting things via anonymous ftp, we can "talk", and run X-window applications against computers everywhere. And it is only the system managing staff that sometimes worry a bit about the potential security problems.

1.1. Security Goals

To find out, what the topic really is, let me present the two different purposes that we try to reach when we tighten up the network security.

Firstly, we do not want anyone from outside the department to do anything to our computers, except for a short list of well-defined operations that we define as "public". Among these are: sending mail to us, fingering us, and logging in to shells after giving legal password/username combination.

Secondly, we do not want to compromise our access to the wide-area network by having any of our users trying to break into other systems. I figure that this consideration is perhaps only of relevance to Computer Science Departments like ours; employees at private companies generally have work to do and no time for hacking, and students at other departments may not have the necessary knowhow. But we certainly have to keep an eye upon the activities of our more competent students.

Having noted this I must admit that our department is not very security conscious. I study security for personal interest, and because being sure that nobody fools around in our systems eases the system managing job. But we don't keep sensitive information on our computers at all, so I won't lose my job if some bright US student some day gets root access on our system. At least, not if I can get him out again in a few days...

1.2. Network Layers and Security

The architecture of networks is layered, and I have chosen to structure the paper after those layers. The lowest layer is the *data link layer*, where packages are sent on local physical nets like Ethernet or token ring.

The next layer is the *network layer*; this layer is independent of network hardware, and takes care of routing packages across gateways.

The upper layer is the *transport layer*. This is the layer where a user process on a UNIX system can open connections and communicate. The two lower layers are hidden in the kernel, and only the superuser can access them.

1.3. Structure of the Paper

The most important class of attacks on a UNIX system is the attacks on the transport layer. This is because any user anywhere on the Internet with access to the protocol specification from manuals and RFCs† can perform that kind of attack from a user level proces. Section 2 is about the attacks on the transport layer

The more clever attacks where the intruder tries to trick the networking software itself on the the lower layers is described in section 3.

In section 4 I describe the miscalleneous measures we have taken to reduce the possibility of attack success, and to improve the possibility of attack exposure.

Section 5 contains the conclusions.

1.4. What the Paper is not about

I will not write very much about general UNIX system security; rather I will assume that the reader knows a lot about this topic. As a matter of fact, the most important thing at all in networking security is having your users use passwords hard to guess, keeping "/" and "/etc" well protected, and other similar and well known security measures. However, much of what is said is also relevant to system security on local TCP/IP environments.

Neither will I suggest improvements that can only be done by recoding the network drivers and standard application programs, using source code for the system. A lot of improvement of network security can of course be done this way, but this is not a task that in my opinion should be done by system managers; it is a task for the software vendors.

The paper is written in the middle of a process. I just started on networking security in January 1989, so this is not a scientific paper describing a completed project, but rather an intermediate report with food for thought, suggestions and ideas, and several loose ends.

My personal experience is almost exclusively with Sun computers running different versions of SunOS, so the paper is somewhat biased against Sun systems; but the principles of networking security is the same for all UNIX brands.

† RFC means "Request for Comments" and is the common title of the now very long (more than 1000 issues) set of articles, specifications, and other kinds of papers that defines the Internet. I have never found out why they are called RFC; I believe that the term is historical, for an exact specification like RFC822 (the specification of the Simple Mail Transfer Protocol), do not request comments; it rather says, stick to this in your coding or everything will go wrong.

The RFCs are available from several sources; most EUnet backbones keep them.

1.5. A short Course in TCP/IP Networking

I will not assume a detailed networking knowledge. In this section I define the most important networking terms. If you know nothing about networking, these should help you understand most of the paper.

The terms defined here originate from the RFCs. Readers who know a lot about networking will know that things are not quite as simple as described here; the definitions are for readers with no or little networking knowledge. Readers who know about SunOS will miss any reference to RPC services; this is for brevity, and because use of the RPC level really does not change anything with respect to networking security (except that the RPC portmapper creates a big security hole in itself).

The DoD internetwork protocol suite defines a networking layer protocol, called *IP*, two transport layer protocols, *TCP* and *UDP* and several application specific protocols on top of these (examples are *telnet*, the terminal emulation protocol and *smtp*, the Simple Mail Transfer Protocol. Collectively, these protocols are known as *TCP/IP*. Several terms are used in the TCP/IP environment:

A *host* is the networking term for a computer. It may be anything from a PC to a supercomputer.

A *network* is a group of hosts who can communicate directly. My network is the "Danish Ethernet" which consists of many Ethernet segments all over the country, connected via long-haul lines and so-called filter bridges. In the sense of TCP/IP this is a single net, because the filter bridges hides the fact that the Ethernets are separated. Most specifically, a broadcast packet from one host will reach all other hosts in a few milliseconds.

The *address* of a host is a 32-bit global number, normally written as four decimal numbers (my host is 129.142.16.13). The number is global because it contains the address of the network (129.142) as well as the address of the host within the network (16.13).

The network layer protocol of the TCP/IP protocol suite is *IP*. It is a packet switched, unreliable protocol, easily mapped upon hardware like Ethernets and token rings.

On top of IP two transport layer protocols are defined. *TCP* is the reliable, bidirectional sequence controlled protocol, and *UDP* is the unreliable datagram protocol, with facilities for broadcasting within a single network.

The address of each end of a specific transport level communication is the concatenation of the host address and the *port number* which is an unsigned 16 bit number.

A certain subset of these numbers are registered as *well known* port numbers. These are typically used when establishing new network connections to remote hosts; as an example, TCP port number 25 is the well known port number of the Simple Mail Transfer Protocol, which carries all the electronic mail of the wide-area TCP/IP network. The port numbers not well known are available for dynamic allocation by application programs.

Typically, a communication language is defined together with the well known port number. The SMTP, for instance, defines that the initiator of the smtp connection must speak ten different four-letter commands with specific arguments, and the server must answer with three-digit state numbers and arbitrary text.

The port numbers below 1024 are said to be *priviledged* port numbers – it is UNIX convention that only root processes can allocate them, so if another UNIX machine connects to you, and the source port number is below 1024 you can be certain that the communicating process runs as root; on the other hand on an IBM PC you can allocate any port number, and you can do the same thing on a UNIX system in single user mode, so the definition is not safe.

Most TCP networking (mail transfer is a good example) works so, that a process (called the *daemon* process) sets itself to wait for outside connections on one or more well known port numbers. When connection is made, the daemon process forks itself; the new process then executes a communication session, typically by exchanging messages via the bidirectional connection in some language with a well-defined syntax, while the original process goes back to the listen state, ready to handle new service requests.

Most UDP-based services also works with a server that listens on a well known port number. The main difference is, that forking is seldomly done because the interaction is brief, one request datagram and one answer (as in Yellow Pages requests) or even an information datagram which calls for no answer (example is a syslog entry).

Another relevant way of using UDP is the broadcasting for information about hosts which offer a specific service. The SunOS-specific process *ypbind* broadcasts a request for a Yellow Pages server in a specific domain; any host on the local network which sees the request can answer this request.

Note that apart from the concept of priviledged port numbers, the TCP/IP protocol suite do not in any way map to "normal" UNIX security concepts, like user and group ids. If someone can connect to a port number on a host, everybody can do so, and it is not possible to have the networking software verify in any way, who the user in the other end of the connection is.

2. Attack at the transport layer

Attack on system security via regular TCP or UDP calls is by far the most dangerous. Most systems are wide, wide open to a whole set of attacks via this channel, when they are put on a network initialized as suggested by the software vendor, System management have to study the system carefully, read a lot of manuals and do a lot of guessing just to close the most evident holes. The famous Internet Worm worked, as everybody probably knows today, through holes created by careless programming in two regular servers, the *sendmail* and *finger* daemons, while exploiting holes left by careless users using *telnet* and *rsh* services.

2.1. Attack scenario

Any attack will consist of two phases. The first thing the intruder wants is to somehow get into the system in order to be able to change information. This is the easiest phase to study; there are a final number of ways of getting into the system.

The next phase is worse. Any intruder's second task is to ensure that he can enter the system again, even if his original entry point is closed. Therefore he will change *something* in the system, creating new security holes.

Thus, when we want to make it more difficult to break into our systems, we must not only close entry holes; it is also very important to know how new holes can be created. This is why the most important facet of our security program is the supervising of those files that defines the entries to the system.

2.2. Establishing an overview

It is a pretty big task in itself to establish an overview of the possible ways of connecting to a system from outside, and trying to figure out if it can be done by intruders.

There are two sources of information relevant for the job. You can study system configuration files like /etc/rc.local, /etc/inetd.conf, /etc/services and the like, to figure out which services are established during system initialization. Then you can read the relevant manual pages to see how these services are configured, especially how access to them are restricted. Later I will go through a list of examples of this.

Unfortunately this is not sufficient. Programs, both from the vendor who delivers the basic system, and third party software vendors, as well as programs you write yourself, have a way of opening network connections without stating so in the manuals. Did you know, for instance, that any SunView window opens a TCP connection and listens on it? I did not, before I began to study networking security. I still do not know why, although I suppose it has something to do with selection service. It is *probably* harmless, but how can I be sure of that?

So you also have to check which connections are actually open. The BSD based systems has a program called "netstat" which is of some use: it gives a complete list of all open network connections. Unfortunately it does not tell which processes holds the connections open. A little program, "fstat", of the BSD 4.3 Tahoe release (published in USENET group comp.sources.unix as v18i107) helps you to obtain a complete list of all processes listening on network connections.

2.3. The well known Entry Points

I will now go through a list of network server daemons that can be found from the source files mentioned above. I will try to group them according to their relevance for networking security.

2.3.1. Those with Real Security Holes in them

These servers have all some kind of bug that was exposed by the Internet Worm or discovered (or perhaps just became wide known) in the turmoil that followed it. Other servers probably have similar holes, they are just not discovered yet. These servers are fingerd, sendmail, ftpd, ypbind, yppasswdd, walld.

Sun has fixes for them all; for sendmail I would suggest that you get the public version 5.61 instead – your Eunet backbone has it. Apart from being an improvement of earlier versions of sendmail, the BSD people have given it a security overhaul.

2.3.2. The virtual terminal services

The *telnetd* is just as safe as your general UNIX user setup, as it demands user/password combination before allowing any login. The only problem I know of is described in section 3: that you have to send the password in clear text along the network where snoopers may be listening.

The *rlogind*, the remote login daemon is a bit less safe than telnet because you can allow remote logins with no passwords from other systems, by specifying the files /etc/hosts.equiv and .rhosts file in the user home directories. This has also advantages, amongst those that you *don't* need to send passwords in clear text along the network, and that the accompanying services, remote shell and remote copy, can be done across the network, preserving user idenfication.

The /etc/hosts.equiv and all the users' .rhosts files must however, be supervised. An intruder might very well like to keep his access to the system by manipulating those files. And the standard setup is to have a "+" in /etc/hosts.equiv, effectively trusting *every* host on the entire Internet. Change that, please!

2.3.3. nfsd, the Network File System Daemon

This is one of those where you just have to do something. Sun's standard system installation leaves the /etc/exports files exporting all the file systems, even the root partition with no access limitation at all. This means that any machine on the entire Internet can mount all your file systems read/write; the only limitation is that files owned by user root cannot be accessed beyond their own access mode.

The cure is simple: at the very minimum, define a netgroup holding your machines and put it into an access specification in the exports file. Also remove the exporting of "/"; I don't know why it is there.

Incidentally, remember that a user on a PC is a superuser; if you export a partition to a PC running PC/NFS you must accept that you have exported the *entire* partition, except those files owned by root. The PC/NFS system do a certain effort to check user id and demand password, but this is just a politeness; the user of PC/NFS can change the verified user id at will.

2.3.4. The Yellow Pages servers

The Yellow Pages system from Sun Microsystems maintains a set of consistent network-wide databases, the most important being the password, group, and hosts database. The YP system is very flexible and efficient and quite stable. Even after correction of the serious errors acknowledged by Sun as mentioned above it is unfortunately still *very* easy to trick. As an example, any host on the local network can claim that it is Yellow Pages server. There is really nothing you can do about it, except for the suggestions I make in subsection 4.6 below.

3. Attack on the network and data link layer

In order to work on the network layer, an intruder needs superuser access on a UNIX system or access to a personal computer, like an IBM PC or a Macintosh with network hardware. This is because the network layer is not accessible from a non-superuser process on a UNIX system.

Attacks on the network layer is not very interesting in itself. The only extra action that is possible on this layer is faking you own address; in this way you could hide your tracks when trying to break into systems, but no more. You cannot even fake the network part of your address, because routing in the IP network is done directly on the network address.

Many more options is present for an intruder who works on the data link layer.

The first option is simply information sampling by listening on the network and reading information not meant for him. It is, for instance, not very difficult to discover when a new telnet or rlogin connection is established, and trace the first score of packages, thereby picking up the password of the user making the connection.

All the other options involve some kind of faking. There are no definite limits to the amount of faking that can be done; the only limits are practical, that is, how much work will the intruder spend on faking.

The easy thing is to fake answers on UDP-based requests. Requests for Yellow Pages information is a good example. I do not know whether a Yellow Pages requester checks that the answers come from the hosts that is asks (it certainly ought to), but this does not matter when the intruder works on the data link layer. From this layer you can also fake the source address of you package, thus completely cheating the requester.

The more complicated thing is to fake multi-package interactions, like TCP connections and NFS file transfers. I don't think this can be done while the host that you are faking is active; but I think it could be possible to confuse a host enough with false packages, perhaps even to bring it down, so that it will be silent while you do some faking.

4. Security Measures – what can be done to improve Network Security

I will now proceed to list the measures taken (or intended to be taken in near future) at our department, in order to improve the network security.

4.1. Make a Single Host Hard to Get At

This is, in my opinion, one of the strongest initiatives. We let one of the hosts run as few services as possible and make it unaccesible to anyone but a few staff members. We then proceed and make it Yellow Pages master server, syslog host and other important functions. The host does not have to be very powerful; in our case we chose a Sun 3/50 with a local disk.

Several of the other security measures are based on the assumption that this one host is invulnerable.

4.2. Supervise Logins

Irrespectively of all the security holes described above, the most frequent attack point is still attack via *telnet* or *rlogin* by an intruder who guesses, steals or breaks the password of our users. Furthermore, an intruder who enters the system by some other means will very soon do a login; the shell is still the most versatile tool for finding out things about a system.

So we keep an (automated) eye upon all logins as they are being recorded in the /usr/adm/wtmp files on the individual hosts. Every few minutes the additions to the /usr/adm/wtmp is transmitted to the master host where they are checked for strange entries. We record for each user which non-local hosts he uses to login from. For most users, this list is empty, and when a new username/remote host combination show up (which happens once a week or so), we check manually, that the correct person is behind it.

4.3. Supervise config files

The following files are generated automatically from a database on the master host; at random intervals it is automatically checked that the files are still in place and not changed:

- /etc/exports
- /etc/fstab
- /etc/inetd.conf
- /etc/printcap
- /etc/rc.local
- /etc/syslog.conf
- /etc/ttytab

Furthermore the files /.rhosts, /etc/hosts.equiv, /.cshrc, /.login, /.profile, and /.logout are identical on all hosts and also checked on irregular basis. More files is expected to be added to the list as our project is advancing.

4.4. Do not allow Single-User Boot.

The best tool for breaking into other UNIX systems is undoubtedly a UNIX system where the intruder has root privileges. In order not to tempt our own users by making things too easy for them, the systems are configured so that single user boot (which is easy to do for any user) demands the root password before giving access to the shell. This is a new feature in SunOS 4.0; at SunOS 3.x there was no protection against single user boot.

4.5. Do not allow Network Window Applications as Superuser

It is a general UNIX security advice that you should run as little as possible as superuser. But as network window applications (X-windows and NeWS) listens on well known ports this is more important than for other applications. Most important is NeWS where it is possible to connect to the window server and actually download and run code.

4.6. Spy on the Network

The Network Interface Tap facility of SunOS 4.0 gives a user with root privileges unlimited access to read any packets on the Ethernet. This ability was seen in section 3 as a possible attack point. It can also be used in the service of the good: we can configure a host to spy on the net, thereby trying to discover any irregular behaviour.

There is no end to the amount of verification you can do this way; the theoretical limit is complete simulation of all the hosts on the net. This is of course beyond the capability of any one CPU!

But within the capability of one CPU is for example looking for intruders trying well known holes. I have no practical experience with this kind of supervising yet, but my first plan is to keep a permanent watch for those irregular Yellow Pages packets which would show up if someone was trying to trick our Yellow Pages system. Secondly, I plan to keep an eye on unusual connections from local hosts to external hosts, thereby catching potential intruders among our own users.

4.7. Write your own Server Application to verify the Requester's Address

We have written a few networking applications (an archive system and an account system) and these are written to check that access to the server is only allowed for hosts in a certain netgroup. This is about three lines of extra code in each server program, and these three lines of code ought to be present in many of the standard servers mentioned in section 2!

4.8. If possible, isolate your Local Network via a Gateway.

We plan to do this during to autumn of 1989. Instead of being part of a big class B network covering all Danish universities, our network will be a class C network with only our own computers on it. All traffic will go through a gateway. This will exclude all possible attacks on the data link layer.

4.9. If possible, use C2 level Security

The SunOS 4.0 has a specific feature that makes it possible to run the system on the security level known as C2 (ordinary UNIX systems run on level C1). Other vendors have or plan to have similar features.

While this feature do not stop intruders at all, it is a very useful defense against "phase 2" of an attack (modifying security-relevant files).

5. Conclusions

It has been demonstrated that the fundamental problem with security in TCP/IP networks of UNIX hosts is that the TCP/IP protocol suite does not map well onto the traditional UNIX security concepts. Everytime a process starts listening for requests on a port number, it opens a hole for any process on the whole Internet to connect to. The listening process can choose to verify two facts about a new connection: the host address of the host from which the connection comes, and whether connection is on a priviledged port number.

And worst of all, the UNIX< vendors are not very worried about this (at least, they were not before November 1988). Scores of utility programs using the network are coded with absolutely no concern for security; they do not even perform those checks that are possible. Even security-crucial utilities like Sun's Yellow Pages service could be broken with a little technical knowledge.

One may wonder how it is possible at all to run a wide-area TCP/IP under these circumstances. I think that the reason is that although attacks are not difficult, it is near impossible to avoid exposure of attacks. And the network has means of punishing attacks: single users will have their access to the system barred, and a system manager who will not stop his users will soon see his whole institution shut out. So in practice the security problems are not big.

How to Protect Your Software Through International Copyright Laws: Step-By-Step Instructions

Alicia Dunbar Gronke

USENIX Association
Berkeley, California
epg@sun.com

ABSTRACT

This article will attempt to clarify some common misconceptions about copyright formalities as applied to intellectual properties. Due to the fact that these laws vary from country to country, the examples presented here should be viewed only as basic approaches to copyright protection, not a legal reference.

Taking these few necessary precautions to protect your work from theft or unauthorized alteration isn't nearly as much of a task as one might believe. Here, we set about taking the mystery out of what should be only a few simple procedures. The idea being stressed here is taking as many preventative measures as possible to protect your work. The effort required to do this is minimal.

Introduction

Copyrights, by my definition, were designed with the idea that if a person is willing to share their works and expressions with the interested populace by distributing "copies" of said works, that person has an exclusive "right" to protection under the law from those ever present scavengers that watch for the drop of the proverbial ball, only to profit at that person's expense. That person is also entitled to the right to reap any rewards or benefits which may become evident upon publication. This type of law provides protection for the form an expression takes, not the ideas being expressed.

Step Number One: The Copyright Notice

Let's say that you've just put the finishing touches on your new C++ program. You're on top of the world and you feel like a new parent. That's fine, but what's to deter anyone from stealing your new offspring? A Notice of Copyright. And this is what a notice of copyright entails. First, the copyright symbol, namely ©.

This should be followed by the word "Copyright", date of first publication and your name. This simple step fulfills the minimum formality requirements set by the UCC, the Universal Copyright Convention, as well as all signatory nations of the Berne Convention.

Example:

© Copyright [DATE OF PUBLICATION] [YOUR NAME]

However, this may be your second or perhaps third revision of this work, in which case your statement will look something like this:

© Copyright 1985, 1989 [YOUR NAME]

In the case of multiple authors, the practice of showing the names of individual contributors (as well as the date of publication of their contribution) is often used:

© 1988 Modesia Fleming

© 1986 Elissa Metterhausen

© 1983 Daniel Karrenberg

These notices are not mandatory requirements of the UCC, however, the use of them makes the owner exempt from any other copyright formalities imposed by any other UCC nation. Although it is implied by the copyright notice, it would also be to your advantage to add the fact that this publication is not to be copied in any way, shape or form without the express written permission of the author. There is no such thing as too much insurance. To make it easier for the user to comply with your wishes, you may want to include the name and address of your publisher.

On works containing trade secrets, such as computer program source code, a notice indicating that the work contains non-public proprietary information should also be added.

As for [DATE OF PUBLICATION], the moment you give, sell, rent or lend the work, consider it "published". The exceptions to this rule are if the people using it are your employees or are part of a limited group of individuals of the understanding that further distribution of said work is unacceptable. In such instances, this could be viewed as a "limited publication". To keep a tighter reign, a friendly legend should be added, such as:

> "This copy is for private circulation only and may not be used in any other manner."

Or you may choose a more formal approach:

> "The material within is an unpublished copyrighted work containing trade secrets or otherwise sensitive information of [THE COMPANY]. This copy has been provided on the basis of strict confidentiality and on the express understanding that it may not be reproduced or revealed to any person, in whole or in part, without express written permission from [THE COMPANY], which is the sole owner of copyright and all other rights therein."

To deter any later arguments, your notice of copyright should be affixed in obvious places throughout the work making sure it is not concealed from view upon reasonable examination. For example, in source or object code, the notice should be placed at the beginning as well as the end of all printouts, not forgetting occasional insertions in the program itself on disk or tape. It is preferable for most users that this notice be placed solely at the very beginning or at the top of the menu. This is also acceptable. There are few things more irritating than having a copyright notice on constant display. And don't forget to affix this notice to all packaging as well, on the outside of boxes and on tapes. If a proper copyright notice is not placed on all "publicly distributed" copies of a work, the work with which the copyright is associated may in time enter the "public domain", as an improper copyright is an *invalid* copyright.

Allow me to stress the fact that under the Berne Convention (more on Berne later), the use of the copyright notice on published copies of a work is optional. However, the use of it is still encouraged in the U.S. As an incentive to do so, a new section was added to the U.S. Copyright Act. This section [401(d)] prevents claims of innocent infringement by defendants where notices have been placed on stolen works.

Step Number Two: Registering Your Work

This is the second most vital step toward protecting your work. Although this is no longer mandatory in most countries, it still carries considerable weight in courts of law throughout the world in cases of infringement. As each country has it's own individual approach to intellectual property law, it is *extremely* important that you seek the advice of legal counsel before making any final decisions. One advantage of this step is that your work, should it be distributed without notice of copyright, (heaven forbid!), will still have some form of protection. Your registry notice is proof that you took precautionary measures to insure against this sort of thing. Another plus: Bringing suit to enforce your copyright sans registration makes the job just that more difficult.

For the sake of argument, say a writer of a non UCC country, say Upper Volta, first publishes their work in Japan, a UCC nation. That work will be automatically protected as in any other UCC nation.

What is the UCC?

The UCC, otherwise known as the Universal Copyright Convention, first came into force 16 September, 1955. The two other major copyright conventions are the Berne Convention and the Buenos Aires Convention. Of the three, the UCC is the most widely adopted treaty. It is the "umbrella" providing protection known as the "national treatment" doctrine. By employing this doctrine, each member country is obliged to grant the same protection to the works of other member countries as works first published within its own boundaries, as long as that protection meets the minimum requirements of the UCC. This rule applies to both published and unpublished works. So far, it sounds like the panacea for all those "copyright protection blues". Not true. The problem with this is the fact that there are still many countries not belonging to the UCC, and even for the ones who do, there are limitations to the protective requirements imposed beyond national treatment. The minimum standards set by the UCC aren't nearly as comprehensive as the alternatives.

Note:

Depending on the country, you may have trouble protecting object code, since some countries still view it as an adaptation or derivative of source code.

What is the Berne Convention?

The Berne Convention for the Protection of Literary and Artistic works is the oldest and most respected of international treaties. It was concluded at Berne, Switzerland in 1886 and is organized as a Union which is open to all countries of the world provided these minimum protective requirements are met:

- the granting of national treatment,
- the granting of certain "moral rights/droit morale" to authors with regard to the exploitation of their works,
- the granting of certain "economic rights", such as the exclusive rights of translation, reproduction, performance, adaptation, arrangement or alteration, regarding protected works, and
- the adoption of certain minimum terms of protection, generally the life of the author plus 50 years, for various works.

In essence, Berne provides a substantially higher level of protection. For example, copies accidentally distributed without copyright notice are given protection under Berne. This is not the case in most UCC countries. Should any conflict in protection arise, an author is assured to receive the most favourable protection offered under either treaty. Both conventions stipulate that any disputes between member States concerning the interpretation or application of the Convention that has not been settled via negotiation should be referred to the International Court of Justice.

Here is a quote from Dr. Henry Olsson, Director of Copyright and Public Information Department, 21 November 1986 at an international conference on copyright in s'Gravenhage, The Netherlands, speaking on the Berne Conference:

> "The Assembly of the Berne Union in September of 1986 declared *inter alia* that copyright is based on human rights and that authors, as creators of beauty, entertainment and learning, deserve that their rights in their creations be recognized and effectively protected both in their own country and in all other countries of the world."

What is the Buenos Aires Convention?

This treaty provides copyright protection for sixteen South and Central American countries as well as the United States. According to Article 3 of the Buenos Aires Convention, no formalities have to be observed in any country other than the country of origin, *provided* a statement appears in the work that indicates the reservation of the property right. The phrase All Rights Reserved or it's equivalent (e.g., Derechos Reservados) are commonly used for this purpose. The three nations of this convention choosing not to sign the UCC treaty are Uruguay, Hondura and Bolivia. To gain equal protection in these countries, the formalities of the Buenos Aires Convention require your notice to look something like this:

© 1989 Jaap Akkerhuis All Rights Reserved

Between contracting states which are also signatories of the UCC, the notice provision, as well as any other conflicting provisions, are superseded by those of the UCC. When distributing software in Latin American countries, take into account the technology transfer legislations which can limit the years during which protection can be provided in contracts. An excellent reference on this subject is *Trade Secrets and Know-how Throughout the World, (1981)* by Aaron N. Wise.

Below are lists containing the names of independent nations which have signed the UCC treaty, the Berne Convention, the Buenos Aires Convention, as well as those without any major treaties. These Conventions are voluntary agreements made by the governments of these signatory countries. (As only three of the sixteen member countries of the Buenos Aire Convention are UCC signatories, time will be spent rather on the two larger Conventions.)

Countries signing the Universal Copyright Convention:

Algeria, Andorra, Argentina, Australia, Austria, The Bahamas, Bangladesh, Belgium, Brazil, Bulgaria, Cameroon, Canada, Chile, Colombia, Costa Rica, Cuba, Czechoslovakia, Democratic Kampuchea, Denmark, Ecuador, El Salvador, Fiji, Finland, France, German Democratic Republic, Federal Republic of Germany, Ghana, Greece, Guatemala, Haiti, Holy See, Hungary, Iceland, India, Ireland, Israel, Italy, Japan, Kenya, Laos, Lebanon, Liberia, Liechtenstein, Luxembourg, Madagascar, Mali, Malta, Mauritania, Mexico, Monaco, Morocco, The Netherlands, New Zealand, Norway, Pakistan, Peru, Poland, Portugal, Romania, Senegal, Spain, Sweden, Switzerland, Trinidad & Tobago, Tunisia, United Kingdom, United States, Venezuela, and Yugoslavia.

Countries signing the Berne Convention:
(as of November 1, 1988)

Argentina, Australia, Austria, Bahamas, Barbados, Belgium, Benin, Brazil, Bulgaria, Burkina Faso, Cameroon, Canada, Central African Republic, Chad, Chile, Philippines, Colombia, Congo, Costa Rica, Côte d'Ivoire, Cyprus, Czechoslovakia, Denmark, Egypt, Fiji, Finland, France, Gabon, German Democratic Republic, Federal Republic of Germany, Greece, Guinea, Holy See, Hungary, Iceland, India, Ireland, Israel, Italy, Japan, Lebanon, Libya, Liechtenstein, Luxembourg, Madagascar, Mali, Malta, Mauritania, Mexico, Monaco, Morocco, The Netherlands, New Zealand, Niger, Norway, Pakistan, Peru, Poland, Portugal, Romania, Rwanda, Senegal, South Africa, Spain, Sri Lanka, Suriname, Sweden, Switzerland, Thailand, Togo, Trinidad and Tobago, Tunisia, Turkey, United Kingdom, United States, (as of 1 March 1989), Uruguay, Venezuela, Yugoslavia, Zaire, and Zimbabwe.

Countries signing the Buenos Aires Convention:

Argentina, Bolivia, Brazil, Chile, Columbia, Costa Rica, Dominican Republic, Ecuador, Guatemala, Haiti, Honduras, Nicaragua, Panama, Paraguay, Peru, United States, and Uruguay.

Countries Without Conventions:

Afghanistan, Albania, Angola, Antigua, Barbuda, Bahrain, Belau, Belize, Bhutan, Botswana, Burundi, Cape Verde, Comoros, Djibouti, Dominica, Equatorial Guinea, Ethiopia, Gambia, Grenada, Guinea-Bissau, Guyana, Indonesia, Iran, Iraq, Jamaica, Jordan, Kiribati, Korea, Kuwait, Lesotho, Malaysia, Maidives, Myanmmar, Mongolia, Mozambique, Nauru, Nepal, Oman, Papua New Guinea, Phillippines, Qatar, Saint Lucia, Saint Vincent and the Grenadines, San Marino, Sao Tome and Principe, Saudi Arabia, Seychelles, Sierra Leone, Singapore, Soloman Islands, Somalia, Sudan, Swaziland, Syria, Tanzania, Tuvalu, Uganda, Upper Volta, Vanuatu, Western Samoa, Yemen (Aden) and Yeman (San'a).

Protection is automatic anywhere as long as you are a citizen of a UCC or Berne country and the work is unpublished. However, once the work is published, you must adhere to the rules imposed by the member country with which the work is registered.

Austria:

> No provision has been made with respect to the protection of software in Austrian law. A court in Wien confirmed the position that under certain conditions, software is protected under the Austrian Copyright Act. *[Oberlandesgerichts Wien, 8 august 1985, GRUR Int. 793 1987]*

Belgium:

Like Austria, no special provision has been allotted in Belgian copyright law for the protection of software. A bill was introduced in Parliament in June of 1988 for the enactment of a new Copyright Act. The proposal provides that software is protected by copyright with a term of protection lasting 25 years. This Act is not expected to become Law until 1990.

There are currently two laws in Belgium for the protection of copyrightable works. Only programs that are original and express the personal creativity of their authors are eligible for this protection. Belgian law does not protect ideas or opinions, only the form in which they are expressed. Protection begins with creation. The term of copyright spans the life of the author plus a period ending 50 years after January 1st of the following year of her death.

Germany:

Like the rest of Europe, filing in Germany is unnecessary. They have what is called the Copyright Revision Act, which includes "programs for the processing of data", ("Programme für die Datenverarbeitung"), Section 2(1)(1) of the German Copyright Act. The term of protection lasts for the lifetime of the author plus 70 years. A computer program is copyrightable only if:

- it is not just a simple program, i.e., the answer to the problem being solved is not obvious,
- while developing the program, various solutions could freely determine variables,
- the program is not restricted to mechanical-technical continuation and development of generally known subject matter, and
- one can perceive an important, creative and original ability of selection, assembly, reviewing, arranging and classification of information and instructions which surpasses the general average ability.

In the eyes of the German court, only an individual can own a copyright, not a company, which temporarily dissolves the idea of "work-for-hire". The common way around this is for companies is to request the worker to execute a license agreement, giving the company exclusive user and marketing rights to the program.

Sweden:

Computer programs in Sweden are protected under the Swedish Copyright Act which states:

"Catalogues, tables and similar compilations, in which a large number of particulars have been summarized, may not be reproduced without the consent of the producer..."

Future legislation will expressly provide for software protection under copyright law.

The Netherlands:

There are really no formal requirements for software copyright protection and only the outward form of the work is protected. Computer programs in Holland are protected under Dutch law only after meeting the general requirements of originality and perceptibility. Original in the sense that the work be the result of creative activity; Perceptible to the point of the work being perceptible to the senses. Protection is recognized at creation and terminates 50 years from the 1st of January of the year following the death of the author.

Japan:

According to the 1982 decision of the Tokyo District Court, only source code is copyrightable and object code is merely a copy which is also protected under the Japanese Copyright Act. The period of protection is for the life life of the author plus 50 years. Copyright is affirmed the moment of creation. As with other signatory countries, registration is not a prerequisite for filing a lawsuit, although it makes proving infringement a much less painful ordeal. It is presumed that the date of creation corresponds with the date of registration. Your best move, should you be creating something in Japan, is to register with the Commissioner of the Agency for Cultural Affairs in order to establish the name of the copyright owner and the date of publication.

Their work-for-hire applications are quite similar to that of the U.S. In cases of infringement, the damages collected are equal to the profit made by the infringer. On the other hand, damages could equal the possible amount accrued had the author sold the programme.

France:

Seeing as France has no formal copyright office, paperwork here is not a problem, even though software is indeed copyrightable. Acceptable evidence of copyright date, in some cases, can be proven by showing the work was registered in the U.S. A new law introducing the principle that software is an intellectual property protectable by copyright law came into effect January 1st, 1986.

Article 1 of this law expressly confers copyright protection on software. The term is 25 years from the date of the program's creation. France requires the program to be an original work, marked by a personal and intellectual contribution of the author.

United Kingdom:

Section 1(1) of the Copyright Amendment Act confirms that programs are to be treated in the same way as literary works under the 1956 Copyright Act. An Act which states that a copyright work first comes into existence when it is reduced to writing or some other material form. Section 2 of the same Act states that computer programs comply with the material form requirement, even if the programs are not printed or written on paper but simply stored on a computer.

Once again, registration here is not required. So please keep careful records of program creations and changes by date. It is still unclear as to whether or not object code is seen by the United Kingdom as copyrightable. Here the author of the program is the owner of the copyright unless it was composed under a work-for-hire situation, in which case the employer would be the owner. Under a new Act passed in November of 1988, (the Copyright, Designs and Patents Act), the electronic copying of a program constitutes infringement.

United States:

All it takes to obtain a copyright registration is to file a copy of the program with the Copyright Office, along with a two sided form and a $10.00 filing fee. There is only a cursory examination to see that the form is filled out completely, the money paid and the deposit made. Once issued, the copyright is assumed valid and the term of protection is essentially 75 years from the date of publication, or 100 years from the date of creation, or the life of the author plus 50 years, whichever is shorter. For more detailed information, contact:

Register of Copyrights
Copyright Office
Library of Congress
Washington, D.C. 20559
U.S.A.

Registration of a work with the Copyright Office used to be required as a prerequisite to a suit for infringement. Section 411 of the Copyright Act has been changed to a two-part system where American authors are required to register while authors of other countries are not.

Judging by Congress' litigation "track record", they still favour the registration system. Works accompanied by a registration certificate continue to be given prima facie standing when the question of copyright validity is in infringement litigation.

Registration provides the copyright owner with a broader range of infringement remedies. For these reasons, registration within 3 months of first publication is generally advisable. (Known as "timely registration".) You'll be needing Form TX for software. Take note that the U.S. Copyright Office now accepts copies of object code under the "rule of doubt" theory. So, in granting registration, no opinion is expressed as to whether the code actually embodies the registered work. The alternatives listed are available should you feel uncomfortable submitting only object code:

- Submit only the first and last ten pages of source code,
- Submit all of the source code (referring to programs consisting of 25 pages or less) with up to half of it blocked out as necessary to protect trade secrets,
- Submit only the first and last 25 pages of source code with up to half of the code blocked out, making sure to block out just enough code in order to protect trade secrets, or
- Submit the first and last 25 pages of object code with an additional 10 consecutive pages of source code.

What are the differences between Copyright, Trade Secret and Patent Law?

Trade Secret is information. It is a way to "protect something of economic value by keeping it a secret". This something generally provides some sort of an edge, otherwise known as a "competitive advantage". Taking careful measures to keep source code under wraps and away from prying eyes and sticky fingers is of utmost importance. If by chance, in spite of all your precautions your work still falls into the wrong hands, you will have a very strong case against the infringer if you can prove your diligence and persistence in defending your work. Here is a list of steps which may prove helpful:

- Store working copies of source code in a safe, a locking filing cabinet, (one that is difficult to carry away), or a safe deposit box. Remember to keep backup copies in separate places.
- When you're finished working with the code or are stepping away from it for a break, lock it up.
- Stamp ALL copies of the source code with a rubber stamp (preferably in red ink) reading "CONFIDENTIAL". This includes diskettes and their jackets.
- Include a notice, along the same lines of the formal notice shown on page 2, at the top of the menu. You of course can use your own discretion as to how many times you'd like to list this throughout the work.
- When showing the source code, have that party sign a Trade Secret Nondisclosure Agreement form *before* viewing it.

There is a veritable plethora of precautionary measures one can take regarding Trade Secrets. I've named but a few. Most countries appear to allow trade secret information to be transferred via license agreement or contract. There is a possibility that in your particular country the transfer of trade secrets may result in tax consequences. Consult with local counsel.

Patent deals with the right of protection to an inventor for the sole monopolization of use and commercial exploitation of an 'invention'. So far, patent is the most costly particularly in terms of the application process, and it takes up to three years to obtain. Now that we've covered the less attractive aspects, here are the positive points. Should your software qualify for a patent, cannot be used by anyone without your permission. And that holds true for 17 years. Of course there's no telling how long it will take before the software becomes obsolete.

Note:
There is an international agreement that covers patents by the name of the Patent Cooperation Treaty (PCT) which establishes procedures for obtaining uniform patent protection. Their aim is to guarantee national treatment internationally, to enhance the creation of national protective systems and to avoid a far-reaching legal disintegration in this somewhat limited field outside the Berne and UCC conventions. For more information, a booklet called the PCT Applicant's Guide can be obtained from the World Intellectual Property Organization.

WIPO
34, chemin des Colombettes
CH-1211 Geneva 20 SWITZERLAND

Piracy: How to Fight It From the U.S. Vantage Point

An acquaintance brings up in casual conversation, this great new C++ program she just bought for a mere fraction of what you're charging for yours. Upon careful investigation, you find yourself face to face with a slightly modified version of **your** program. Being a victim of piracy is bad enough, but how can you fight him if he's living in Switzerland? (Piracy seems to have paid off handsomely in his case). Besides having a screaming fit and tearing your hair out by the roots, what do you do? You've done everything your lawyer told you to do in the first place. You slathered warnings and copyright notices on the box, on inside labels, in the source code. In the words of Douglas Adams, "Don't Panic".

First, call a *good*, (please note the emphasis on the word good) lawyer. Not all who advertise themselves as "copyright lawyers" know what they're talking about. Try to get a referral from someone whose judgement you trust. Be sure you have all the necessary documentation on hand, i.e., notice of registration or other proof of creation, such as sealed self-addressed envelopes with dated cancellations. Once you've found your attorney, she will clarify the procedure for you and make sure all is in order before moving on to the next step.

This next step entails making sure that your copyright registration is recorded with the U.S. Customs Service, [19 CFR Part 133 Subpart (D)]. This should slow down any imports bearing a suspicious likeness to your work until your case makes it into court and the court can issue an order preventing any furthur importation.

Now you're ready to move into the final stretch. You must seek an order from the International Trade Commission to bar the counterfeit work from entering the country on the grounds that its importation would prove to be an unfair act or would constitute an unfair method of competition [19 USC. Section 1337]. A similar rule applies to trademarks as well.

When filing this complaint, you must be armed with the following information:

- Is this product produced in another country?
- Does it threaten to destroy an existing industry in the U.S.?
- Is the threatened industry efficiently and economically operated?

After the answers to these questions are reviewed and the formal complaint is filed, the case will be heard by an Administrative Law Judge, empowered to grant immediate relief and to ban the imposter work in whole or in part.

Conclusion

Last and most importantly, I must stress the importance of seeking legal counsel on a country-by-country basis to protect your intellectual property. There is no worldwide panacea for the piracy menace as yet. As the needs change, so will the laws, and no one is better qualified to keep you up to date than a lawyer who specializes in the area of Intellectual Properties. Every effort has been made to assure all the facts stated in this paper are current and accurate.

Acknowledgements

The author would like to express her deepest appreciation to all those who have been so giving of their time, patience, expertise and support. Edward Gronke, husband, devil's advocate and friend; Dan Appelman, whose insight proved to be most invaluable, and Michael Kerekes, both friends and lawyers extraordinaire; Peter Salus and Shelly Anderson, true inspirations when it comes to dealing with the human animal; and lastly, the Data Communications group of Sun Microsystems.

Bibliography

Neil Boorstyn. *Copyright Law*. The Lawyers Co-operative, 1981

Brad Bunnin & Peter Beren. *Author Law & Strategies*. Nolo Press, 1983

R.M. Gadbaw & T.J. Richards. *Intellectual Property Rights*. Westview Press, 1988

R. Lee Hagelshaw. *The Computer User's Legal Guide*. Chilton Book Company, 1985

Jozef A. Keustermans & Ingrid M. Arckens. *International Computer Law* Mathew Bender, 1989

Melville B. Nimmer. *Nimmer on Copyright*. Mathew Bender, 1988

Daniel Remer. *Legal Care for Your Software*. Nolo Press, 1984

Robert H. Rines, et al. *Computer Software: A New Proposal*. IDEA: The Journal of Law & Technology, Vol. 29 Franklin Pierce Law Center, 1988

M.J. Salone, Stephen Elias. *How to Copyright Software*. Nolo Press, 1988

William S. Strong. *The Copyright Book*. MIT Press, 1986

UNIX in German speaking countries

Wolfgang Christian Kabelka

Hockegasse 17
A-1180 Vienna

ABSTRACT

With hardware becoming less lucrative the next great battle in computers may be over software. Customers increasingly see versatile software as strategic tool to gain competitive advantages. This paper should help to answer the question if UNIX grants the application coverage in German speaking countries?

1. Who knows about UNIX applications?

Over the past decade, practically every major computer maker has tried to convince the executive in charge of computer systems that its machines were the answer to his every problem. But more and more of these executives still doubt.

They know that committing themselves to a certain brand of computer might make it harder to buy the best product in the future.

UNIX was obscure and had the touch of being a bit hard to use. But it has a big advantage: unlike the proprietary operating systems it is available on many different machines.

Nonproprietary systems such as UNIX put more bargaining power in the buyers hand and consultants predict: "The marketplace is getting a lot smarter".

The industry did not fail to spread a lot of information about "The UNIX Wars" between IBM, Digital Equipment and AT&T and the battles over standards such as OPEN LOOK, MOTIF, PM/X and NEXT STEP. Compared with this there is little information about running UNIX applications.

It is well established that the customers first buying priority is application coverage. Still we are often asked by our clients if we do know about a UNIX solution for a specific application field.

"The role of independent software vendors has become pivotal in the future development of the open systems market" said J. Totman at the 4th UNIX Forum in Vienna. It should be demonstrated that the suppliers for UNIX applications in the German speaking countries play a growing role in the software arena.

2. ISIS – the source of data

The following data is condensed from the ISIS (International Software Information Services) reports.

They are published by Nomina, a company specialized on syndicated catalogue marketing. Nomina, the independent source for information concerning the information industry in German speaking countries, is situated in Munich and cooperates with offices in Basel and Vienna.

The ISIS reports collect data by means of prestructured questionnaires. The reports are updated biannually.

3. Comparison of UNIX with other system solutions

The underlying concept of the survey follows a top down approach beginning with overall statistics on the software market and leads to a thorough analysis of the spectrum of UNIX applications.

Beginning in 1970, a historic documentation shows the market development of standard software (Figure 1). It should be mentioned that the development, measured by the complete number of programs, in the meantime doubles every fifth year.

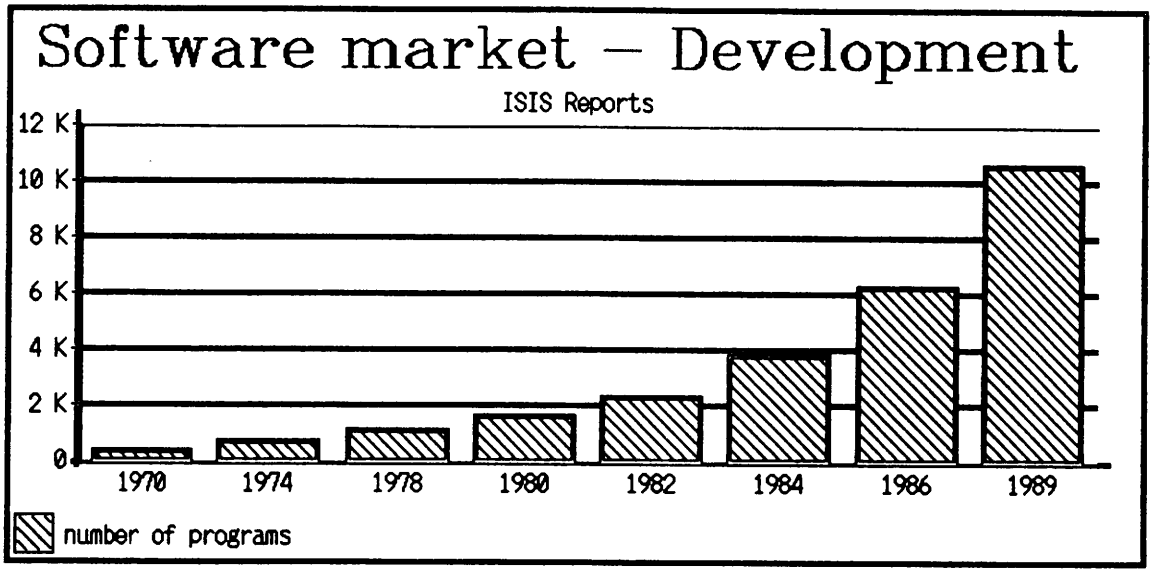


Figure 1

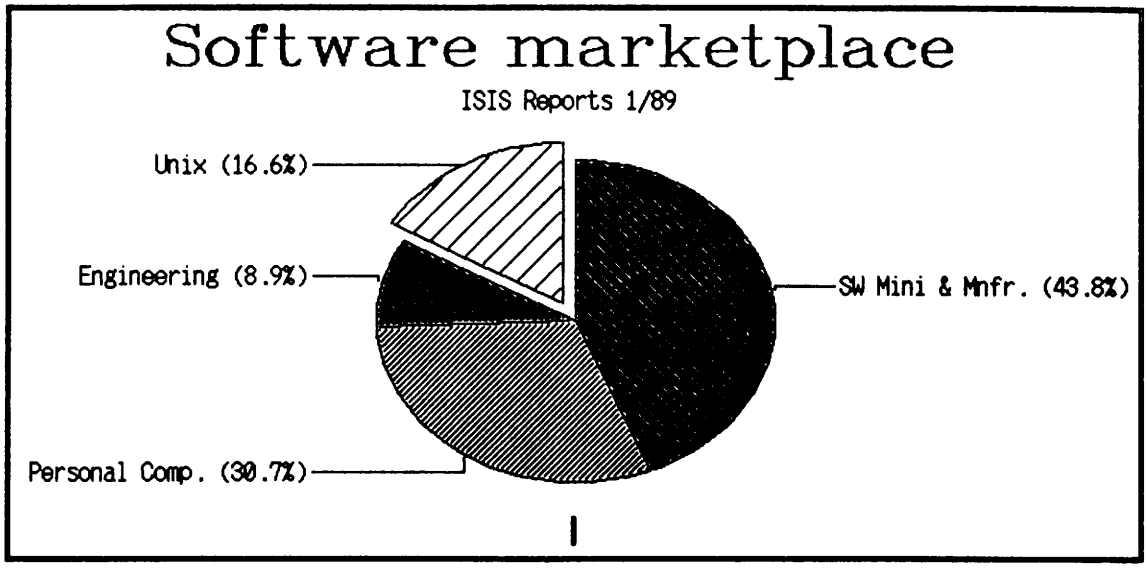


Figure 2

Based on the actual figures, the 1989 segment was split according to operating systems or application fields (Figure 2):

- ISIS UNIX report
- ISIS Personal computer report
- ISIS software report (mini and mainframe)
- ISIS Engineering report

More than 10,000 programs are stored in the current database. Approximately 16% of them relate to UNIX.

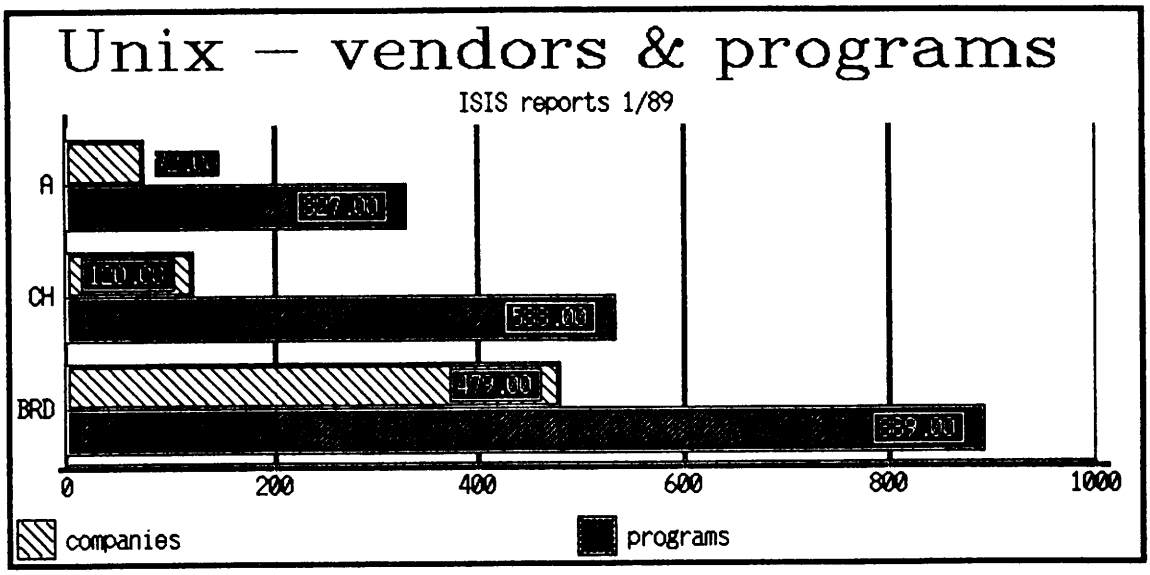


Figure 3

At the moment 685 companies in three German speaking countries – Austria, Federal Republic of Germany and Switzerland – offer programs based on UNIX. On average (Figure 3) each company produces 3 UNIX based applications.

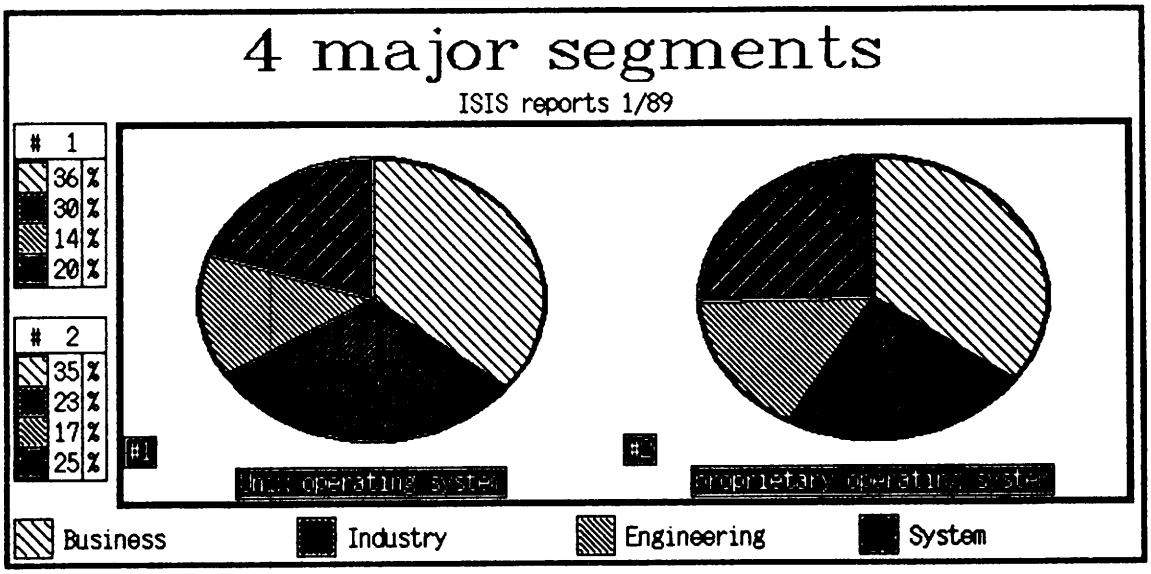


Figure 4

The question was if there is a significant difference (Figure 4) in the application fields between UNIX and applications based on proprietary operating systems. Although direct comparison is not possible relative figures show a very good fit.

The three bar charts (Figures 5, 6, 7) are based on a classification of applications:

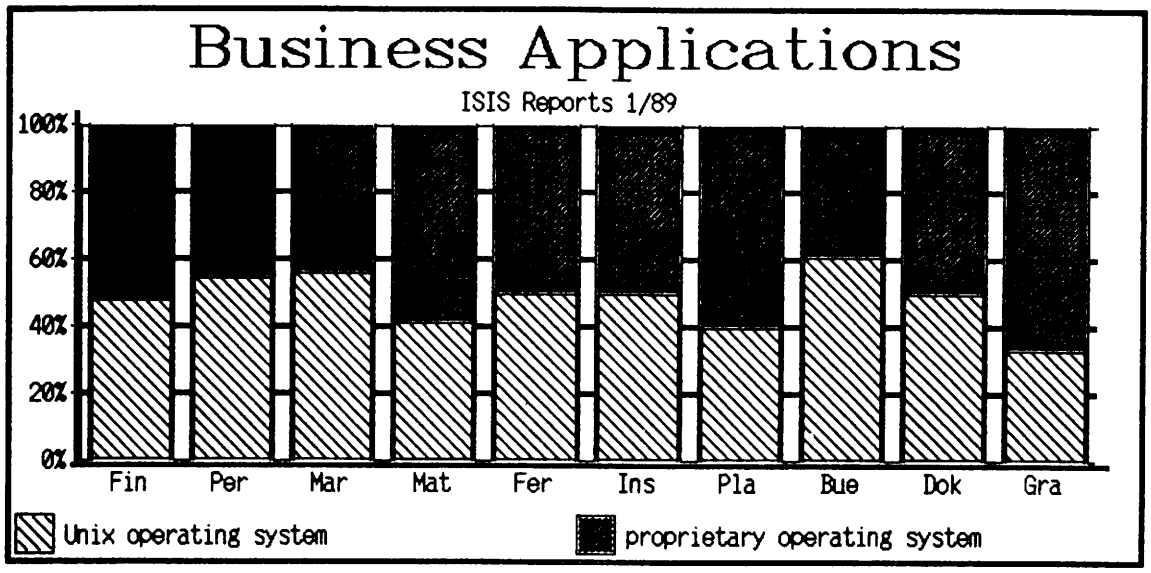


Figure 5

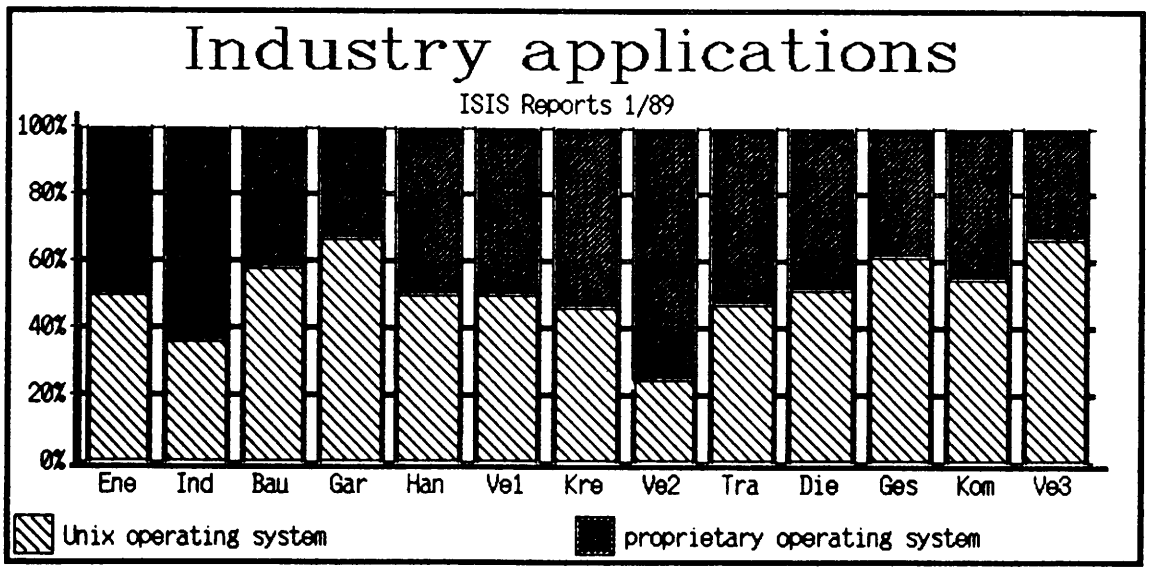


Figure 6

- business applications
- industry applications
- system software

The bars show the cumulative values of both solutions. Each bar represents a specific field in the application area. A value of 50% shows, that there is the same number of programs in each of both programming worlds, relative to the total amount.

The following tables (Tables 1, 2, 3) shows the absolute and relative figures and explains the application specific abbreviations.

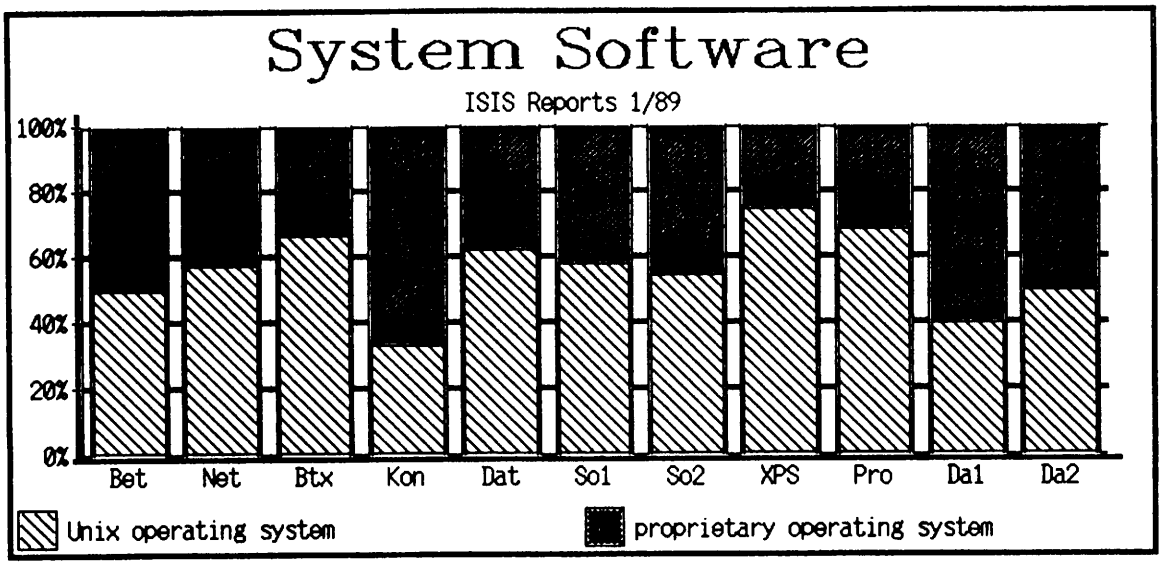


Figure 7

Business Applications		UNIX	%	Prop.SW	%
Fi	nanz-/Rechnungswesen	169	27	573	29
Pe	rsonalwesen	75	12	199	10
Ma	rketing	90	14	212	11
Ma	terialwirtschaft	60	10	278	14
Fe	rtigungswirtschaft	80	13	262	13
In	standhaltung	5	1	26	1
Pl	anungssysteme	28	4	111	6
Bu	erokoimunikation	90	14	181	9
Do	kumentationssysteme	16	3	49	3
De	sktop Publishing	6	1	6	0
Gr	aphik-Software	7	1	33	2
Au	torensysteme	3	0	14	1
TOTAL		629	100	1944	100

Table 1: *Business Applications*

In addition there is also a fourth category: technical application. But because of slight differences in the representation only the cumulative figures are presented:

	UNIX	Prop.SW
Technical Applications	249	936

Table 4: *Technical Applications*

Industry Applications	UNIX	%	Prop.SW	%
Energiewirtschaft	13	2	28	2
Industrie	46	9	199	16
Bauwesen	57	11	98	8
Gartenbau	11	2	18	1
Handel	136	25	324	25
Verlage	17	3	41	3
Kreditwesen	34	6	89	7
Versicherungswesen	4	1	36	3
Transportwesen	45	8	112	9
Dienstleistungen	87	16	188	15
Gesundheitswesen	44	8	67	5
Kommunalwesen	31	6	66	5
Verbaende	10	2	17	1
TOTAL	535	100	1283	100

Table 2: *Industry Applications*

System Software	UNIX	%	Prop.SW	%
Betriebssysteme	50	14	202	14
Netzwerke	52	15	154	11
Btx	6	2	19	1
Konvertierung	4	1	30	2
Datenverwaltung	34	10	90	6
Software-Entwicklung (1)	100	29	301	21
Software-Entwicklung (2)	21	6	76	5
Software-Entwicklung (3)	22	6	28	2
Programmiersprachen	45	13	80	6
Datensicherung	6	2	40	3
Datenerfassung	5	1	14	1
TOTAL	345	100	1404	100

Table 3: *System Software*

The interpretation of the data (e.g. business applications) shows a lot of UNIX "activity" in personnel (Per), marketing (Mar) and office automation (Bue). On the other hand independent software vendors can hope to get into business with insurance companies (Ve2) which seem to be heavily oriented on proprietary operating systems.

4. Relevant Information

Many customers see UNIX as a major alternative and want to be free in mixing brand and size of computers to build any network according to their needs. There is still a lack of relevant information although the technical solution is ready on the market.

We are convinced that the short summary helps to get a first impression about UNIX application development in German speaking countries and represents a first step towards better evaluation of better evaluation applications in different computing worlds.

5. Bibilography

ISIS UNIX Report, 1. Jahrgang, 1/89
ISIS Software Report, 20. Jahrgang, 1/89
ISIS Personal Computer Report, 7. Jahrgang, 1/89
ISIS Engineering Report, 6. Jahrgang, 1988/89
ISIS Reports: Nomina Gesellschaft fur Wirtschafts – und Verwaltungsregister mbh, Munchen

Using an Object-Oriented Model of UNIX for Fault Diagnosis

Anita Lundeberg

Department of Artificial Intelligence

Gail Anderson, Paul Chung

Artificial Intelligence Applications Institute

Alex Zbyslaw

University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
United Kingdom
gail.anderson@ed.ac.uk

ABSTRACT

The modification of an existing system based on a model of UNIX is described. Object-oriented programming techniques are employed in a new version of the model. This is combined with code for diagnosis of problems with UNIX and with a program to generate parts of the model automatically, producing a general diagnostic system. A demonstrator application for the system is developed.

1. Modelling UNIX

This paper describes project work towards the Degree of Master of Science (Information Technology) at the University of Edinburgh. The aim of the project is to modify and supplement an existing diagnostic system [And88a, And89a] for the UNIX Operating System.

The existing system used a model of UNIX which was based on the filesystem structure. It was written in Inference's Automated Reasoning Tool (ART) [Ind88a], a sophisticated programming environment for Artificial Intelligence applications. The model was based on the user's view of the UNIX filesystem and included representations of the filesystem structure, processes and programs, and of the relationship between hardware and UNIX devices. The model was used successfully as the basis of a program to help the user run filesystem checks on UNIX machines, and was also used (rather less successfully) to perform diagnosis of problems which occur during a boot procedure and cause the boot to fail. The version of UNIX modelled was SunOS Version 3.2.

The goal of the current project was to improve and extend the existing system. The filesystem model was to be refined and the modelling of processes and programs expanded using object-oriented programming techniques. The method for diagnosis was to be improved by making better use of the facilities within ART for reasoning about hypothetical and parallel worlds. The system was to be generalised, first by producing facilities for automatic generation of the filesystem model, and second by adding code to handle diagnosis of problems with a running UNIX system.

The project has resulted in a useful demonstrator system, which can diagnose simple problems which occur with line printing. The system cannot cope with multiple faults. It incorporates a slightly updated version of the filesystem check helper and a program to generate models automatically. The practical work has been completed – it will be documented in [Lun89a].

The project work was divided into several parts. First, the program for automatic model generation was written. Second, the model of UNIX was updated and refined, and the filesystem check module was updated to work with the improved model. Third, the general diagnostic code was written and a specific application (diagnosis of problems with line-printing using a simplified model of the line-printing software) was developed. Lastly, all the constituent parts were integrated to form the demonstrator system.

2. Why Use Model-Based Reasoning?

There are several reasons why it makes sense to build systems which base their reasoning on models of the "real world". Humans tend to base their thinking on internal models of the world around them. This makes it easy for them to notice implications and draw conclusions based on the information stored within their internal models. It allows them to infer how an object will behave in any given situation from their knowledge of its structure. Computer programs which use model-based reasoning are able to use reasoning which is closer to the way in which people think, and which embody a deeper understanding of their domains than do programs which are based merely on a collection of observations. In addition, constructing a model of a domain encourages the programmer to structure the information; systems constructed from lists of observations and conclusions, on the other hand, do not by their nature encourage the programmer to represent knowledge in a tidy, maintainable and understandable fashion. See [ART84a] for further information on model-based reasoning.

3. Changes to the Existing System

There were three changes made to the existing system:

- the filesystem model was refined
- an object-oriented representation of processes and programs was introduced
- the method for diagnosis was improved (partly by making better use of the facilities within ART for reasoning about alternative worlds)

Frames are available within ART as ART *schemata*. The original system used schema hierarchies to represent the filesystem. There was a *unix-object* hierarchy to represent the directory structure. Several types of unix-object were identified, including *directory, plain-file, special-file*. Within the sub-hierarchy *plain-file* files were organised according to increased specialisation – there was s sub-type *exec-file* to represent executables and a sub-sub-type *bootable-file* to represent bootable images. The new filesystem model has abolished those sub-types; executability is represented completely in terms of the mode slot and bootability in terms of the contents of the file. The new representation is more consistent and more in tune with the "real" UNIX. The revised *unix-object* hierarchy is shown in Figure 1, and a sample *plain-file* schema in Figure 2.

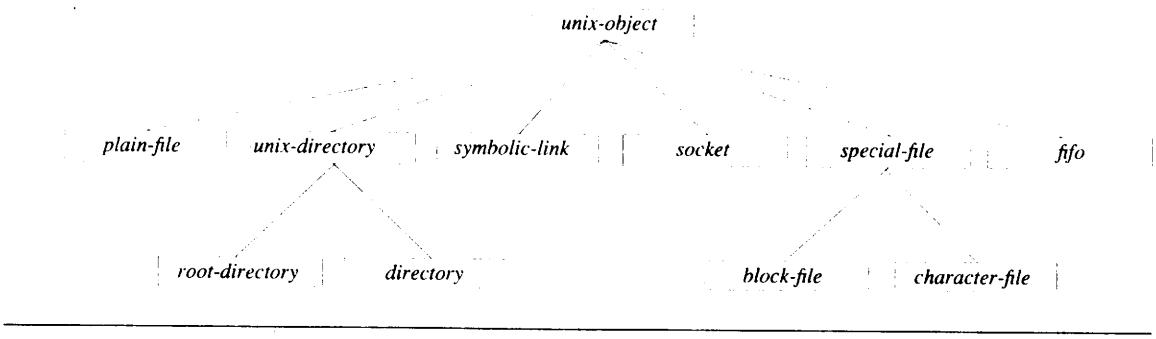


Figure 1: *The* unix-object *Hierarchy*

```
;;; Name:   bin-rm
;;; Purpose: Represents /bin/rm – the command which removes files.

(defschema bin-rm
   (instance-of plain-file)
   (real-name #Ll/bin/rml)
   (in-directory bin)
   (mode ((r w x)(r - x)(r - x))))
```

Figure 2: *A Sample* plain-file *Schema*

3.1. Object-Oriented Modelling

Within the original system there was a rudimentary representation of processes. Transient processes were never created as such within the model; they were modelled by executing the LISP functions which represented the programs (for example, the LISP function *u-rm* represented the actions of the UNIX program *rm*). Daemon processes were represented as schemata, which were created from the appropriate LISP functions.

One of the goals of the project was to improve the representation of processes and programs by making use of the facilities for object-oriented programming within ART. First, the notion of the contents of a file was separated from the notion of the file itself. The old *contents* slot was modified so that, instead of containing the data itself (usually a LISP-like representation of the data on the real disk) it contained a pointer to the schema in which the information was stored. A *file-contents* hierarchy was constructed, within which various types of file-content were identified. This is shown in Figure 3.

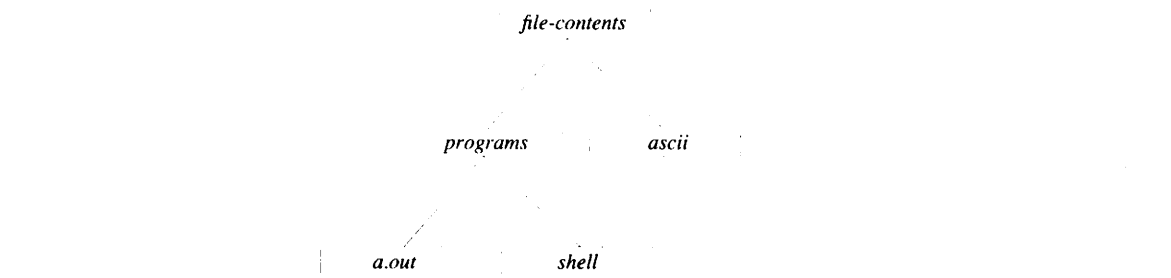


Figure 3: *The* file-contents *Hierarchy*

Separating the contents from the file allows the representation of hard links (previously omitted from the model). In addition, it models more closely the way users think of UNIX; a *program* will not normally be thought of as equivalent to a *file*.

The contents of a text file will be represented as an instance of *file-contents*. The representation of programs and processes is approached in a rather different way. The actions of a program are represented as a *method* invoked on a *program* schema. See Figure 4 for the instances of *plain-file* and *a.out* which represent *lpr*. When a process is fired up from an executable file in the model (either from within a rule, or from another method) the following sequence of actions occurs (see Figure 5):

- a message is sent to *usr-ucb-lpr* (an instance of *plain-file*) – which is where *lpr* is normally kept
- the name of the appropriate *file-contents* schema is looked up – it is *p-lpr* – and a message is sent to it
- this message invokes a LISP method which emulates the program action and creates an instance of a *process* schema

```
;;; Name:     usr-ucb-lpr
;;; Purpose:  Represents file /usr/ucb/lpr

(defschema usr-ucb-lpr
   (instance-of plain-file)
   (real-name #Ll/usr/ucb/lprl)
   (in-directory usr-ucb)
   (contents (program p-lpr))
   (condition 0))

;;; Name:     p-lpr
;;; Purpose:  Represents the program lpr

(defschema p-lpr
   (instance-of a.out)
   (stored-in usr-ucb-lpr))
```

Figure 4: *The Schemata which Represent* lpr

- when the execution on the method is complete, the process schema is destroyed and control is regained by the rule (or method) which called the program originally

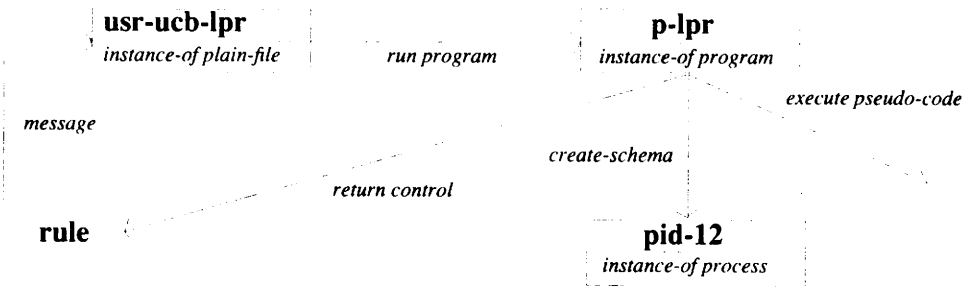


Figure 5: *Running Programs within the Model*

3.2. Better use of Viewpoints

The original system attempted to use the model to diagnose problems occurring with the boot process by generating a search tree of all possibilities and pruning this tree according to the answers given by the user to the system's questions. The search tree, however, proved unmanageably large. The boot diagnosis part of the system was never completed because of time constraints. One of the project goals was to improve the mechanism for diagnosis by making better use of the facilities within ART for reasoning about alternative worlds. Better ways were used to generate possible solutions – see section 5 for a description of the diagnostic method.

4. Approaches to Generalisation

Two approaches to generalising the system were adopted. First, a program to generate the model of the filesystem structure automatically was developed. This generates only the *unix-object* hierarchy; it does not generate the file-contents. The automatic generator is described in section 6.

Second, the system was generalised so that it could be used to diagnose problems with a running system. This work was combined with improving the diagnostic method, and is discussed in section 5.

5. The Diagnostic System

The diagnostic system is composed of several elements:

- the model of the filesystem structure (which can be generated automatically)
- general diagnostic rules (representing meta-knowledge about reasoning about problems using this type of system)
- several domain-specific elements
  - a file-contents hierarchy appropriate to the domain, and knowledge about where certain programs usually reside
  - a series of tests which can be carried out by the user
  - a set of mappings between the results of these tests and their representation in the model.

The system uses the model to reason about the problem by hypothesising a few possible faults. It then asks the user to carry out a series of tests, and compares the results of the tests with the current hypotheses, eliminating those which conflict with the user's answers.

5.1. Generating Possibilities

The different components defined within the domain are all possible candidates for causing a system failure. Examples of components from the line printing domain are the user command, the line printer daemon, and the printer queue. Associated with each component is its reliability (its likelihood of causing a failure). Reliabilities are decided according to expert advice. In a complex system not all possible components can be considered at the same time, and a fallibility threshold is set. Components are considered only if their fallibility exceeds this threshold. For each component under consideration a hypothetical world is constructed in which that component is assumed to be faulty. Later, each world will be compared with the user's description of the real situation.

A hypothesis is removed from the tree of possibilities if it is found to be contradictory to the information given by the user. If all alternatives in the current set are eliminated, a new set of less likely faults should be considered, and so the fallibility threshold is lowered.

5.2. Selecting Tests

At each stage of the fault localisation the system recommends to the user the test which it considers to be most suitable at that time. The method used for selecting a test is similar to that described in [Ind88b]. The recommended test must involve at least one component from the set of those currently under consideration.

There is a reliability defined for each component within the system and a difficulty defined for each test. Also, associated with each test is a list of those components which are involved in that test.

A utility factor is calculated for each possible test using:

- the total component reliability for the components currently under suspicion which are involved in the test (calculated from the individual reliabilities of the components)
- the total component reliability for the components currently under suspicion (calculated from the individual reliabilities of the components)
- the difficulty of the test

The system compares the utility factors of all possible tests, and selects the one it considers most suitable. In general, for a test to be most effective, it must be equally likely to succeed or fail (if it is very likely to succeed or very likely to fail, the system will be likely to gain little new information from it). This means that the system's ideal test will involve half of the total reliability of the components still under suspicion.

6. A Demonstrator Application – Line Printing

A demonstrator application was developed. This is a simplified diagnosis system for problems which occur with line printing in a running UNIX system.

The model of the domain was restricted as follows to allow for rapid development of the demonstrator:

- the model covers local line printing only, so the line printer daemon listens at only one socket
- the only request understood by *lpd* is *start printer*
- there can be only one printer for each spool queue (so there will be a maximum of one running *lpd* for each queue
- the information in */etc/printcap* is complete, but only the queue and printer definitions are used

Among the faults the system can detect are faulty hardware, printer or print queue disabled, hung printer daemon and certain mistakes in the user command.

7. An Example Run

The operation of the system is best described by going through an example run. The example below is greatly simplified, but adequately shows the methods used for setting up, and reasoning about, the hypothetical alternatives. The tree of alternatives is shown in Figure 6.

The system first asks the user a few basic questions. It then generates two hypotheses using the method described in section 5.1.; one in which the printer is assumed to be the failing component, and one in which the line printer daemon is suspect. The system reasons that at this stage it is most effective to test whether the printer is on-line. If it is found to be on-line, a new test (whether the printer is enabled or disabled) is chosen. If the user replies that it is disabled, the test has failed, and all components involved in the test are marked as suspects. All other components are marked as working properly. In general, a failing test implies that all components not involved in the test are working properly, and are marked as such. In this example, the world in which the line printer is assumed to be faulty is is inconsistent with what is currently believed, and this hypothesis is eliminated. The resulting search tree is shown in Figure 7.

After pruning the search tree, the system compares the different hypotheses (in this example only one) with the situation described by the user. If it finds a matching world, and if the answers the user has given are considered to be enough evidence for the world to be believed, the fault is found and appropriate advice is offered. A simple trace of the system's reasoning is available.

8. Automatic Model Generation

The automatic model generation is done by a C program which examines the real filesystem structure and generates the appropriate ART definitions. It uses a template file, recognising certain tokens and generating file-specific information in their place. It can generate schemata for individual *unix-objects* specified in the command list, or can recurse automatically down a directory structure. A sample template is shown, with the corresponding output, in Figure 7.

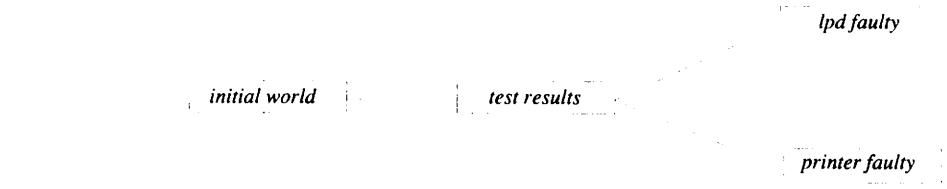
The automatic modeller is available from within the diagnostic system; the user can choose to generate her own choice of files, or can ask the system to generate all those necessary for a particular domain (in the demonstrator system the only domain available is line printing).

9. Further Work

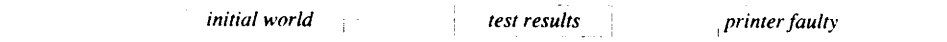
Two major areas for further work have been identified, each of which would help to increase the general applicability of the system.

9.1. More Sophisticated Automatic Model Generation

The automatic model generator can construct only the *unix-object* hierarchy at present. One possible direction for development lies in examining the possibilities for generating the contents of files automatically. It is not possible to generate the LISP code necessary to represent the actions of compiled programs automatically, but there are possibilities for parsing UNIX text files and generating other types of file contents. Alternatively, the system could read the real data files (in line printing, for example, */etc/printcap*) rather than examining a model of them.



while two alternatives are being considered



once *lpd* has been eliminated as a subject

Figure 6: *The Search Tree*

9.2. Interaction with UNIX

This opens up the possibility of running the system in conjunction with the UNIX system under diagnosis. That is, it would be possible to interface the diagnostic system to UNIX. One obvious way of doing this would be to have the diagnostic system running on a networked machine, and using the network to interrogate the faulty UNIX system. This approach could be extended by making the diagnostic system actually fix the problem rather than just suggesting a solution. For example, it could restart the printer daemon itself, rather than saying " go and type */usr/lib/lpd &*".

10. Conclusions

The construction of a model of UNIX lends itself to several applications. The part of the system which advises a user running *fsck* uses the model directly (by running a simulated boot) to discover the importance of individual files. The line-printing diagnostic system demonstrates some of the possibilities for diagnosis of problems with UNIX through modelling aspects of the operating system. Although it is not feasible to construct a model of UNIX as a whole, and although parts of the model must be "hand-crafted", models of particular domains can be combined with heuristic information to produce useful help systems.

```
;;; Template for a directory

(defschema #d #N
  (instance-of directory)
  (real-name #R)
  (sub-directory-of #D))

;;; Output

(defschema  usr-spool
  (instance-of directory)
  (real-name #Ll/usr/spooll)
  (sub-directory-of usr-root))
```

Figure 7: *A sample template and output of a directory*

References

[ART84a] *Artificial Intelligence*, 24, 1984.

[And88a] Gail Anderson, "A Model-Based Diagnostic System for Sun Workstations," M.Sc. Dissertation, Department of Artificial Intelligence, University of Edinburgh, 1988.

[And89a] Gail Anderson, Paul Chung, and Robert Inder, "A Model-Based Diagnostic System for the UNIX Operating System," *UNIX: European Challenges, EUUG Conference Proceedings*, Spring 1989.

[Ind88a] Robert Inder, "The State of the ART," Technical Report AIAI-TR-41, A.I. Applications Institute, University of Edinburgh, 1988.

[Ind88b] Robert Inder, "Experience of Constructing a Fault Localisation Expert System Using an AI Toolkit," *Proceedings of the First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE–88*, June 1988. Also published as AIAI-TR-47, A.I. Applications Institute, University of Edinburgh, 1988

[Lun89a] Anita Lundeberg, "A Model-Based Diagnostic System for UNIX," M.Sc. Dissertation, Department of Artificial Intelligence, University of Edinburgh, 1989. due for submission 1989

Trademarks

Modelling the NFS service on an Ethernet local area network

Floriane Dupre-Blusseau

Centre National d'Etudes des Télécommunications
Service LAA/ITP/GMI
Route de Trégastel
BP 40
22301 Lannion Cédex
France
blusseaf@lannion.CNET.FR

ABSTRACT

This article contributes to the field of performance evaluation based on simulation of queueing networks. Obtaining information on the performance of a local area network supporting the distributed file system traffic of NFS (Network File System) is the main goal of our work. We want to find out beforehand, with scientific methods and no more empirically, whether excess load brought about by the addition of connected workstations on the network, saturates the network. To achieve this goal, we studied NFS and the underlying protocols with a view to performance. Based on the generated load on the network and the processing delays, we retained the most relevant features in order to complete an NFS model. Then we chose the most appropriate software to build our model, which was QNAP (Queueing Network Analysis Package). Finally we designed and solved the NFS model, using simulation and by varying entry parameters.

One of the main results relates to the acquisition of an NFS model which does not yet exist, and of a methodology also usable in other environments. We are able to determine, by solving the model, the greatest number of connected workstations working together on the local area network. One also can use the model to choose between several network utilization policies, the most appropriate one, provided that the criteria were first fixed.

1. Overview

The model of the NFS service was arrived at through modular modelling where each module corresponded to one of the protocols used for NFS operation. Once models have been obtained for each protocol, it is possible to reconstitute the overall model of the NFS service, associating the inward flow (respectively outward) of the model of one protocol of a given level to the outward flow (respectively inward) of the model of a protocol of the immediately superior (respectively inferior) level, in the terms of the OSI reference model.

The chief advantage of this method, which consists of modular modelling of a service by associating each module to one of the protocols on which it is based, lies in the ease of writing the model, and in the ease with which the model is understood. In addition, any modifications which might have to be made to the model are easier to perform. If the service were to be performed by new protocols, or if the model were to be enhanced, it would be quite simple to modify the model accordingly by inserting a new model. Lastly, modular modelling makes it possible to obtain intrinsic performance characteristics for each level or layer of the protocol, and thus makes for easier identification of the various bottlenecks in the system under evaluation.

We can attempt to draw a parallel between protocols used by the NFS service and those of OSI architecture, but it must not be deduced that NFS and subordinate protocols use an OSI standard. The NFS operation level is equivalent to the application layer of the OSI reference model. In order to perform its distributed file system service, NFS uses the following protocols as a base [Ste86a]:

- XDR (eXternal Data Representation): performs a task equivalent to that of the Presentation layer of the OSI reference model;
- RPC (Remote Procedure Call): performs a task equivalent to that of the Session layer of the OSI reference model;
- UDP (User Datagram Protocol): operates at a level equivalent to the Transport layer of the OSI reference model;
- IP (Internet Protocol): operates at a level equivalent to the Network layer of the OSI reference model;
- Ethernet: operates at the level of the Physical and Link layers of the OSI reference model.

2. Study of performance of NFS and subordinate protocols

2.1. Introduction

Study of the performance of NFS and its subordinate protocols called for choices to be made at parameter level so as to keep only the most pertinent ones, for it would be unthinkable to run a model of the NFS service and its subordinate protocols which took account of all their characteristics, partly because we need only information on the performance of the service, and partly because the model obtained has to be resolved.

The criteria adopted to obtain a list of parameters judged pertinent to the NFS service model are essentially those of *load* and *delay*. This choice is motivated by the fact that the results of interest to us concern the network load factor, the user request processing delay, the collision rate, and the rates of request resends and drop out.

Not all the parameters screened through these criteria are necessarily used in creating the model: some of them will be seen to be of negligible value with respect to other processing delays or to the load already created by other parameters.

2.2. Ethernet performance study

The performance criteria mentioned in the introduction of this section make it possible to determine which characteristics are likely to influence the performance of the Ethernet network. Essentially, the elements of the protocol which modify the processing times and network load will be dealt with.

The Ethernet protocol breaks down into three levels: LLC (Logical Link Control) layer which provides the interface with protocols of the next level up, the MAC (Medium Access Control) layer which submits frames to the medium and resolves contentions, and the PLS (PhysicaL Signaling) layer which conveys frames and detects collisions.

The performance characteristics of the LLC layer correspond to a single delay for processing data, i.e. packing of data frames for transmission and unpacking upon reception. This delay is 0.375 ms [Mei87a].

The MAC layer submits frames to the medium, which gives rise to a preparation delay of 0.175 ms [Mei87a] and a time between submission of two frames defined in the standard as 9.6 microseconds. To these service times can be added a wait time due to the fact that the medium is busy transmitting another frame. Sun workstations have a special feature that allows them to submit several frames to the Ethernet board simultaneously. This means that the time between transmission of two frames corresponds solely to the inter-frame delay. Frames transmitted one after the other are prepared during transmission of the first frame. The number of frames which can be submitted in this manner is limited to six. The MAC layer also resolves contentions by using the following characteristics:

- Maximum number of resends used in calculating wait time = 10
- Maximum number of resends permitted = 16
- Jam size = 25 bits

The wait time before the n^{th} resending are random whole numbers uniformly distributed between 0 and 2^k, where $k = \min(n, 10)$.

The performance characteristics of the MAC layer upon receipt of a frame correspond solely to the time required to find the destination address in the frame, i.e. 0.175 ms [Mei87a].

The PLS layer transmits frames on the medium. The propagation delay on the bus is a basic characteristic of the Ethernet protocol. This value depends on the length of the bus, its transmission rate, and its propagation speed. The local area network of the CNET at Lannion comprises a 500 m long bus with a transmission rate of 10 Megabits per second and a propagation velocity of 200,000 km/second. This gives an end-to-end propagation delay of 2.5 microseconds and a minimum frame size of 50 bits (twice the end-to-end propagation delay). The maximum amount of data contained in an Ethernet frame is 1518 bytes.

2.3. Study of performance of UDP and IP protocols

The processing performed by the UDP protocol is very limited since there is no check at this level. The associated service time is therefore negligible.

The size of the UDP buffer on the other hand can influence network load, for messages from the higher levels of protocol have to be broken up in order to fit into the buffer. Consequently, the smaller the UDP buffer is, the greater is the number of messages sent. The standard defines a size of 8192 bytes for the UDP protocol. However, current implementation of the NFS service on the Amdahl computer uses buffer whose size is restricted to 2048 bytes.

The processing performed by the IP protocol basically consists of determining if the datagram received has to be fragmented before transmission to the subordinate protocol. In our study, the lower network is of the Ethernet type. It has been seen above that the maximum data size of an Ethernet frame is 1518 bytes. When the IP protocol receives a datagram larger than this, it has to break it down into several smaller elements. A boolean symbol is placed during fragmentation to indicate if each fragment is the last in the datagram.

When the IP protocol receives frames from the Ethernet protocol, it must rebuild the original datagram (if it has had to be fragmented) and then send it to the higher protocol.

The processing times of the IP protocol were obtained with an analyser for the Ethernet network (LANanalyser EX 5000E by EXCELAN). The times for fragmentation and re-assembly of a datagram are each 4.5 ms, giving 9 ms in all. When a datagram contains no more than one Ethernet frame, the IP protocol processing time simply corresponds to the data length test, i.e. 1.5 ms out and in, or 3 ms in all.

2.4. Study of performance of the NFS service

2.4.1. Introduction

The performance characteristics of the NFS service are obtained by analysing the processing sequence of a command sent via the service, from the moment of transmission to receipt of the corresponding reply, retaining only those parameters which are likely to effect network performance, i.e. those which increase load or processing delays.

This means that for each protocol (NFS, XDR, and RPC), as well as for client application of the NFS service and for the NFS server, the characteristics considered essential for the performance evaluation will be dealt with.

Current implementation of NFS on the Amdahl computer makes use of only the server aspect. The client application of NFS will therefore be represented by user commands sent from the Sun workstations via NFS, the server being represented by the Amdahl computer.

2.4.2. Performance characteristics of the client application

Within the scope of NFS service modelling, we dealt only with user commands which use this service, i.e. user commands using files or directories. The main commands concerned are as follows [Sun86a]:

cat file_name	displays the contents of the file on the screen.
cd dir_name	change of directory.
cp f_n_1 f_n_2	copy of files.
diff f_n_1 f_n_2	supplies the differences between the two files.
ls	lists the contents of the current directory.

mkdir dir_name	creates a directory.
mv f_n_1 f_n_2	renaming of files.
pwd	prints the working directory.
rm file_name	removes the file file_name.
rmdir dir_name	removes the directory dir_name.

Once the list of user commands likely to use the NFS service has been obtained, it is important from the point of view of performance to determine the time span between the moment when the reply to a user command is received and that when the next user command is sent.

For this, the Ethernet network analyser already referred to was used. The distribution function of the random variable representing the user thinking time is obtained from values given by successive experiments. They are as follows:

1s (twice) – 1.2s (4 times) – 1.4s – 1.6s (twice) – 1.8s – 1.9s – 2.4s – 2.5s – 3.5s – 3.9s – 4.3s – 4.4s – 4.6s (twice) – 5s – 5.3s – 7.6s – 8.1s – 8.2s – 8.5s – 9.6s – 9.9s – 11.4s – 15.7s – 22.4s – 39.4s – 42.4s – 57.8s – 233.5s.

It is clear that the longest thinking time (233.5s) is far longer than the others (1sec. to 1 min.): it will not be taken into account in what follows so as not to invalidate the analyses with a non-representative value.

By regrouping the remaining times into whole-second categories, we obtain the distribution function of the "user thinking time", for which 50% of the values are less than 5 seconds, 85% are less than 16 seconds and 95% are less then 43 seconds.

2.4.3. Performance characteristics of NFS server

The NFS server must perform the tasks requested of it via the NFS requests received. This concerns basically the disk access required upon receipt of data-read or -write requests.

In order to determine the disk access time on the Amdahl computer, read/write routines were applied to a data block. The results are as follows:

Read access time:	12 ms (4 times) – 16 ms (twice) – 17 ms – 21 ms – 23 ms – 24 ms – 25 ms – 26 ms (twice) – 27 ms – 28 ms – 39 ms – 40 ms – 41 ms – 57 ms – 148 ms.
Write access time:	27 ms – 32 ms – 37 ms – 39 ms – 41 ms – 42 ms – 49 ms – 53 ms – 54 ms – 55 ms – 56 ms – 57 ms – 58 ms (3 times) – 59 ms (twice) – 106 ms – 169 ms.

The RPC protocol uses a timeout to prevent losing messages. It could be worthwhile giving priority to processing of remote requests to the Amdahl so as to prevent the clock activated by the client RPC protocol intervening too often in the event of heavy load on the Amdahl. However the Amdahl makes no distinction in processing remote or local requests, which means it would not be possible to give priority to remote requests, even if that could have improved network performance.

2.4.4. Performance characteristics of NFS protocol

A user command transmitted via the NFS service is processed due to a certain number of NFS requests. The type and number of requests is important since they will have a greater or lesser effect on network load. Yet it is not easy to appreciate the relationship between user commands and the number of NFS requests of each type generated as a result.

Using the *nfsstat* command on the Sun workstations, it can be determined which NFS requests are necessary for processing a given user command. Additionally, the command determines the proportion of each type of NFS request transmitted in a given time. On the other hand, neither the distribution of user commands or that of NFS requests are known.

A different approach must therefore be used to obtain the relationship between user commands and NFS requests. The Ethernet network analyser is again used. Through observation it was possible to associate a captured frame with an RPC request, and the RPC requests with the user commands which generated them. From this can be determined the relationship between user commands and RPC requests. Among the requests, only read requests (comprising several frames on receipt) and write requests (comprising several frames on transmission) can be distinguished from other types (each comprising one small Ethernet frame). It is not possible to make a distinction between the various RPC requests other than read/write requests. Through successive observations with the network analyser, it was thus possible to work out a distribution pattern for the number of read, write or other RPC requests per user request.

RPC read and write requests have 8192 bytes, or in other terms are the same size as the UDP buffer. The other requests have 150 bytes on average.

2.4.5. Performance characteristics of the XDR and RPC protocols

It has been seen that an NFS request is transferred through the XDR and RPC protocols before being processed by the Transport protocol. The purpose of the XDR protocol is simply to reorganize data and make it comprehensible for the various machines communicating with each other. This task has no effect on network load, since all the work is done locally and the request is not split up. On the other hand, the processing time taken by the protocol is included in the predetermined overall time for generation of a user request.

To process an NFS request, the RPC protocol uses routines running locally as well as a procedure call which corresponds to a request for service by the server [Ste86a]. Only the procedure call is transmitted via the network, and it alone therefore constitutes a load. For our study, it will be considered that for an NFS request there is a single RPC request, i.e. that which makes the remote procedure call.

The RPC protocol uses a timeout which triggers retransmission of requests. Its default value is 0.7s, which is multiplied by two for each retransmission. After three retransmissions, processing of the request is dropped out.

2.5. Summing up of a user request processing

2.5.1. Introduction

In order to set the ideas out clearly, we will sum up the different information gathered, in the example of a file copy.

Let us assume that a user has mounted two directories, one under the local directory *mnt1* and the other one under the local directory *mnt2*, and that the user wants to copy file *mnt1/file_name*, under the directory *mnt2*, with the same name. He will use the command *cp mnt1/file_name mnt2/file_name*. The size of the copied file is taken to be 3500 bytes.

2.5.2. Study of the command processing sequence

We will study stage by stage the processes carried out on the user command from transmission to the reception of its reply. It should be noticed that the copy was done from one remote file to another remote file.

First of all we must determined which NFS requests are produced by the user command *cp*, by applying the *nfsstat* command to the Sun workstation. The NFS requests transmitted are: getattr – lookup – read – create – write. So the *cp* command sends five NFS requests and therefore five RPC requests.

The RPC protocol functions in a synchronous mode, therefore the requests are processed one after the other, the processing of a request starting only when the preceding request has been replied.

The RPC request is then processed by the IP protocol (the UDP protocol is not taken into account as it adds on negligible load and delay). The processing by IP protocol requires a time delay of 3 or 9 ms. At this level the request can be fragmented into several datagrams if the received message contains more than one data frame. The messages corresponding to the NFS requests "lookup", "getattr", "read" and "create" will not be segmented. However the message corresponding to the write request is split into three datagrams as it concerns a file of 3500 bytes.

Each of the datagrams is then processed by the LLC layer with a delay of 0.375 ms, and then by the MAC layer with a delay of 0.175 ms. At the level of the MAC layer, it is necessary to wait until the communication medium is free before transmitting. In the event of collision, the wait time is determined before retransmission and then the frame is retransmitted.

The medium of communication transfers each frame to all the stations which are connected to it. This is expressed by a delay equal to the length of the frame in bits, divided by the medium throughput (here, 10 Mbits/s), that is 1.2 ms for a frame of maximum length.

At reception, the frames are only retained by Amdahl. Their progress across the MAC and then the LLC layers, at reception, adds on a delay of 0.175 ms and 0.375 ms respectively. The datagram is reassembled at the level of the IP module. The processing of messages corresponding to NFS requests other than "read" or "write" does not cause a processing delay on Amdahl. On the other hand the "read" and "write" requests require a disk access, that is a delay corresponding to the values mentioned in the preceding part

of the section. Once the read or write operations has been carried out, a reply is produced and transmitted across the IP and Ethernet protocols with the same delays as before. Only the reply to a "read" request causes a fragmentation into three datagrams at the IP level. When the reply to the RPC request has been received, the next RPC request can be processed.

When all the replies to the RPC requests have been received, the NFS service is terminated and the user can continue with his work.

3. The NFS service model

3.1. Introduction

Modelling of a system in order to evaluate it can be broken into phases. The first one has been described above: it consists of simplifying the system so as to retain only the most pertinent parameters. This phase is very important, since the accuracy of the model, and consequently of the results, depends on the choices made.

After writing the model itself, the model must be validated so that it can be relied on to a certain extent if not totally. In our case, this aspect consists of comparing the results provided by the model with those obtained with the network analyser.

When it is deemed that the model is sufficiently reliable, the next phase of analysis and interpretation of results is launched.

3.2. Selecting performance evaluation software

3.2.1. Selecting the most appropriate tool

A model's resolution methods call for the use of specific tools. The trend is towards integration of all resolution methods and a model description language in the same package so that the appropriate resolution module can be chosen in accordance with the model described.

Of the software available, the most appropriate to the problem at hand must be chosen. The information required concerns load, wait times, etc., or in other words, *quantitative* information. The tool chosen should therefore be based on queueing networks. Furthermore, it should enable us to resolve the model of NFS service in a local area network environment and should therefore be especially adapted to this kind of problem. Lastly, the software must be available at the Lannion CNET site.

Consequently, the QNAP (Queueing Network Analysis Package) package was chosen for this task. It enables description and resolution of queueing networks by simulation and exact or approximate analytical methods, and provides quantitative results. The fact that it was available on Multics at the CNET site at Paris, and its relative ease of use were of importance in the choice. Use of QNAP is justified, for it is perfectly appropriate for the problem in hand since the system to be evaluated can be regarded as a queueing network. In addition, the object-oriented approach of QNAP makes for easier follow-up of programme writing, as well as making it possible for the programme to be used subsequently by people other than its creators.

3.2.2. Presentation of the selected tool

QNAP [Pot84a, Ve84a] was developed jointly by INRIA and Bull company. The latest version (QNAP2, 1982) is fitted up with a user interface which defines the network and supplies a print-out of results in the form of summarized tables. QNAP2 also includes a command language which defines the different elements in the model, and an algorithmic language. The writing of a programme is quite simple once the queueing network, on which the model is based, has been clearly defined. Figure 1 shows a waiting queue.

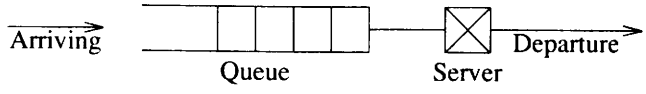


Figure 1: *A waiting queue*

When building the model, it is necessary to know the description of the customers arriving in the queue, the description of the server process for customers, and the queue discipline, that is the procedure by which the customers are chosen in the queue to be processed.

The QNAP language describes the configuration of the queueing network which is represented by a group of *stations* (a station includes one or more *servers*, serving one single *queue*), which the *customer* pass according to the *routing rules*. The customers can be allocated to different *classes* which characterize the various processes in the stations. The QNAP stations represent the physical or logical processing characteristics of the system which is to be modelled (central unit, protocol). The customers represents processes carried out by the stations.

Next, the QNAP language describes the processing performed by each station on the customers. In the case of a simulation resolution, this procedure can be described as a simple delay defined by its probability distribution, or by a more complex algorithm, including synchonization operations or customers generation.

Lastly the QNAP language allows the user to carry out a resolution check, either by initialising or updating parameters, and by activating the resolution methods, and finally by the printing of the results.

Once the tool for design and resolution of the NFS model has been decided upon, it must be determined which resolution method will be used. The level of detail required in modelling the protocols and the volume of data to be processed led to choosing simulation since the model is enormous, complex, and non-homogeneous.

3.3. Resolution of the NFS service model

3.3.1. Introduction

NFS service is modelled with the QNAP tool by associating NFS queueing networks and the subordinate protocols dealt with in the preceding study with a network of QNAP stations. The network thus comprises as many stations as there are protocols implemented by NFS, multiplied by the number of machines in the network.

The customers passing through the stations represent the commands, requests, datagrams or frames transmitted or received by each protocol.

It must be noted that for the moment, the implementation of the NFS service on the Amdahl computer takes account only of the server aspect. Therefore in the model the Sun workstations are clients and the Amdahl is their server.

3.3.2. Validation of the NFS service model

The queueing network representing the system to be modelled allowed us to write the corresponding programme in QNAP language on Multics. The resolution method chosen is discrete-event simulation, with calculation of 95% confidence intervals.

The results obtained from simulation initially correspond to a network with three Sun workstations, the Amdahl computer, and a set of other machines represented by one QNAP station.

The printout of results of the QNAP programme resolution gives the service time, the busy rate, the average number of customers on the station, the reply time, and the total number of customers served by the station, all for each station and each class of customer. To this data can be added the statistics programmed or calculated during resolution, which include the user command processing delay, the number of collisions, etc.

It is obvious that the need to obtain a model whose resolution is not too costly forces certain simplifications relative to the real system.

Regardless of the difference (be they additions or simplifications) between the model and the real system it represents, it is important that the model obtained be validated by comparing the results obtained with the model with those obtained by observing the real system.

In addition, the model will eventually be used in an operational environment, in order to determine the optimum policy for file saving for example. It is therefore vital that the model be fully reliable.

To perform this check, the Ethernet network analyser was used to compare simulated and real values. However, it is not possible to make direct comparison between these values since the difference between UDP buffer sizes on the Sun workstations where real observation is made (8192 bytes) and on the Amdahl computer used for modelling (2048 bytes) must be taken into account.

It is therefore necessary to resolve the NFS service model for a UDP buffer size of 8192 bytes in order to validate the model. The following table shows the results obtained by observation with the analyser (first column), results of model resolution with a 8192-byte unit for data handled by UDP (second column), the deviation between the first two columns (third column), and the results of simulation of the NFS model which will be used in what follows (UDP buffer size of 2048 bytes).

	Values observed on analyser (8192 byte UDP buffer) (a)	Values obtained by resolution of NFS model (8192 byte UDP buffer) (b)	Deviation [(b) - (a)] / (a)	Values obtained by resolution of NFS model (2048 byte UDP buffer)
Number of user requests generated per Sun per second	0.056	0.055	1.78%	0.069
Number of (1) RPC requests per user request	6.26	6	4.15%	13.4
Number of (2) frames per RPC request	2.34	2.25	3.84%	1.55
Number of (3) frames per reply to an RPC request	1.7	1.69	0.59%	1.21
Number of (1)*(2) frames per user request	14.65	13.5	7.85%	20.77
Number of (1)*(3) frames per reply to a user request	10.64	10.14	4.69%	16.21
Number of frames generated per Sun per second	0.8	0.75	6.25%	1.45
Mean busy rate of medium	0.89%	0.85%	4.49%	0.978%
Mean user request processing delay	0.212s	0.2191s	3.35%	0.6794s
Mean frame size	916 bytes	869 bytes	5.13%	631 bytes

Comparison of values obtained by resolution of the NFS model with a UDP buffer of 8192 bytes and values observed with the network analyser reveals differences of between 0.59 to 7.85%. If a confidence interval of 95% is assumed, three parameters are invalid. If the interval is extended to 92%, all the parameters are validated.

It can therefore be deduced that the NFS model is valid with respect to observations made with the network analyser. The model having thus been validated, it can now be used at real-life scale by performing simulations with an increasing number of connected Sun workstations, until the network saturates. This is dealt within the next section.

3.4. The results obtained by resolution of NFS model

We resolved the NFS model by simulation for different values of the number of Sun Workstations and different user thinking time values.

So we measured the relationship between these parameters, and five results, that are : the medium load, the average user request processing delay, the collision rate on the network, the resending rate of RPC requests, and the drop out rate of user requests.

The results are summarized in the graphs in the appendix. The first graph shows the medium load. We have studied this rate in a number of Sun workstations ranging from one to twenty five, and for a user thinking time with an average value of 15 seconds. The model is very useful in that it allows us to see beyond the only point supplied by the network analyser.

The second graph shows the average user request processing delay, according to the connected workstations, and to the user thinking time. We observed that for a user thinking time of 15 seconds there is a big increase in the average user request processing delay, from 20 connected workstations. The increase is greater for a lower user thinking time, with the processing delay lasting almost 50 seconds. The point at which the delays increase relates to the fact that the RPC timeout ends (0.7 second). Therefore the request has to be retransmitted, which in turn extends the processing delay accordingly.

On the third graph also, it can be seen that from 20 connected Sun workstations, the collision rate, for a user thinking time of 15 seconds, increases greatly.

The observations of the second graph are confirmed on the fourth graph, where the resending rate of RPC requests goes up to 200%.

The last graph shows the drop out rate of user requests, which remains low for a user thinking time of 15 seconds. For a user thinking time of 1 second, the drop out rate reaches 20% for 15 connected Sun workstations, and 50% for 25 connected workstations (that is drop out for one in two requests).

3.5. Use of the NFS service model with file saving

The Amdahl 5840 has a disk capacity that allows it to be a file server on the Ethernet LAN at CNET Lannion. UNIX System V is the operating system on the Amdahl, so the NFS service could be implemented.

Thus we can believe that the connected Sun workstations will save their files onto Amdahl using NFS. We want to know the LAN performance evolution while doing file saving on the Amdahl host. Furthermore, it is important to obtain the average file saving delay when there is no other load on the LAN.

We modified the NFS model considering that the connected workstations made one, and only one, file saving during the simulation. That is, the only NFS request is a *write*, whose length equals the file saving data length. When the file saving is done, the Sun work is finished.

First of all, we solved the NFS model for only one Sun workstation doing savings. We obtained a data rate of 70 Mbytes per hour. Different tests on the LAN validate this value. Averaging the available disk capacities of Sun workstations, the length of data to be saved is approximately 500 Mbytes. So, with a rate of 70 Mbytes/hour, the file saving delay is a bit more than seven hours. In this case, only one saving can be made per night.

Since successive file savings are not feasible in one night, is it possible to make simultaneous savings? To know that, we solved the NFS model for more than one connected Sun workstation. When two Sun workstations save their files simultaneously, the data rate is 41 Mbytes per hour, giving a file saving delay of about twelve hours. Thus simultaneous savings are no more feasible in one night than successives ones.

Finally we tried to know if a file saving can be made when there is other load on the LAN. We solved the NFS model for one Sun workstation doing a file saving, and ten Sun workstations sending requests every fifteen seconds. The result is the drop out of the file saving due to the retransmission of one RPC request more than three times, without receiving any reply. Thus file saving during the day is not feasible.

4. Conclusion

The methodology we used consists of breaking a service into its different underlying protocols, carrying out a study of each of these protocols with a view to performance, and then building a model based on the fundamental points of the study. Thus a collection of reusable models is built up. If, for example, one wanted to carry out a performance evaluation of a service based on the TCP (Transmission Control Protocol), IP and Ethernet protocols, one only has to model the elements concerned with load and delay in the TCP protocol (opening of connection, ...). The model could be stacked with the IP and Ethernet models

which are already built. With this methodology, it is also possible to go back later to the modelling of a protocol, in order to improve it, without having to rebuild a complete model.

One of the main results of our work is the acquisition of a model of a service such as NFS which is considered to be the standard, and was recently implemented on the Amdahl computer. Another important result is the future use of the model by the Amdahl France company, in order to evaluate other configurations.

We resolved the model by simulation for different values of the number of Sun workstations connected to the Ethernet network, and different user thinking time values. We observed that for a user thinking time of fifteen seconds, all the studied parameters increase from twenty connected workstations. When the user thinking time is one second, the great increase is near fifteen connected workstations.

The NFS model is used to evaluate different file saving policies between Sun workstations and Amdahl. We end in a transfer data rate of 70 Mbytes per hour for one Sun workstation doing savings, and 41 Mbytes per hour for two workstations simultaneously saving. These results show that only one Sun workstation could make file savings per night. Giving the increase of connected workstations, we can wonder if it was not more feasible to use other protocols such as FTP (File Transfer Protocol), indeed give up the idea of using Amdahl as a file saving server.

Now we must continue with the simulation to refine the results already obtained. Lastly, we envisage to build the model of a bridge, that reduce the network load.

References

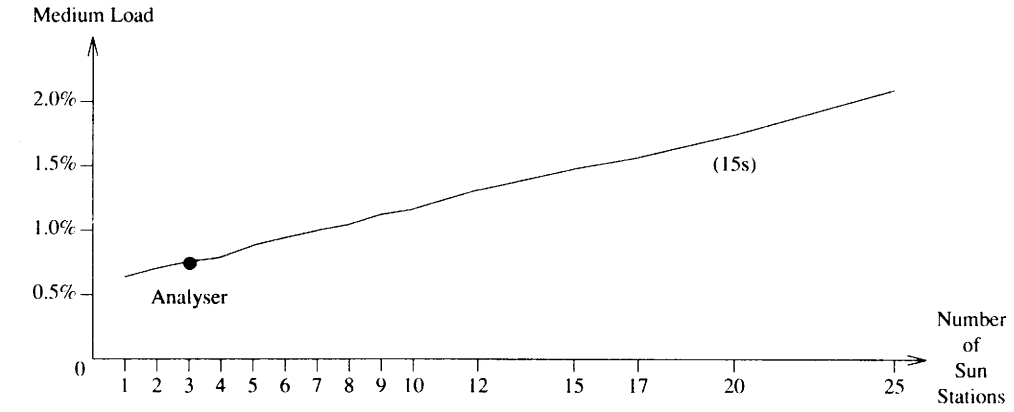
[Sun86a] *Sun System Overview*, Sun Microsystems, 1986.

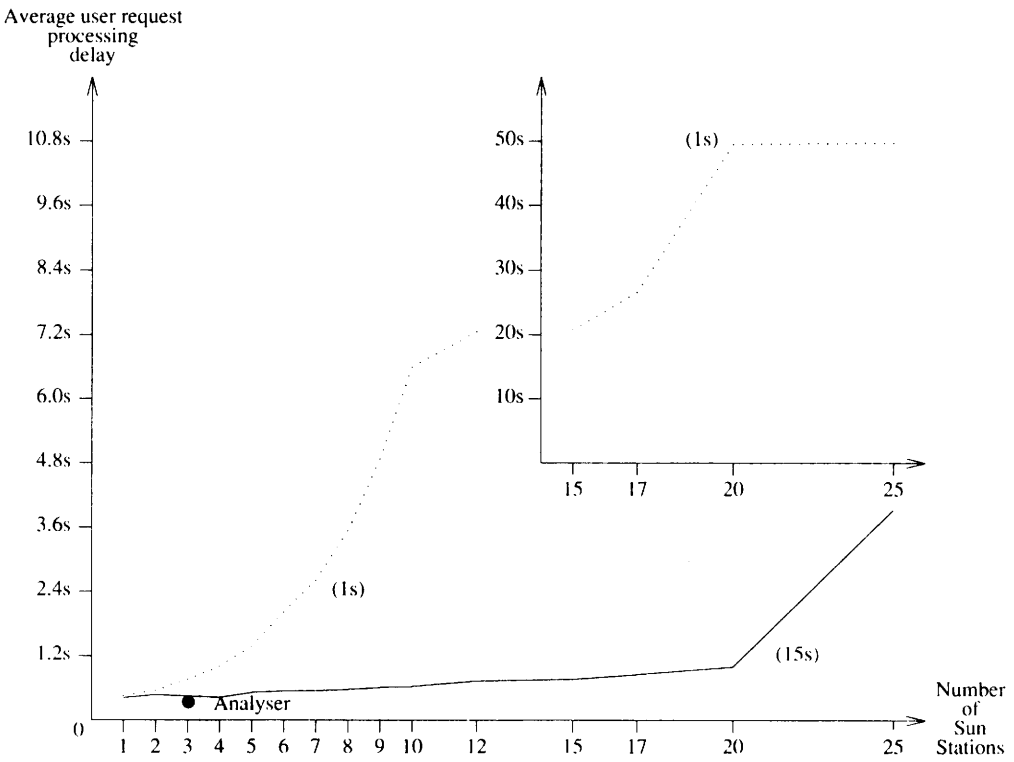
[Mei87a] B. Meister, "A performance study of the ISO Transport protocol," *The 7th international conference on Distributed Computing Systems*, Berlin, 21 to 25 September 1987.

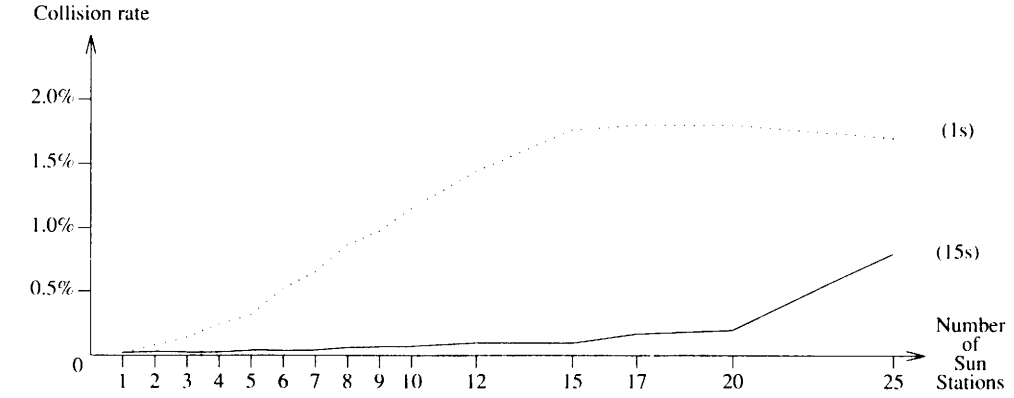
[Pot84a] D. Potier, "New users' introduction to QNAP2," Rapport Technique INRIA No 40, INRIA, October 1984.

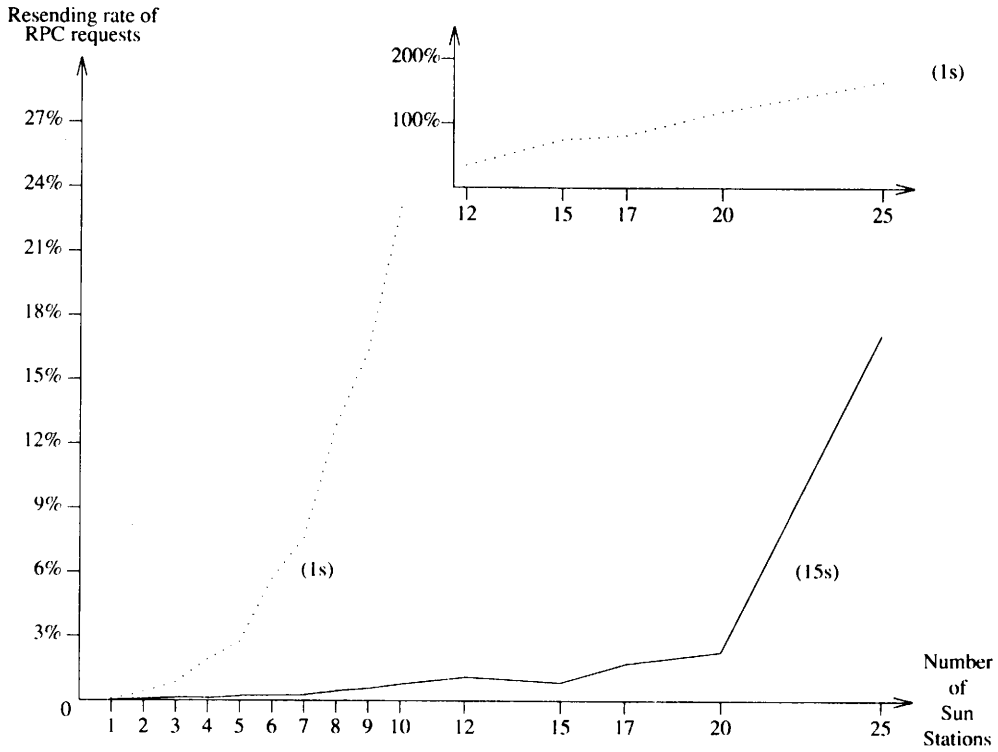
[Ste86a] Mark Stein, *The Network File System*, Sun Microsystems, 1986.

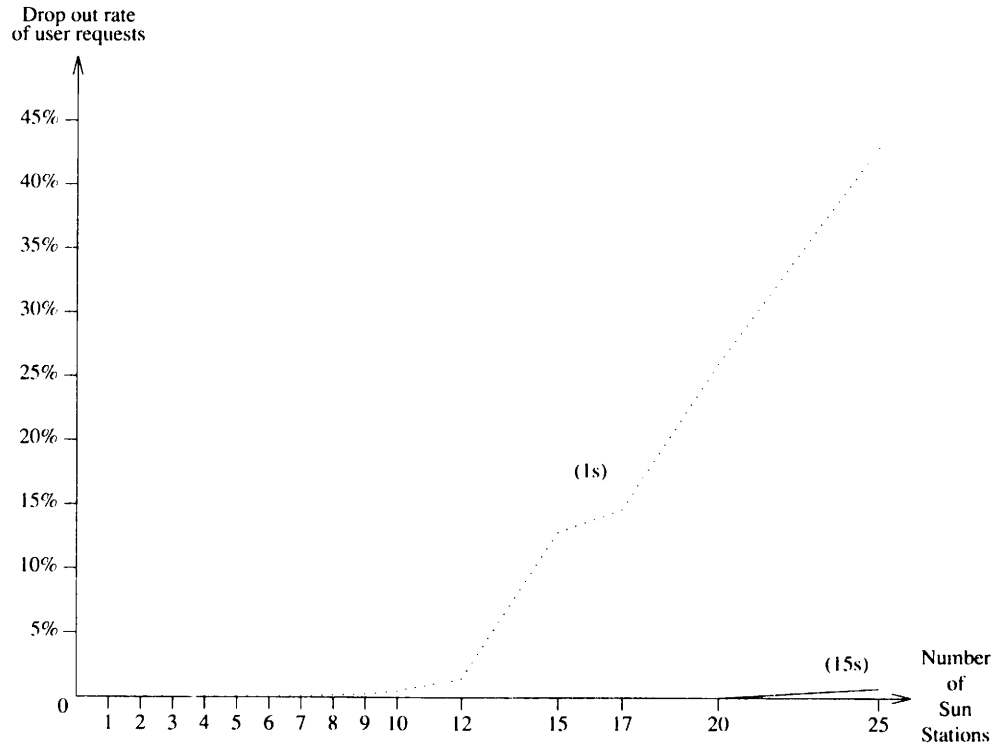
[Ve84a] Michel Vèran and Dominique Potier, "QNAP2: A portable environment for queueing systems modelling," *Colloque international sur la modèlisation et les outils d'analyse de performance*, Paris, May 1984.


Medium Load
2.0%
1.5%
1.0%
0.5%
0
(15s)
Analyser
1 2 3 4 5 6 7 8 9 10 12 15 17 20 25
Number of Sun Stations


Average user request processing delay
10.8s
9.6s
8.4s
7.2s
6.0s
4.8s
3.6s
2.4s
1.2s
0
50s
40s
30s
20s
10s
(1s)
(1s)
(15s)
Analyser
15 17 20 25
1 2 3 4 5 6 7 8 9 10 12 15 17 20 25
Number of Sun Stations


Collision rate
2.0%
1.5%
1.0%
0.5%
0
(1s)
(15s)
1 2 3 4 5 6 7 8 9 10 12 15 17 20 25
Number of Sun Stations


Resending rate of
RPC requests
200%
100%
(1s)
12
15
17
20
25
27%
24%
21%
18%
15%
12%
9%
6%
3%
0
(1s)
(15s)
1 2 3 4 5 6 7 8 9 10 12 15 17 20 25
Number
of
Sun
Stations


Drop out rate
of user requests
45%
40%
35%
30%
25%
20%
15%
10%
5%
0
(1s)
(15s)
1 2 3 4 5 6 7 8 9 10 12 15 17 20 25
Number
of
Sun
Stations

RISC vs. CISC From the Perspective of Compiler/Instruction Set Interaction

Daniel V. Klein

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15217
dvk@sei.cmu.edu

ABSTRACT

This paper compares the utilization of a number of different computer instruction sets by a collection of compilers. Wherever possible, several compilers were used for each architecture. This paper demonstrates that CISC instruction sets are underutilized by compilers, while RISC instruction sets are nearly completely utilized. We observe that if an instruction exists on a computer, it should be usable by the compilers for that computer. Because CISC computers have large numbers of instructions which are not effectively used by compilers, the instructions are superfluous. By eliminating superfluous and redundant instructions from architectures, future systems can run more efficiently, and algorithms can be executed with greater celerity.

1. Introduction

The data reported on in this paper are the result of a **R**educed **I**nstruction **S**et **C**omputer (RISC) assessment project conducted at the Software Engineering Institute.† When I entered into the RISC assessment project, it was with a strong bias toward **C**omplex **I**nstruction **S**et **C**omputer (CISC) architectures. I was a vocal proponent of the VAX and MC680x0 architectures, and looked at this project as an interesting exercise in which I would have my (negative) suspicions about RISC processors confirmed, and one in which I would find vindication for the CISC side in the great *"RISC versus CISC"* debate.

I have, however, come to the opposite conclusion. My research on this project has convinced me (quite consistently, I might add) that, if there is a "right" side of the debate to be on, it is the RISC side. In all features – execution speed, compiler efficiency, language consistency, and code size – the concept of a *reduced* instruction set computer has proven to be the correct architectural choice.

This paper, however, does not compare benchmark statistics, nor does it contrast the execution speeds of various machines. In general, benchmarks results present highly misleading statistics, and offer little or no insight into the subtle effects that can alter benchmark performance. The performance of the Whetstone benchmark on a machine tells you little about how fast it will run *your* programs – instead, it tells you is how fast the Whetstone benchmark ran. The Dhrystone benchmark specifically requires you to disable certain compiler optimizations, a request which defeats optimizations that would otherwise show off the power of a compiler – power which can be used quite effectively in "real life" applications.

What this paper does discuss is how well compilers can use the instruction sets for which they are targeted. The purpose of this study is to ask (through demonstration) "if the great majority of programmers write in a high level language (and not assembly language), and if certain instructions cannot be used by a compiler, then why are they in the instruction set of the machine?" Of course, some instructions will never be used by a compiler, and these are not considered in our discussion. Examples of these instructions are those used to effect context switch, return from trap or interrupt, etc. These instructions are necessary for the proper functioning of an operating system, and not for user level utilities. The "superfluous" instructions in question include the `polyf` instruction on the VAX (which calculates a floating point polynomial from a vectored set of coefficients), the redundant logical operations on the condition codes of the 680x0 (which are just as easily simulated using a simple load/store instruction and the logical operations on integers), and

† This work was sponsored in part by the U.S. Department of Defense.

other overly complex instructions which no compiler can utilize.

Although not specifically called out in this paper, many CISC architectures have instructions which the compiler is sometimes able to use, but which are so "special case" that their inclusion into the architecture is quite questionable, especially when these instructions could easily be simulated with simple combinations of other simple instructions. One example of this type of instruction is the `aobleq` instruction on the VAX, which adds one to a variable, and branches to a location if the result is less than or equal to a comparison variable. This instruction is (at best) infrequently used, and its function could easily be performed by the trio of an `add`, a `cmp`, and a `bleq` instruction. A compiler (and certainly most assembly language programmers) can just as easily use a template containing 3 instructions as they can use a template containing a single instruction.

For an audience such as the UNIX community, the next question which usually comes to mind is "of what use is this information to UNIX systems?" Answering this question entails placing the cart before the horse, and giving some conclusions before we present our methods and results. The answer to this question is simple. A great deal of UNIX systems are based on CISC architectures – the VAX and the 680x0 family are the most prevalent of these. Yet if it is indeed the case that UNIX compilers† (and in fact, compilers in general) are unable to utilize the instruction sets efficiently, is there then not a great deal of "wasted silicon" – hardware which is never used, but which nonetheless takes up space and more importantly *time* – in these systems? If a RISC system can be used more efficiently than a comparably sized and priced CISC system, then not only will the users of the RISC system benefit from the increased speed of the individual instructions, but the compilers will be able to make better use of the hardware, and thus return a second benefit to the users – that of efficient (and thus, celeritous) use of the instruction set of the machine.

2. Methods

The methods used to test the instruction set utilization of the compilers was rather simple, but we believe it was effective. A collection of integer based applications were given to the compiler, and the resultant assembly code was examined. Purely integer based applications were chosen over those that also contained floating point for a number of reasons:

1) Some processors followed the IEEE 754 floating point standard, while others implemented their own floating point instructions. While following a standard is laudable, the machine architecture should not be penalized for elaborate nature of the IEEE standard.

2) Some architectures utilized a floating point co-processor. While this is a valid option, floating point instructions can either be executed as separate instructions, or as sub-codes passed to a single co-processor instruction. Evaluating these instructions in either manner can introduce a bias in the measurement techniques.

3) Integer applications abound in the UNIX environment, but finding a representative floating point application is difficult. The use of *float* versus *double* variables can affect the use of one or another instruction mode, and it is very difficult to find a generic application which uses both variable types (while it is easy to find applications which mix *long* and *short* integers).

The eight integer applications and libraries were chosen for their size (a larger program is more likely to use a large number of language features) and breadth (different applications do different things, and hence exercise different aspects of the instruction set). The programs chosen for the evaluation were:

1) *csh* – The UNIX C-shell, containing 15,058 lines of code,†

2) *vi* – A screen based editor, containing 24,414 lines of code,

3) *libcurses* – The UNIX screen display library, containing 5,496 lines of code,

† To make this a fair evaluation, we used multiple vendor's compilers for the same architecture wherever possible. For example for the VAX, our comparisons were based on the Berkeley, Tartan Labs, and DEC (VMS) C compilers. For the 68020 and SPARC, the Sun and Gnu compilers were used. By using multiple compilers, we hoped to eliminate the bias that could be introduced by a single good (or bad) compiler.

† The lines-of-code count in all cases was determined with the command

```
cc -E *.c | sed -e '/^$/d' -e '/^#/d' | wc -l
```

4) *dc* – An arbitrary precision arithmetic package of 2,000 lines of code,

5) *bc* – A *yacc* based front end for *dc*, containing 830 lines of human and machine generated code,

6) *efl* – An extended FORTRAN language preprocessor, containing 18,281 lines of code,

7) *libmp* – The UNIX multi-precision math library, with 778 lines of math intensive code, and

8) *troff* – The UNIX typesetter runoff program, containing 7853 lines of truly baroque code.

To maintain accuracy of our results, the same source code was compiled on each of the tested machines. We present the statistics for the eight packages together, rather than inundating the reader with individual analyses. In truth, the compiler generated roughly the same instruction mix for each program, so we feel it is fair to present the average mix for each compiler. Although we would have liked to include other classes of programs in our analysis (e.g., an operating system, graphics tools, CAD tools, and database applications), the highly consistent output from the compilers for the applications which we did consider leads us to believe that we would have seen an insignificant difference in instruction and addressing mode usage from the set of applications selected.

In all cases, we did *not* count the instructions in the run-time libraries or the C initialization or finalization routines, since some of these are written in assembly language, and are not generated by the compiler.

For the CISC architectures, where the final machine instructions are nearly identical to the input assembly language, instructions and addressing modes are counted from the assembly language output by the C compiler. Since it would not adversely affect our statistics, the UNIX *jcond* instructions were counted as a single conditional branch instruction (instead of actually seeing which resulted in a simple branch, and which were a combination of a branch-around and jump instruction). For the RISC architectures, however, where the assembler typically also performs code reorganization, the instructions and modes were counted (wherever possible) from the disassembled object files. In this latter case, where assembler reorganization could substantially affect our results, extra care was taken to insure accuracy.

For each architecture, we present three tables. The first table shows the frequency with which each class of instruction is used. In these tables, the following definitions are used: "Load" instructions fetch data from memory into registers, while "store" instructions place the contents of registers into memory locations. "Shuffle" instructions move data between registers. "Arithmetic" instructions include add, subtract, and shift, while "logical" instructions include bitwise and, or, and exclusive or. "Compare" instructions are similar to logical instructions, except that they compute single bit results (for use in subsequent conditional branch instructions). The terms "conditional branch", "unconditional branch", and "call" should be self explanatory.

In the second table we show the patterns of instruction usage – the frequency with which instructions are used irrespective of type. From this we can see the percentage of the instruction set which is used the most, and what percentage is used the least. The third table compares the use of the various addressing modes available in the architecture.

3. RISC Architectures

We first consider three RISC architectures – the MIPS R2000, the Sun SPARC, and the Motorola 88100 – which will be later compared and contrasted with three CISC architectures – the Digital VAX, the Motorola 68020, and the Intel 80386.

It should be noted that the instruction mix presented here is that *generated* by the compiler, and not a list of the instructions *executed* by the applications. In other words, we present a static analysis of the code generated by the compiler, rather than a dynamic one. The two concepts are fundamentally different, and for this paper we did not research the latter. However, we have no reason to doubt that the two values will be substantially the same.

3.1. Analysis of the MIPS R2000 C Compiler

Tables 1 through 3 show the instruction mix generated for the six applications on the MIPS R2000. The compiler was version 1.31 of the MIPS compiler, running on a DECstation 3100.

The MIPS R2000 is a classic load-store RISC architecture. Data values must be loaded into registers before they can be changed by other instructions and stored back into memory. Almost all the instructions execute in a single machine cycle, but some instructions require a delay state before data becomes valid (e.g., a load requires a delay state before the value enters memory) or before a state change (branch instructions do not complete until the second machine cycle). Many of these delay states can be filled with

Instruction Class	Count	% used
Load	38098	30.0
Store	15270	12.0
Shuffle	8407	6.6
Move	**61775**	**48.6**
Arithmetic	16590	13.1
Logical	1958	1.5
Compute	**18548**	**14.6**
Compare	2120	1.7
Conditional Branch	11886	9.4
Unconditional Branch	6042	4.8
Call	9091	7.2
No-op	17646	13.9
Control	**46785**	**36.8**
Total	**127108**	**100.0%**

Table 1: *R2000 Instruction Use*

other instructions (in the case of a load instruction for example, any instruction which does not rely on the loaded data can be used). In those cases where no suitable instruction can be moved into the delay state, a `nop` instruction is used. This accounts for the high percentage of `nop` instructions in the count.†

Percentage use of Instructions	Number of Instructions
Never Used	8
< 0.05%	8
≤ 1.0%	19
≤ 2.5%	6
≤ 5.0%	4
≤ 7.5%	3
≤ 10.0%	1
≤ 15.0%	2
> 15.0%	1

Table 2: *R2000 Patterns of Usage*

The pattern of instruction usage shown in Figure 2 shows another expected result – namely that a large percentage of the instructions are used with approximately 1% of the time (each), while a few instructions are used with great frequency. The low frequency instructions and the high frequency instructions each take up approximately half of the work load.

Of the 51 instructions on the R2000, 35 are used at least 0.05% of the time, and only 8 instructions are never used. This means that the compiler uses 84% of the instruction set, and would indicate that the compiler and the instruction set are well matched (in that the architecture is adequate for the task, and the compiler covers the instruction set).

Recall that we are not considering floating point or co-processor instructions in our count, but that we *are* considering even those instructions which are likely to be used only in an operating system context. Of the 8 instructions which the MIPS compiler does not generate, 3 are associated with operating system functions, another 3 are used only when arithmetic overflow checking is necessary (which C does not require, but other languages do), and yet another is used only when a branch target is farther from the source than any of our tests allowed. If these are eliminated from our count, the R2000 C compiler is able to use all but one operation of the instruction set!

† The version of the MIPS assembler reorganizer (that part of the compiler which is responsible for filling in delay slots with instructions other than `nop` instructions) that was available to us is somewhat conservative in its reorganization strategy. The most recent version is more aggressive, and is able to reduce the percentage of `nop` instructions.

For all of the RISC architectures, some leeway should be allowed in the instruction usage count. The MIPS R2000 assembly language reference guide lists a `nop` instruction, when in fact this operation is performed by a shift by zero bits of the zero register into itself. (Other non-operations exist in the instruction set, such as an add immediate to self of 0. I am told that this choice of no-op, however, presented itself as a result of the layout of the R2000 IC mask, in that it has an instruction code of 0x00000000). A `move` instruction is similarly listed as being in the instruction set, when in fact a `move` is really just an unsigned `add` with an addend of zero. The SPARC and MC88100 instructions perform similar prestidigitization.

Address Mode	Example	Count	% used
Immediate	`35`	23178	9.8†
Absolute	`label`	28852	12.2†
Register	`r2`	140419	59.8
Displacement	`35(r2)`	42557	18.1
Total		**235006**	**100.0%**

Table 3: *R2000 Addressing Mode Use*

Looking at the addressing modes available on the R2000, we see that all of them are used, and that all of them are used with a reasonable frequency. Although absolute addressing is infrequently used, it is essential for accessing global variables (and its function cannot be easily duplicated by any combination of any other addressing mode).

3.2. Analysis of Motorola 88100 C Compilers

The 88100 is Motorola's new RISC processor, announced in late 1988. We were able to compare two different compilers for the 88100. In this case the compilers were:

1) The Green Hills 88100 compiler (version 1.8.4), and

2) The Gnu 88100 compiler (version 1.30).

The Gnu compiler was billed as untested, but generated what appeared to be correct code. Occasionally, however, it dumped core and was unable to completely process a source file (this happened extensively in the source code for *efl*, which does all sorts of questionably legal things). Because of this, the instruction counts and addressing mode usage counts for the Gnu compiler are lower than they should be. The results obtained from the two compilers were similar, although as with other sets of compilers, the individual code idioms and instruction counts varied. Tables 4 through 6 summarize the results obtained with the two 88100 compilers.

A familiar pattern is seen in the 88100 pattern of instruction usage. For this processor, the number of move instructions is somewhat low, and the logical instructions are rather high. One reason for this is that there is no "move" instruction on the 88100. This function is assumed by an `or` with the zero register into the destination register. If the `or` instructions used for this purpose (approximately 19.4% of the total instruction count for the Green Hills compiler and 10.3% of the total for the Gnu compiler) are counted as load instructions, the ratio of **Move : Compute : Control** instructions becomes **52.4 : 16.4 : 31.1** for the Green Hills compiler and **54.4 : 11.7 : 33.9** for the Gnu compiler, figures which are much closer to the norm we will see throughout this paper.

Unlike the R2000 and the SPARC, the 88100 does not use `nop` instructions to fill in delay slots following load operations. Instead, it uses a "scoreboard" register to keep track of which data registers are presently "in transit". It uses the contents of this register to cause delays whenever needed (e.g., when the contents of a data register is not yet valid due to a load in progress). Consequently there are no `nops` needed on the 88100. This accounts for the lower fraction of control operations in the instruction mix.

† In order to examine the real instructions an addressing modes used on the R2000, an object code disassembler must be used. In the disassembler, absolute mode is used only for jump instructions (including calls to subroutines). Branches and load-address instructions, although coded with labels in the assembly language, are reported by the disassembler as immediate mode operands. By counting the branches and certain `lui` instructions, many uses of immediate mode can be assumed to be absolute mode – a translation which has been performed here. Due to the inherent inaccuracy of this method, the reported values may be in error by a few tenths of a percent.

Instruction Class	*Green Hills* Count	% used	*Gnu* Count	% used
Load	23907	22.5	24611	30.4
Store	9959	9.4	9887	12.2
Shuffle	1215	1.1	1115	1.4
Move	**35081**	**33.0**	**35613**	**44.1**
Arithmetic	12996	12.2	8386	10.4
Logical	25060	23.6	9428	11.7
Compute	**38056**	**35.8**	**17814**	**22.0**
Compare	4898	4.6	4905	6.1
Conditional Branch	11502	10.8	9288	11.5
Unconditional Branch	7893	7.4	5256	6.5
Call	8762	8.3	7958	9.8
Control	**33055**	**31.1**	**27407**	**33.9**
Total	**106192**	**100.0%**	**80834**	**100.0%**

Table 4: *88100 Instruction Use*

Percentage use of Instructions	Number of Instructions *Green Hills*	*Gnu*
Never Used	23	24
< 0.05%	10	6
≤ 1.0%	10	10
≤ 2.5%	4	9
≤ 5.0%	7	4
≤ 7.5%	3	4
≤ 10.0%	2	2
≤ 15.0%	2	2
> 15.0%	0	0

Table 5: *88100 Patterns of Usage*

When we look at the pattern of instruction usage for the 88100, we see the similar curve of roughly half of the instructions being used for roughly a third of the work, another 4 instructions performing roughly half of the work, and the remainder taking up the slack. This pattern repeats itself throughout most of the architectures examined (with the greatest variation being in the number of instructions which were unused).

The Green Hills compiler used 38 of the integer instructions, meaning that 62% of the instruction set (61% for the Gnu compiler) is used by the compiler. Again, this indicates that the compiler is making effective use of the instruction set. As with the R2000, some of the never used instructions are designed for operating system use, or are used for array bounds checking. If these instructions are eliminated from our count, the percentage of instructions used rises to 70% for the the Green Hills compiler (69% for the Gnu compiler).

† The Condition and Bit Field modes are really just mnemonic devices for specifying single bits or collections of bits, and are another form of Immediate operand. They are included only for completeness, but should be counted as immediate operands.

‡ The Green Hills compiler does not use Register Indirect Index mode. One reason for this is that when it needs an index of zero, it uses Register Indirect Immediate mode with an immediate operand of 0. On the other hand, the Gnu compiler accomplishes this by using Register Indirect Index with the zero register. These are essentially equivalent when the index value is zero, and counts for 13733 of the accesses with the Green Hills compiler, 8871 of the Gnu compiler.

Address Mode	Example	Green Hills Count	Green Hills % used	Gnu Count	Gnu % used
Immediate	`35`	31207	12.4	12864	7.0
Condition†	`eq0`	9231	3.7	8715	4.8
Bit Field†	`3<0>`	826	0.3	1100	0.6
PC Relative	`label`	26663	10.6	21457	11.8
Register	`r2`	149673	59.5	103174	56.7
Register Indirect Immediate	`r2,35`	32674	13.0	24054	13.2
Register Indirect Index‡	`r2,r3`	0	—	9706	5.3
Register Indirect Scaled Index	`r2[r3]`	1192	0.5	931	0.5
Total		**251466**	**100.0%**	**182001**	**100.0%**

Table 6: *88100 Addressing Mode Use*

The addressing modes on the 88100 almost all involve registers, and syntactically appear quite similar. The differentiation is found in the instruction to which the registers are applied. The three register indirect modes are only used with the load, store, and memory exchange instructions.

Again, we see that the modes used (and as we will see later, most frequently used by complex architectures, too) are immediate, register, absolute, and some form of register indirect. In the 88100, some instructions automatically use registers as a source of an indirect address, so Register mode is counted rather high. In truth, a large fraction of this mode could be counted with Scaled mode (a special type of indirection which is explicitly called out in the assembler).

3.3. Analysis of SPARC C Compilers

The SPARC is the new "standard" architecture designed by Sun Microsystems. At least one other hardware manufacturer has adopted the SPARC instruction set architecture and is developing a GaAs version of the chip. We were able to evaluate two different compilers for the SPARC. In this case the compilers were:

1) The Sun SPARC compiler, and

2) The Gnu SPARC compiler (version 1.30).

Regrettably, the Gnu compiler was incomplete, and often dumped core during compilation (again, typically while compiling parts of *efl*). We therefore present the statistics of this compiler with the *caveat* that the results may be incomplete and inconclusive.

The results obtained from the two compilers were, however, similar. As with other sets of compilers, the individual code idioms and instruction counts varied, although the general patterns of instruction and addressing mode usage was consistent between the two compilers. Tables 7 through 9 summarize the results obtained with the two SPARC compilers.

A familiar pattern is seen in the SPARC pattern of instruction usage. For the Gnu compiler for this processor (as with both compilers for the 88100), the number of move instructions is somewhat low, and the compute instructions are rather high. One reason for this is that there is no "load address" instruction on the SPARC. This function is broken up into two instructions: `sethi`, which loads the high 22 bits of the address, and an `or` instruction which loads the low bits of the address. If the `or` instructions used for this purpose (approximately 9.5% of the total instruction count) are counted as load instructions, the ratio of **Move : Compute : Control** instructions for the Gnu compiler becomes **47.4 : 9.7 : 42.9**, which is much closer to the norm we have seen previously.

The SPARC is a difficult architecture for which to classify instructions when considering the patterns of use. On the one hand, it clearly follows many of the precepts established for load-store RISC architectures. On the other hand, it has 48 different conditional branch and conditional trap instruction mnemonics (none of the latter being used by the C compiler), branch instructions which either execute or annul the following instruction on condition FALSE, and instruction mnemonics for arithmetic and logical instructions that modify the condition code register, as well as arithmetic and logical instructions that do not.

For purposes of this study, our best intuitive sense was that the paired conditional branch and branch-annul instructions should be counted as a single instruction. We also felt that the "modify condition codes" bit was an option to the various instructions (since it essentially gates the output of the condition code calculations into the condition code register) rather than altering the essential function of the instruction.

Instruction Class	*Sun* Count	*Sun* % used	*Gnu* Count	*Gnu* % used
Load	29018	29.4	13320	19.2
Store	6822	6.9	4026	5.9
Shuffle	11717	11.9	8806	12.8
Move	**47557**	**48.2**	**26062**	**37.9**
Arithmetic	8312	8.4	5930	8.6
Logical	4354	4.4	7254	10.6
Compute	**12666**	**12.8**	**13184**	**19.2**
Compare	9866	10.0	5023	7.3
Conditional Branch	9636	9.8	5424	7.9
Unconditional Branch	4467	4.5	3465	5.0
Call	10703	10.8	7136	10.4
No-op	3779	3.8	8454	12.3
Control	**38471**	**39.0**	**29502**	**42.9**
Total	**98694**	**100.0%**	**68748**	**100.0%**

Table 7: *SPARC Instruction Use*

Percentage use of Instructions	Number of Instructions *Sun*	Number of Instructions *Gnu*
Never Used	37	40
< 0.05%	5	5
≤ 1.0%	13	14
≤ 2.5%	10	6
≤ 5.0%	8	6
≤ 7.5%	1	2
≤ 10.0%	2	3
≤ 15.0%	2	2
> 15.0%	0	0

Table 8: *SPARC Patterns of Usage*

The SPARC instruction set seems to be covered very well by the two compilers, showing the same even distribution of instruction usage of the other RISC compilers. Looking at the raw numbers, though, the Sun compiler uses 41 out of 78 instructions, or 52.6% of the instruction set (48.7% for the Gnu compiler) – a rather poor showing for a RISC architecture. If, however, we eliminate the kernel and system specific instructions from consideration (including the conditional trap instructions which are never used by C but are probably used extensively by Ada), the coverage of the instruction set rises to a much more respectable 69.5% (64.4% for the Gnu compiler).

With the SPARC, we see the same 4 basic modes being used with roughly the same frequency as the other two RISC architectures. The dual register mode – Indirect-2 – is used infrequently in a static count. This mode is very useful for stepping through arrays and structures, and we feel that this mode would be used extensively in a dynamic analysis.

As shall be shown in the subsequent sections, the four addressing modes that are shared by the three RISC architectures are the same as those modes which are used most frequently by the CISC architectures we shall now examine.

† On the SPARC, there are actually only 5 addressing modes. Register-1 is a special case of Register-2, where the zero register is used as the second register (and hence is not expressed in the assembler output). Similarly, Indirect-1 is a special case of Indirect-2, where the second register is the zero register. In the interests of examining all the possible permutations and their frequency of use, the special cases are separated from each other.

Address Mode	Example	*Sun* Count	*Sun* % used	*Gnu* Count	*Gnu* %used
Immediate	`35`	40950	23.2	10784	11.3
Absolute	`label`	21276	12.0	19363	20.3
Register-1†	`%2`	93772	53.1	54599	57.2
Register-2	`%2+%5`	0	—	0	—
Indirect-1	`[%3]`	3718	2.1	8580	9.0
Indirect-2	`[%3+%5]`	722	0.4	0	—
Indirect-3	`[%3+37]`	16145	9.1	2083	2.2
Total		**176583**	**100.0%**	**95409**	**100.0%**

Table 9: *SPARC Addressing Mode Use*

4. CISC Architectures

The second part of our analysis is to compare the results of CISC compilers with those of the RISC compilers. The three CISC machines that were chosen were those found in most UNIX systems currently on the market, namely the Digital VAX, the Motorola 68020, and the Intel 80386.

4.1. Analysis of VAX C Compilers

The VAX is a classic CISC register architecture – almost all instructions can use almost all of the addressing modes, with many instructions having both a two and three operand format. There is no need to load the instruction operands into registers – the addressing modes can reference memory as well as register based data.

The same six applications and libraries were run through the following VAX compilers

1) Berkeley VAX C Compiler (Ultrix version 1.2),
2) Tartan Labs C Compiler (version of March 12 1986), and
3) DEC VMS C Compiler (version 2.4).

The results obtained from the three compilers were very similar. While the actual code idioms generated by the three compilers were different, and while the count of individual instructions varied somewhat, the statistics that we examined for this report are surprisingly similar. All three values are reported in Tables 10 through 12.

Instruction Class	*Berkeley* Count	*Berkeley* % used	*Tartan Labs* Count	*Tartan Labs* % used	*DEC (VMS)* Count	*DEC (VMS)* % used
Move	**26949**	**40.2**	**28217**	**40.7**	**30941**	**39.8**
Arithmetic	5359	8.0	5460	7.9	6320	8.1
Logical	804	1.2	956	1.4	1012	1.3
Compute	**6163**	**9.2**	**6416**	**9.2**	**7332**	**9.4**
Compare	8952	13.3	8734	12.6	10945	14.1
Conditional Branch	10551	15.7	10632	15.3	10894	14.0
Unconditional Branch	6016	9.0	6602	9.5	6737	8.7
Call	8473	12.6	8776	12.6	8994	11.6
No-op†	0	—	0	—	1926	2.5
Control	**33992**	**50.7**	**34744**	**50.1**	**39496**	**50.8**
Total	**67104**	**100.0%**	**69377**	**100.0%**	**77769**	**100.0%**

Table 10: *VAX Instruction Use*

† The DEC VMS C compiler uses `nop` instructions to cause labels (i.e., branch targets) to be placed on even byte address boundaries. Often, these `nop` instructions immediately follow a branch instruction (the branch around the "else" clause of an "if-then-else"), so that they are never executed and incur no run-time penalty. Neither the Berkeley nor the

The first point of interest is the comparison between the VAX and the general RISC instruction usage. Although the actions of the various instructions are quite different on the two machines, the ratios of move instructions to compute and control instructions is similar on the two machines. This similarity means two things:

1) The applications and libraries used in this evaluation are a correct choice, since they produce similar results on two highly different architectures, or

2) The simple instructions on the R2000 are as adequate to the task as are the complex ones on the VAX.

To further address the second point, Table 11 examines the frequency of instruction usage on the VAX.

Percentage use of Instructions	**Number of Instructions**		
	Berkeley	*Tartan Labs*	*DEC (VMS)*
Never Used	117	101	111
< 0.05%	34	41	26
≤ 1.0%	44	48	54
≤ 2.5%	5	10	8
≤ 5.0%	3	5	7
≤ 7.5%	4	2	1
≤ 10.0%	0	0	1
≤ 15.0%	3	3	2
> 15.0%	0	0	0

Table 11: *VAX Patterns of Usage*

Some very interesting information now presents itself. Ignoring those instructions which are never used, or used only rarely, the frequency of instruction use very closely parallels that of the R2000 instruction usage.‡ What is most notable, however, is that of the 210 integer instructions on the VAX, over 100 are *never used* by either of the three compilers, and another 33 (on the average) are used < 0.05% of the time. This means that depending on the compiler, between 65% and 72% of the integer instruction set that is available to the VAX C compilers is *never used*, or used with such a small degree of frequency as to make one ask "why are these instructions present in the architecture?"

In addition to the 210 integer instructions, there are an additional 110 floating point instructions which are not being considered in our calculations. To be sure, some of the instructions which the compiler never or rarely generates are used only in an operating system context, but many are what would be expected to be generated. To be fair to the designers of the VAX, some of these instructions indeed have a function. There are some instructions that are specifically designed for FORTRAN or for PL/1, but even these perform functions that could very easily be executed using one or two other instructions.

One wonderful example of this is the `ediv` instruction, which calculates both the quotient and remainder for an integer division. Unfortunately, neither the Berkeley, Tartan Labs, nor the DEC (VMS) compiler take advantage of this fact, and use this instruction only for calculating remainders (the division being performed by other means). The Minsky exception principle says that the compilers should probably have some special case processing to recognize when both a quotient and remainder are being calculated, and use the `ediv` instruction for just this purpose. The fact is, however, that either none of the compiler teams thought of this special case, or that the level of effort required to implement it was sufficiently high to warrant not including it. In either case, much the complexity of the instruction set is unused for these and similar reasons.

The VAX also has a very large number of addressing modes. The great flexibility in addressing modes is considered one of the strong selling points of CISC architectures. However, when the frequency of use of these modes is examined, some questions arise as shown in Table 12.

Tartan Labs compiler uses this technique.

‡ The author would hasten to point out that the ordinate values in Tables 2, 5, 8, 11, 15, and 18 were chosen *prior* to knowing those on the abscissa. This is not a case of massaging the data to fit a curve – the data fits the curve all by itself!

† The VMS C compiler exhibits a substantially higher percentage of Register and Deferred (i.e., indirect) mode usage than do the other two compilers. This is readily understood when one realizes that the UNIX compilers use only registers R6-R11 for local (non-temporary) variables, while the VMS compiler is allowed to use registers R0-R11 for the same purpose. This extra allowance enables the VMS compiler to store addresses in registers instead of using the Relative addressing (i.e., memory direct) mode.

Address Mode	Example	*Berkeley* Count	*Berkeley* % used	*Tartan Labs* Count	*Tartan Labs* % used	*DEC (VMS)* Count	*DEC (VMS)* % used
Immediate	`$270`	4685	4.6	5105	4.9	4279	3.8
Literal	`$24 (n < 64)`	14942	14.8	15122	14.5	16449	14.4
Absolute	`$*label`	0	—	0	—	0	—
Absolute Indexed	`$*label[r4]`	0	—	0	—	0	—
Relative	`label`	37067	36.7	38692	37.0	32043	28.1†
Relative Indexed	`label[r4]`	526	0.5	464	0.4	221	0.2
Relative Deferred	`*label`	337	0.3	380	0.4	0	—
Relative Deferred Indexed	`*label[r4]`	0	—	58	0.1	0	—
Register	`r3`	25480	25.3	28684	27.5	36894	32.3†
Deferred	`(r3)`	2334	2.3	3256	3.1	9552	8.4†
Deferred Indexed	`(r3)[r4]`	97	0.1	44	—	568	0.5
Autoincrement	`(r3)+`	480	0.5	632	0.6	553	0.5
Autoincrement Indexed	`(r3)+[r4]`	0	—	0	—	0	—
Deferred Autoincrement	`*(r3)+`	0	—	0	—	0	—
Deferred Autoincrement Indexed	`*(r3)+[r4]`	0	—	0	—	0	—
Autodecrement	`-(r3)`	935	0.9	1290	1.2	1076	0.9
Autodecrement Indexed	`-(r3)[r4]`	0	—	0	—	0	—
Displacement	`24(r3)`	13059	12.9	9827	9.4	11568	10.1
Displacement Indexed	`24(r3)[r4]`	86	0.1	89	0.1	167	0.1
Displacement Deferred	`*24(r3)`	753	0.7	670	0.6	568	0.5
Displacement Deferred Indexed	`*24(r3)[r4]`	122	0.1	169	0.2	123	0.1
Total		**100903**	**100.0%**	**104482**	**100.0%**	**114061**	**100.0%**

Table 12: *VAX Addressing Mode Use*

Of the 21 addressing modes available on the VAX, 7 are *never used* by the Berkeley and Tartan Labs C compilers, 8 by the DEC C compiler. Another 9 are used (on the average) less than 1% of the time, so that simulating their actions through a combination of other instructions and modes would be worthwhile. In fact, when we examine the addressing modes that are used most frequently (namely Immediate, Literal, Relative, Register, and Displacement), we find that the modes correspond exactly to those available on the simpler RISC instruction set of the MIPS (Immediate and Literal modes on the VAX are identical but for the length of the operand, and correspond to the MIPS Immediate mode). Considered together, the usage of these five simple modes comprises between 96.4% and 97.8% of all the addressing mode usage on the VAX!

Why then are all of these addressing modes present, if they are hardly ever used, and when they can be simulated using other modes? The VAX was supposedly designed with the help of compiler writers. With all the superfluous instructions and addressing modes, one is inclined to ask "What happened?" There can be no answer to this question – all of the fancy addressing modes of the VAX are superfluous, and need not be present in an architecture at all. Compiler technology has only gotten better over the last 10 years, so one can only conclude that these frills were never necessary.

Before we leave the VAX, let us examine one more point, namely that of the 3 operand instructions. Many VAX instructions allow the programmer to specify three operands instead of two, so that the high level instruction `a = b + c` may be coded as:

```
addl3 b, c, a
```

instead of the slightly more cumbersome:

```
movel b, a
addl2 c, a
```

Many compilers go to great length to try to use this 3 operand mode, since it results in smaller and more efficient code. However, in spite of all this effort on the part of the compilers, the 3 operand mode is greatly underutilized as shown in Table 13.

Number of Operands	Berkeley		Tartan Labs		DEC (VMS)	
	Count	% used	Count	% used	Count	% used
2 Operand	29155	91.0	30233	91.0	33884	92.0
3 Operand	2870	9.0	3004	9.0	2930	8.0
Total	**64576**	**100.0%**	**66689**	**100.0%**	**72913**	**100.0%**

Table 13: *2 Operand vs.* **3 Operand Addressing**

As can be seen, the 3 operand mode is used (at best) only 9.0% of the time. The instruction logic necessary to distinguish between these modes is made unnecessarily complicated (and slowed) by requiring it to handle addressing modes that are rarely used. To be sure, the 2 operand mode requires that more instructions be executed. The whole machine is slowed by its complexity. If the architecture only had 1 and 2 operand modes, then the whole system would run faster (having one less mode to decode), and only in 4.5% of the instructions (at worst) would extra work be incurred. We feel that the payoff is on the side of simplicity, and not on that of complexity.

4.2. Analysis of MC68020 C Compilers

The MC68020 is another classic CISC architecture that is widely used in the industry, and especially in the UNIX marketplace. It differs from the VAX in that the 68020 has specialized address and data registers, while the VAX has general registers which can be used for either purpose. Other than this primary difference, it shares with the VAX the ability to directly address both memory and registers from most instructions, and the large number of addressing modes and instruction types.

We were able to evaluate two different compilers for the 68020. In this case the compilers were:

1) The Sun 680x0 compiler (operating system version 3.5), and
2) The Gnu 680x0 compiler (version 1.31).

The results obtained from the two compilers (both with the 68020 code generation option enabled) were again similar. As with the VAX, the individual code idioms and instruction counts varied, although the general patterns of instruction and addressing mode usage (as seen in Tables 14 through 16) was consistent across the two compilers.

Instruction Class	Sun		Gnu	
	Count	% used	Count	% used
Move	**41383**	**46.3**	**39518**	**46.6**
Arithmetic	12566	14.1	10527	12.4
Logical	1386	1.6	1370	1.6
Compute	**14021**	**15.7**	**12011**	**14.2**
Compare	9281	10.4	9050	10.7
Conditional Branch	9950	11.1	10385	12.3
Unconditional Branch	5844	6.5	5561	6.6
Call	8856	9.9	8226	9.7
Control	**33931**	**38.0**	**33222**	**39.2**
Total	**89335**	**100.0%**	**84751**	100.0%

Table 14: *68020 Instruction Use*

Examining the instruction class coverage, we see patterns that are quite similar to the other processors and compilers in the test suite. This similarity of use of the various instruction classes adds to our belief that our methods and test suite were valid.

When we examine the pattern of instruction use, we find that as with the VAX, the 68020 compilers are unable to effectively utilize the instruction set of the target architecture.

Percentage use of Instructions	**Number of Instructions**	
	Sun	*Gnu*
Never Used	87	89
< 0.05%	13	11
≤ 1.0%	22	24
≤ 2.5%	7	3
≤ 5.0%	7	9
≤ 7.5%	1	2
≤ 10.0%	2	1
≤ 15.0%	0	0
> 15.0%	1	1

Table 15: *68020 Patterns of Usage*

Without considering the floating point co-processor instructions, of the 140 instructions listed in the MC68020 instruction set, 88 (on the average) of the instructions are *never used* by the compilers, and another 12 (on the average) are used < 0.05% of the time. This means that roughly 71% of the 68020 instruction set is *not used* by either the Sun or the Gnu C compiler. Of course, some of these "unused" instructions are used only in an operating system context (i.e., return from trap or test-and-set instructions). Nevertheless, the very large fraction of unused or unusable instructions seems to indicate that the 68020 instruction set architecture is overly complicated. The 68030 introduces even more addressing modes – we wonder how, or even whether they will be used by compilers.

		Sun		*Gnu*	
Address Mode	**Example**	**Count**	**% used**	**Count**	**% used**
Immediate	`#270`	18911	13.9	17031	13.0
Absolute	`label`	40042	29.4	35352	27.0
Register	`d3`	48901	35.9	52905	40.4
Register Indirect	`a3@`	5449	4.0	8740	6.7
Postincrement	`a3@+`	510	0.4	1229	0.9
Predecrement	`a3@-`	6062	4.5	8019	6.1
Displacement	`a3@(4)`	14952	11.0	6351	4.8
PC Displacement	`pc@(4)`	0	—	0	—
Indexed	`a3@(4,d3:w:2)`	1076	0.8	1237	0.9
PC Indexed	`pc@(4,d3:w:2)`	152	0.1	150	0.1
Memory Postindex	`([4,a3],d3:w:2,300)`	0	—	0	—
Memory Preindex	`([4,a3,d3:w:2],300)`	0	—	0	—
PC Memory Postindex	`([4,pc],d3:w:2,300)`	0	—	0	—
PC Memory Preindex	`([4,pc,d3:w:2],300)`	0	—	0	—
Total		**136055**	**100.0%**	**131014**	**100.0%**

Table 16: *68020 Addressing Mode Use*

The most notable difference in addressing mode usage was that the Gnu compiler is a little more clever in its use of register based address modes. It also seems to keep better track of addresses in registers, and consequently is able to use the register indirect, predecrement, and postincrement modes with greater facility.

However, of the 14 addressing modes available on the 68020, the three modes which are used the most frequently are (again): immediate, absolute, and register. Displacement mode and register indirect mode (the latter being a special case of the former, with a zero displacement) fill in the remainder of the main usage of the addressing modes. The more complicated modes involving indexing are used rarely or not at all. While we can see their occasional utility, we feel that the architecture would be better off without them. Again the question arises, "if simple instructions and address modes can perform the same functions as complex ones, and if the very complex functions of CISC architectures are rarely used, why are they present, instead of allowing the compiler to generate a sequence of simple instructions?"

4.3. Analysis of 80386 C Compilers

The 80386 is another yet classic CISC architecture that is widely used in the industry, in both the UNIX and PC marketplace. The notable features of this processor are a baffling array of addressing modes, and the fact that the specialization of registers in the 80386 is even stranger than that in the 68020.

We were able to evaluate two different compilers for the 80386. In this case the two compilers were:

1) The Sun 80386 compiler (operating system version 4.0), and
2) The Gnu 80386 compiler (version 1.31).

The results obtained from the two compilers were again similar. As with the other two CISC architectures, the individual code idioms and instruction counts varied, but the general patterns of instruction and addressing mode usage (as seen in Tables 17 through 19) was consistent across the two compilers.

	Sun		*Gnu*	
Instruction Class	**Count**	**% used**	**Count**	**% used**
Load	6891	7.4	9034	11.7
Store	14521	15.5	13172	17.1
Shuffle	27310	29.2	18585	24.1
Move	**48722**	**52.1**	**40791**	**53.0**
Arithmetic	7933	8.5	4173	5.4
Logical	2801	3.0	1508	2.0
Compute	**10734**	**11.5**	**5681**	**7.4**
Compare	9596	10.3	8753	11.4
Conditional Branch	10018	10.7	9131	11.9
Unconditional Branch	5860	6.3	4881	6.3
Call	8534	9.1	7737	10.1
Control	**34008**	**36.4**	**30502**	**39.6**
Total	**93464**	**100.0%**	**76974**	**100.0%**

Table 17: *80386 Instruction Use*

Examining the instruction class coverage, we see patterns that are quite similar to the other processors and compilers in the test suite. This similarity of use of the various instruction classes adds to our belief that our methods and test suite were valid.

When we examine the pattern of instruction use, we find that as with the other two CISC machines, the 80386 compilers are unable to effectively utilize the instruction set of the target architecture.

Percentage use of Instructions	**Number of Instructions**	
	Sun	*Gnu*
Never Used	105	106
< 0.05%	6	4
≤ 1.0%	18	21
≤ 2.5%	6	6
≤ 5.0%	7	4
≤ 7.5%	1	0
≤ 10.0%	1	2
≤ 15.0%	0	0
> 15.0%	2	2

Table 18: *80386 Patterns of Usage*

Ignoring the floating point co-processor instructions, of the 146 instructions listed in the 80386 instruction set, 105 of the instructions are *never used* by the compilers† and another 5 (on the average) are used <

† As with the other CISC architectures, earnest attempts were made to portray the 80386 instruction set in a fair light. To this end, many different instruction mnemonics were considered to be identical for instruction counting purposes. Examples of this consolidation include grouping the `jcc` (near) and `jcc` (short) instructions, the `cmps`, `cmpsb`, `cmpsw`,

0.05% of the time. This means that roughly 75% of the 80386 instruction set is *not used* by either the Sun or the Gnu C compiler – the worst showing of any architecture reviewed for this paper. As with the other two CISC architectures, some of these "unused" instructions are used only in an operating system context (i.e., return from trap or bit test-and-set instructions). Nevertheless, the very large fraction of unused or unusable instructions seems to indicate that the 80386 instruction set architecture suffers from the same overcomplicated design that the other CISC machines do.

		Sun		*Gnu*	
Address Mode	**Example**	**Count**	**% used**	**Count**	**% used**
Immediate	`$76`	19396	14.2	14838	13.4
Register	`%ebp`	62042	45.3	47003	42.5
Register Indirect	`*%eax`	4	—	32	—
Offset	`12(%ebp)`	20041	14.6	17371	15.7
Offset Indirect	`*12(%ebp)`	16	—	0	—
Absolute	`_foo`	34050	24.9	30098	27.2
Indirect	`*_foo`	13	—	0	—
Indexed	`(%eax,%edx)`	229	0.2	343	0.3
Indexed Offset	`4(%eax,%edx)`	14	—	72	0.1
Scaled Indexed	`(%eax,%edx,4)`	697	0.5	479	0.4
Scaled Indexed Offset	`4(%eax,%edx,4)`	1	—	132	0.1
Based Indexed	`_foo(,%edx,4)`	396	0.3	268	0.2
Indirect Indexed	`* foo(,%edx,4)`	57	—	52	—
Total		**136956**	**100.0%**	**110688**	**100.0%**

Table 19: *80386 Addressing Mode Use*

It was not clear exactly how we should describe and partition the 80386 addressing modes. One way of describing them is that there are a few modes with many optional components. Another is to attempt to describe each of the options as a separate addressing mode. The compromise description which we arrived at was to simply list all of the addressing modes that either of the two compilers used, and simply make the parenthetical remark that there are many more addressing modes possible on the 80386 than are shown in Table 19.

It should come as no surprise that the predominant addressing modes used by the 80386 compilers are (yet again): immediate, register, absolute, and offset. Combined, these four modes account for 99% of all instruction addressing in the sample code suite.

5. Comparing Object Code Size

What we have seen so far is that CISC instructions and addressing modes are used inefficiently, and that RISC architectures are used much more efficiently. The question then arises "if RISC architectures require multiple instructions to execute what is done in a single CISC instruction, won't the program size on a RISC architecture be concomitantly larger?" The answer is quite surprising. Comparing the total instructions in Tables 1, 4, 7, 10, 14, and 17, we see that the number of instructions on the RISC architectures is generally much larger than the CISC architectures. This might lead us to believe that the RISC architectures are less efficient in storing programs. However, the *number* of instructions is not the true measure of program size. Rather, it is the number of *bytes of memory* that the instructions occupy that is of concern.

Each instruction on the R2000, SPARC and MC88100 occupies 4 bytes of memory, and generally executes in a single machine cycle.† Although each instruction on the VAX typically occupies a single byte in memory (the `g` and `h` format floating point instructions take two bytes each), the operands for the instructions may require many bytes each. Each operand requires at least one extra byte of memory, and some complex addressing modes on the VAX require an extra six bytes to store. The execution speed on a

and `cmpsd` instructions, all of the different `call` instructions, rotate instructions, shift instructions, etc. Without this consolidation, the instruction count (and the fraction of unused instructions) for the 80386 would be much larger.

† Some instructions, such as the multiply or divide instructions of the R2000, always take multiple machine cycles. On the 88100 and the SPARC, instructions can take multiple cycles when the scoreboard register indicates a need to "stall" the pipeline. These instructions are the exception, and not the rule.

VAX depends on the instruction, but generally the more complex an addressing mode, the longer the execution time. On the 68020, each instruction occupies two bytes of memory. Depending on the instruction and the addressing mode, between zero and four extra bytes of memory are needed per operand. The 80386 instructions are either one or two bytes long, the complex addressing modes can take as many as ten extra bytes. The instruction execution speed in the 80386 varies considerably, and can range from one clock cycle to well over a hundred for some types of procedure call instructions.

When we compare program size in bytes (as in the following table), we see some surprising results.

Architecture	Compiler	Number of Instructions	Size in Bytes	Fraction of VAX *pcc*	Bytes per Instruction
R2000	MIPS	127108	508432	1.70	4.00
88100	Green Hills	106192	424768	1.43	4.00
	Gnu	80834	323336	n/a†	4.00
SPARC	Sun	98694	394776	1.33	4.00
	Gnu	68748	274992	n/a†	4.00
VAX	Berkeley	67104	297928	—	4.43
	Tartan Labs	69377	313384	1.05	4.51
	DEC (VMS)	77769	280876	0.94	3.61
68020	Sun	89335	337024	1.13	3.77
	Gnu	84751	248828	0.83	2.78
80386	Sun	93464	??‡	??‡	??‡
	Gnu	76794	??‡	n/a†	??‡

Table 20: *Program Size Differences*

As can be seen, the difference in program size is not so pronounced when we measure the actual number of bytes that the program occupies in memory. To be sure, the 30–60% increase in program size on the RISC architectures is of some concern. However, there are a number of issues to consider other than just instruction size.

1) The size of the instructions does not account for the total size of the program – statically and dynamically allocated data must also be considered. By examining the contents of the directories */bin*, */usr/bin*, */usr/ucb*, and */etc* on a μVAX, it was found that the number of bytes of instructions (i.e., *text* size) accounted for only 39% of the total number of bytes (i.e., *text+data+bss*) in the program. The percentage increase in program size becomes even smaller when dynamically allocated data (e.g., that allocated with *malloc*) is counted in the total.

2) Some compilers (e.g., the MIPS R2000 compiler) perform routine inlining and interprocedural optimizations. These techniques can lead to larger executable images, which nonetheless execute much faster than those generated without this technique. All compilers used for this paper were run with the highest levels of optimization available, and should not be penalized for generating efficient code that happens to be larger than inefficient code.

3) Most UNIX systems, especially the newer RISC architectures, have demand paged executable files, so an increase in program size is not necessarily reflected in a larger memory image.

4) In spite of the extra instructions that need to be executed, RISC processors execute programs much faster than comparably priced CISC processors (the MIPS M/500 executes integer applications 6-10 times faster than the μVAX; the more recent models – the M/1000 and M/2000 – perform even better).

† Because the Gnu compilers for the SPARC, 88100, and 80386 were untested, and because they occasionally dumped core and were unable to completely process some files, the ratio of code sizes between these compilers and *pcc* are not reported in this table.

‡ Unfortunately, we were unable to determine the size of the 80386 object code.

5) Memory density is increasing and memory costs are decreasing at breakneck rates. People nowadays think nothing of a μVAX personal workstation with 8 Mbytes of memory, when only 10 years ago some of the most powerful timesharing mainframes had only 1 Mbyte of memory. Any extra cost incurred in memory with the new RISC processors is well worth the benefit of increased throughput that the faster RISC processors provide.

As an aside, it is also interesting to note that the DEC (VMS) compiler produces smaller code than either the Berkeley or Tartan Labs compiler, in spite of having a higher instruction count. This is accomplished through an efficient use of the smaller sized addressing modes of the VAX.

6. What do the Results Tell Us?

Tables of figures are interesting in their own right, but what this paper sought to do was to provide *insight*, and not just statistics. A number of things can be inferred from the tables. The first is that within each architecture (where multiple compilers were evaluated), the instruction coverage and address mode coverage of the different compilers did not vary substantially. This tells us that although the different compilers were written by completely different groups, who at times had completely different goals (e.g., speed *vs.* space optimization), the overall functionality of the instruction set architecture was utilized in the same way by each group.

The addressing mode usage was nearly identical for every compiler within an architecture. Although they were not called out explicitly in any of the tables presented here, the individual instruction coverage (and not just the instruction class) was also very nearly the same within an architecture. From this statistic we may infer that if a feature of the instruction set architecture was valuable (i.e., a particular instruction or addressing mode), it was used. Conversely, if it was not valuable, it was not used.

The second point worthy of note is an observation that crosses architectural boundaries. Although the actual instructions differ between architectures, the use of the different *classes* of instructions (i.e., move, compute, and control) is also very similar.† This indicates that the test suite that was selected was a valid one – that although the programs and hardware varied, the compiler's use of the hardware for the programs was consistent.

Finally, when we examine the difference between the RISC and CISC architectures in this paper, we observe three things.

1) The reduced instruction sets are as adequate to the task of implementing (in assembly language) the test suite as are complex instruction sets,

2) The size of the resultant programs on the RISC architectures is not substantially larger than that on the CISC architectures, and

3) Generally speaking, this study showed that roughly 70% of RISC instructions are used by the compilers and 30% are not. For CISC compilers, the statistic is reversed – roughly 30% of the instructions are used, and 70% are not.

4) The "extra" instructions available on the CISC architectures are simply not used, and provide frills that cannot be taken advantage of by the compilers considered in this evaluation.

To be fair, we must point out that the tests only evaluated C compilers, and did not examine FORTRAN, Pascal, or Ada. It is entirely possible that compilers for these languages might generate slightly different code (especially in the case of Ada and Pascal, which would almost certainly use the "bounds check" instructions – the `chk` and `chk2` instructions of the 68020, the `tbnd` instruction of the 88100, and the `bound` instruction of the 80386 – to test the validity of array indices). However it is also the case that many compiler writers (Berkeley, MIPS, and Tartan Labs included) use the same (or substantially the same) code generator for different language front ends. In these cases, we feel that it would be unlikely that different language compilers would produce substantially different results than those we have seen here.

† The slight deviations (i.e., between VAX and other architectures) can be attributed to different number of registers (requiring more data motion where fewer registers are available), and to the difficulty of classifying an instruction such as "subtract one and branch if less than zero". We classified it as a conditional branch instruction, but it also could be considered a compute or comparison instruction.

7. Conclusions

In all the machines examined in this report, the more complicated the instruction set architecture, the less utilized were the features of that architecture. While attractive to assembly language programmers, the complex features of CISC architectures are simply not used by compilers. Since the great majority of programmers write in high level languages (and not in assembly language), we feel that new architectures should be kept simple, to allow compilers to make full use of their features.

It has been amply shown in the past few years that RISC architectures execute faster than comparably priced and comparably sized CISC architectures. With the advent of GaAs technology, this speed differential will become even more pronounced. Since GaAs technology is currently limited by the size of the chips that can be produced, RISC becomes even more attractive, as it requires a smaller "footprint" to implement.

The term *reduced* has in no way implied *restricted*, nor has it caused the major increases in code size that CISC proponents claim will occur to support their cause. The slight increase in program size on the RISC processors is more than offset by a substantially faster execution speed. If UNIX hopes to be the system of tomorrow in addition to that of today, manufacturers of UNIX systems should concentrate their efforts more on RISC machines, than on CISC machines.

8. Acknowledgments

I would like to gratefully acknowledge the invaluable assistance of Tony Birnseth, Mike O'Dell, and John Devitofranceschi for gathering statistics on compilers to which I did not have direct access, and to Robert Firth for his usual scathing (and technically brilliant) suggestions and commentary.

9. Bibliography

Mario Barbacci, William Burr, Samuel Fuller, and Daniel Siewiorek, *Evaluation of Alternative Computer Architectures*, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, February 1978, Technical Report no. CMU-CS-77-EACA

C. Gordon Bell, "RISC: Back to the future?," *Datamation*, 32(11), June 1986

Mark Himelstein, *et. al.*, "Cross Module Optimization: Its Implementations and Benefits," In *Usenix Conference Proceedings*, June, 1987

Daniel Klein and Robert Firth, *Final Evaluation of MIPS M/500*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1987, Technical Report no. CMU/SEI-87-TR-25

Veljko Milutinovic, *et. al.*, "Architecture/Compiler Synergism in GaAs Computer Systems," *IEEE Computer*, 20(5):72-93, May 1987

Marvin Minsky, *The Society of Mind*, Simon and Schuster, NY, 1986, p.127

David Patterson, "Reduced Instruction Set Computers," *Communications ACM*, 28(1):8-21, January 1985

Daniel Sieworick, C. Gordon Bell, and Alan Newell *(eds.)*, *Computer Structures: Principles and Examples*, McGraw Hill, New York, NY, 1982

UNIX Assembler Reference Manual, AT&T Bell Laboratories, Holmdel, NJ, 1979

VAX Architecture Handbook, Digital Equipment Corporation, Maynard, MA, 1981

MC68020 32-bit Microprocessor User's Manual, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985

Assembly Language Programmer's Reference Guide, MIPS Computer Systems, Inc., Sunnyvale, CA, 1986

The SPARC Architecture Manual, Sun Microsystems, Inc., Mountain View, CA, 1987

MC8810 User's Manual, Motorola Microprocessor Group, Austin, TX, 1988

80386 Programmer's Reference Manual Intel Coporation, Santa Clara, CA, 1986

On the Evaluation of the Performance of RISC Systems

Kurt P. Judmann

Technical University of Vienna

ABSTRACT

Modern computer systems from the PC-level up to Supercomputers claim to gain performance from using of a "Reduced Instruction Set". From the users and system analysts point of view it is hardly visible whether increased performance over other systems results from a reduced instruction set or from other design features. Such features can be cache hierarchies, parallel processing, pipelining or different implementation of the memory interface. This paper describes a model which can be used to calculate an average instruction execution time for instruction groups, and mixes of instruction groups of a given processor. It allows comparisons of different processors by using a code substitution method and helps in evaluating the effects of the different architectural features used in RISC systems. Although the model can be used for processors in general it is tailored to describe the techniques used in RISC processors.

1. Introduction in RISC concepts

Although there is no definition of the concept of Reduced Instruction Set Computers one can find certain common strategies in the processors which claim to be of RISC nature. Since the concept calls for a reduction of the instruction set it is obvious that the number of instructions in such a processor is limited. This does not necessarily mean that the semantic level of the instructions, which is also frequently characterized by the term complexity of the instructions needs to be lower than in other processors. However the basic RISC concept which is demonstrated in the development of the Berkeley RISC I and the Stanford MIPS uses both the reduction of the number and of the complexity of the instructions. The goal is to have a simple instruction set which can be executed fast by a relatively simple control unit. The gain in speed achieved can be higher than the loss caused from the possible necessity of executing more instructions to solve a given problem. Following this idea one could see Reduced Instruction Set Computers in a way that they represent a subset of the Complex Instruction Set Computers but simply run on a faster clock cycle. RISC Processors however take more advantage of other effects than of the simple reduction of the instruction set. These effects include mostly the effective use of pipelining within the CPU and the implementation of large register files on the CPU chip. Examples for these basic RISC concepts are the Berkeley RISCs [Dit80a, Pat82a, Pat85a], MIPS [Hen82a] and SRISC [Jud87a]. In other RISC concepts like Transputers one may find communication support which is targeted to support multiprocessing as well as floating point support. Modern microprocessors like the M88100 or the Intel i860 which claim to be RISC processors combine standard CISC features with basic RISC ideas like fixed instruction length, pipelining or register files on chip. Therefore these processors could also be seen as even more complex CISCs with some simplifications in regard to the number and complexity of the instructions. Therefore basic RISC processors which have few and simple instructions are furthermore called "Simple" Reduced Instruction Set Computers. Processors having either few but dedicated and complex instructions or more instructions above the basic semantic level are called "Complex" Reduced Instruction Set Computers.

2. The execution time of a RISC instruction cycle

2.1. Von Neumann type instructions

Current RISC systems still follow some of the most important principles of the von Neumann concept [Bur63a]. These principles describe a main storage which contains the instructions which are at least partially stored in the same sequence in which they are to be executed. They have an address reference included in the instructions which refers to the data to be processed. Because of the fact that RISC processors tend to have bandwidth problems at the CPU – Memory interface some follow the so called "Harvard Architecture" with separated memories for data and instructions. This can be considered as an extension to the "von Neumann" concept. According to the timing sequence in von Neumann machines one can split an execution cycle into three main parts:

- Instruction fetch
- Operand fetch
- Execution

2.2. Instruction cycles of RISC processors

Although following general von Neumann rules RISC processors generally do have some of the listed extended features:

- Three address architecture within the register file, one address architecture when referencing the main memory. (Frequently restricted to Loads and Stores.)
- Harvard Architecture
- Integrated Cache memories for instructions and data
- Pipelining within the CPU with a fixed pipeline cycle aimed at synchronous operation of the pipeline
- Special multi-register management instructions on a built in register file (window management).

For the purpose of calculating the execution time of a cycle, its actions are described separately as related to executing hardware units like ALUs, interfaces, data paths, etc. This enables the modeling of a given architecture which is partially reflected in the hardware layout.

The execution time of any instruction is represented in (1):

$$ti = tf + ta + + te + ts \tag{1}$$

tf .... instruction fetch
ta .... address calculation
to .... operand fetch
te .... instruction execute
ts .... result storage

To be able to build a relationship between the execution time of the different stages of instruction execution and a program, each of the stages is described using parameters which can be derivated by measurements or calculation for instructions and groups of instructions. The importance of the execution time of a single instruction or a group of instructions for a given application can also be determined by measurements or statistics so tha an average instruction time including all instructions with weight factors can be calculated.

To represent the execution time of a simple RISC processor we use the SRISC [Jud87a],which has a Load/Store architecture, Harvard architecture and the usual register file with window management. In a first step the effect of the four stage pipeline is not considered to show the effect of pure reduction of instructions.

The following description of the execution phases can be made:

$$tf = tm \tag{2}$$

tm .... main memory access time

$$ta = tc * (po1 + po2) \quad (3)$$

tc .... CPU base cycle time
po1, po2 .... probability of the presence of operands 1 and 2

$$to = (pr * tc + (1-pr) * tm) * (po1 + po2) \quad (4)$$

pr .... probability of working on an operand contained in a CPU register

$$te = tc \quad (5)$$

$$ts = pr * tc + (1-pr) * tm \quad (6)$$

For any given instruction poi is evident, for an instruction mix it is the weighted average of the instructions within the group.tm and pr describe the given environment.

2.3. Pipelined execution

In a pipelined processor the execution cycle is divided into phases which are independent from each other by using different hardware resources within the processor and the environment for execution. These phases are executed in parallel for subsequent instructions as described in detail for RISC I in [Pat85a]. If a time frame is chosen so that all phases can be executed within the same time slot, the pipeline is working synchronously. SRISC for example splits the execution cycle into four phases which can be executed within the pipeline cycle time. Considering full synchronism the execution time for an average instruction equals the phase cycle time tp, and is four times lower than in an equivalent non pipelined processor. Pipelines however are mostly designed in a way that most, but not all cycle activities can be performed within the cycle time. Activities which take much longer to execute than the average action, like extensive address calculation or activities which cause a hardware conflict with other pipeline stages, cause the execution of an extra cycle and a delay. There are even more reasons for delaying a pipeline which can be found in software dependencies [Dit80b]. For SRISC the average execution time of an instruction can be described as follows:

$$ti = tp + (po1 + po2) * (tm * (1-pr)) + 3 * pj * tm + pd * tp \quad (7)$$

The first term represents the synchronous cycle.

The second term represents the fact that SRISC will delay the pipeline for one memory cycle, if an operand is accessed in the main memory.

The third term reflects the so called instruction dependency with the probability pj for the performance of a jump instruction which causes the reorganization of the pipeline.

The fourth term reflects the so called data dependency with the probability pd for the occurrence of an ALU instruction using a result which is produced by the previous instruction. This causes a delay of one pipeline cycle to correctly pass the result to the next instruction.

The delay caused by terms 2 and 3 can be eliminated either by software regrouping or the implementation of additional hardware features in the architecture like a cache memory and a second prefetch buffer for the alternative jump sequence.

2.4. Cached memories

Pipelined RISC processors execute one instruction every pipeline cycle. With higher processor clock speeds on one hand and lower memory access speed due to multiprocessor capabilities of modern processors on the other, the use of cache memories becomes essential. According to the cache organisation one would have to distinguish between cache hits and misses in tm and also between instruction and data accesses. Concepts for the organisation of caches are described in detail in [Smi82a].

3. Performance evaluation using instruction groups

Performance analysis based on instruction groups requires knowledge about the statistical instruction usage of a given application on one hand and a model for calculating the execution time for average instructions as described in this paper on the other. Once one has focussed on an instruction mix which is believed to represent a specific task, the average execution time for each group in the mix and finally for the whole mix can be calculated. One could either use common instruction mixes like the Gibson Mix [Gib70a] or define

ones own mix as well. To be able to compare different processors and even different system architectures one would still be able to use instruction mixes and groups when modeling the executing hardware according to chapter 2. The instruction mix however has to be uniform for the processors to be compared, so that the need appears to include groups which a given processor may not have implemented. Common samples are floating point operations on simple RISCs or register file manipulation in CISC processors. Since one is not able to model these instructions in some processors the average execution time has to be represented through constructions using other instructions (subroutines).

4. Results

4.1. Instruction Mixes

Using a specific mix which distinguishes between transfer operations, jumps, arithmetic operations, test operations and input output, the motorola M68020 and the SRISC processor have been compared. For Pascal and C code which has been generated with compilers the following instruction group usage and code substitution factors have been found:

Group	p of group usage	substitution factor SRISC/CISC
Transfer	0.41	0.6
Jump	0.30	1.1
Arithmetic	0.18	1.6
Test	0.09	2.1
I/O	0.02	1.2

This gives an overall factor for the average code length of a SRISC program compared with a M68020 program of:

Code SRISC/M68020 = 1.03

It appears that an average SRISC program is practically of equal length as the corresponding CISC program.

4.2. Execution time

Using the simple model developed in 2.2 one can easily evaluate the effect of reducing the instruction set and implementing other features such as a large register file. Provided both SRISC and M68020 run on the same clock cycle the relationship between the execution times is:

tSRISC/tM68020 = 2.1

This shows that, ignoring the pipeline effect and not using a cache with the SRISC the M68020 is double as fast as SRISC.

Considering the more detailed model and the four stage pipeline as well as modeling a simple cache memory with a hit rate of 60 % the relationship between the execution times is:

tSRISC/tM68020 = 0.23

This shows that the gain af a factor 4 the RISC provides over the CISC is mostly due to the undisturbed operation of the pipeline. This effect is lately also brought into CISCs with the sacrifice of some addressing modes and some complex instructions.

5. Further investigations and designs

The advantage of evaluating processors by representing cycles according to their execution hardware can be found in the fact that the architectural influences on the system level can be investigated separately from the software and application issue. Current work focuses on modeling cache hierarchies for multiprocessor systems, parallel processing as well as on the design of such systems.

References

[Hen82a] J. Hennesey *et al.*, "The MIPS Machine," *Proceedings of the Spring Compcon 1982*, pp. 2–7, IEEE, Feb 1982.

[Bur63a] Burks, Goldstine, and von Neumann, *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*, pp. 34–79, McMillan, New York, 1963. In A. H. Taub: Collected Works of John von Neumann

[Dit80a] D. R. Ditzel and D. A. Patterson, *Retrospective on High Level Language Computer Architecture*, pp. 97–104, 1980.

[Dit80b] D. R. Ditzel and D. A. Patterson, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News*, vol. 8, no. 6, pp. 25–33, 1980.

[Gib70a] J. C. Gibson, "The Gibson Mix," IBM Technical Report TR-00.2043, June 1970.

[Jud87a] Kurt P. Judmann, "On the Influence of the Reduction of the Instruction Set on the Architecture of Microprocessors," Ph.D. Thesis, Technical University of Vienna, 1987.

[Pat82a] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE Computers*, no. 9;, pp. 8–21, 1982.

[Pat85a] D. A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, vol. 28, pp. 8–21, January 1985.

[Smi82a] Alan J. Smith, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, Sept 1982.

SPARC – Scalable Processor Architecture

Dr. Martin Lippert

Sun Microsystems GmbH
Bahnhofstrasse 27
D–8043 Unterfoehring
mlippert@sunmuc.uucp

ABSTRACT

This paper gives an overview over the technical concept of a new RISC architecture, called SPARC, developed by Sun Microsystems. SPARC stands for Scalable Processor ARChitecture – the design specification is published and licensable. Sun has licensed SPARC processors to several semiconductor vendors and over 50 computer manufacturers.

Its well known that RISC based microprocessors offer substantially more cpu power than systems with processors based on a traditional architecture with complex instruction sets. The term "scalable" refers to the size of the smallest lines on a chip. As lines become smaller, chips get faster. However, some chips designs do not shrink well – they do not scale properly – because the architecture is too complicated. Because of it's simplicity, SPARC scales well. Consequently, SPARC systems will get faster as better chip–making techniques are perfected.

In combination with efforts of unifying UNIX, SPARC based systems will allow binary compatibility among systems from different vendors like in the PC world today.

1. Introduction

This paper covers the evolution of RISC and the ideas behind SPARC very briefly in the beginning.

After this introduction an overview over the technical concept of the new RISC architecture, called SPARC, is presented. SPARC is an acronym for Scalable Processor Architecture, which was developed by Sun Microsystems between 1984 and 1987.

After that some competitive information is provided together with the status, what of SPARC's vision is reality today.

2. Evolution of RISC

The term RISC entered the lexicon in 1980, but the earliest designs similar to today's RISCs were done by Seymour Cray in the late 1960s for supercomputers. More recently, John Cocke led a group at IBM in the early 1970s that worked on a simple architecture for nonscientific codes that resulted in the 801 minicomputer. This minicomputer was not very successful as a product. So RISC principles were superceeded by other designs for quite a while.

Current RISC concepts didn't crystalize until 1980 or 1981. Back then, UC Berkeley professor Dave Patterson and his students were developing RISC microprocessors, while at Stanford, John Hennessy was also working in RISC.

This historical vignette comes from RISC pioneer Patterson, who served as a consultant on Sun's SPARC project. Sun borrowed from Patterson's work at Berkeley. The emphasis was on a simple architecture on a single chip. They were particularly interested in UNIX and C because Bill Joy was working on Berkeley UNIX at the same time.

At Stanford, Hennessy was advancing the state of the art of compilers, with particular emphasis on Pascal, while also building a VLSI chip. Although UNIX and C predominate in RISC, the IBM, Berkeley, and Stanford research led to commercial machines based on RISC processors. See Figure 1.

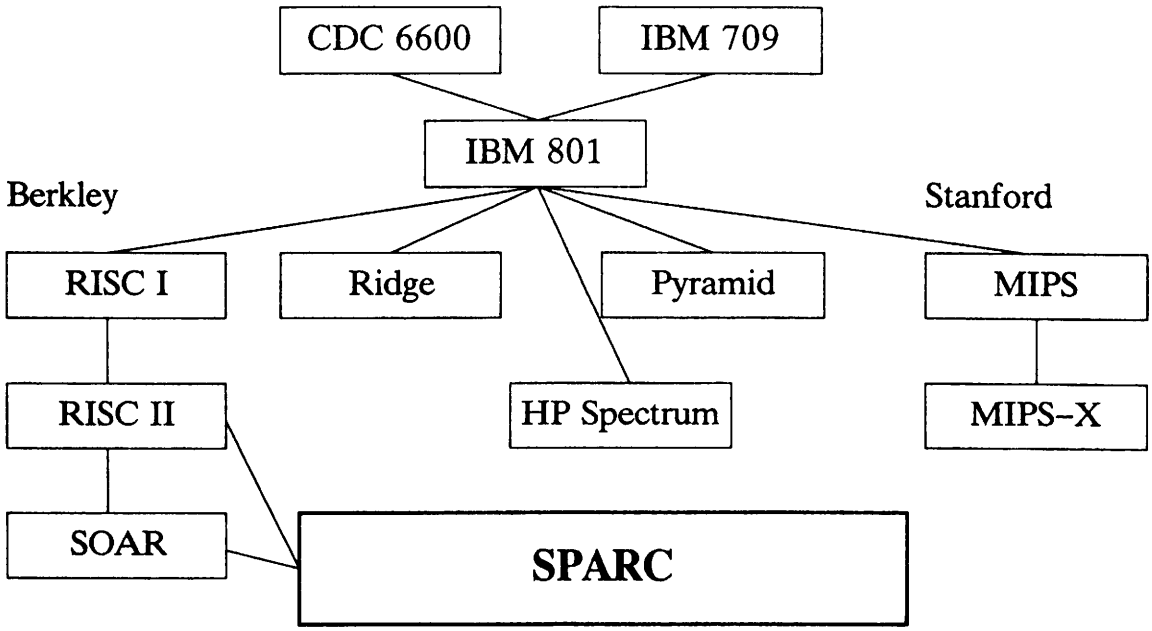


Figure 1: *RISC Architectures*

3. The SPARC Vision

In 1964 Intel founder Gordon Moore was at a conference. Someone asked him how many transistors it would be possible to put on a chip. Moore said that the answer to that question was a formula: 2 raised to the power of the current year minus 1964. For example, in 1980, the number of transistors on a chip was 65,000. The formula became known as "Moore's Law" and it held true for about 20 years. Twenty years later, Bill Joy was at a conference and predicted that the number of MIPS that could be squeezed from a microprocessor in a given year would be "2 to the current year minus 1984".

This is, in other words, the SPARC vision. See Figure 2.

SPARC is a RISC microprocessor architecture, that is believed to be a leading standard for high performance computing in the 90s. Due to the scalable architecture multiple binary–compatible SPARC processors can be made by using different semiconductor process technologies, such as Gate Array, CMOS, ECL or based on GaAs or superconductors.

SPARC processors will cover a broad range of performance points according to semiconductor technology used, starting with PC's and workstations, minicomputers and mainframes up to supercomputers.

4. SPARC and UNIX

SPARC and UNIX are highly complementary. SPARC can power a wide range of fast, flexible, low–cost computers from the high to low end. Meanwhile, UNIX lets these machines offer important capabilities, such as networking and windowing, linking all SPARC computers together through their common Application Binary Interface (ABI).

Perhaps the most important element of a SPARC/UNIX system is a broad universe of software applications, all compatible, all interchangeable between different SPARC computers from different vendors. The key is standards. Specifically, UNIX System V, Release 4, which is endorsed by far more leading companies than any other UNIX operating system. The other standard is SPARC, rapidly becoming the CPU solution for tomorrow's fast computers. See Figure 3.

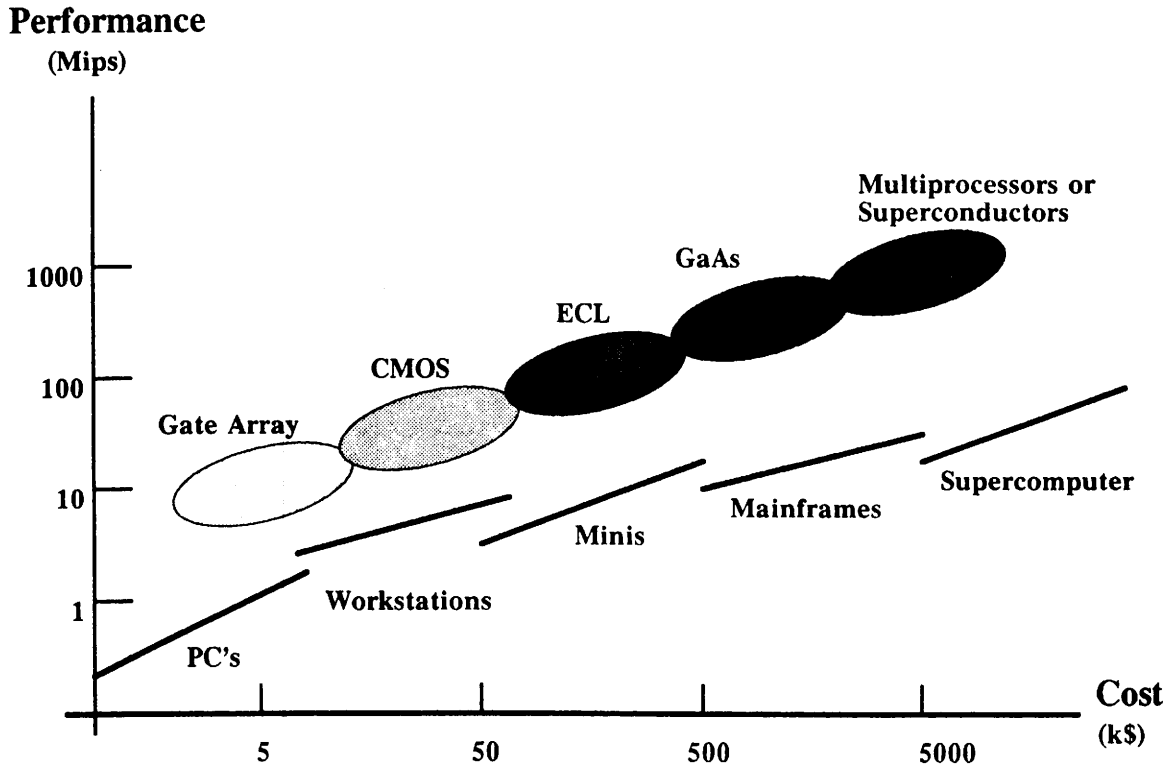


Figure 2: *SPARC-Vision*

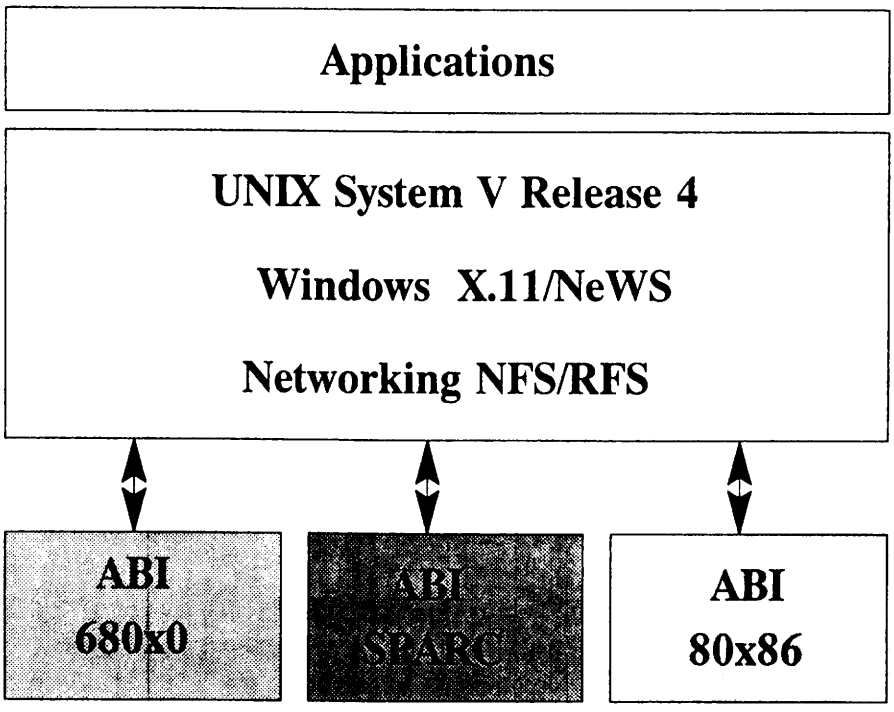


Figure 3: *SPARC and UNIX*

5. SPARC Architecture Features

In general a reduced instruction set computer maximizes speed of the processor by making it perform only simplest and most used functions. Software performs infrequent functions minimizes instruction count and data transaction count.

The SPARC architecture provides the following features:

- Simple instructions – Most instructions require only a single arithmetic operation.
- Few and simple instruction formats – All instructions are 32 bits wide, and are aligned on 32–bit boundaries in memory. There are only three basic instruction formats, and they feature uniform placement of opcode and register address fields.
- Register–intensive architecture – Most instructions operate on either two registers or one register and a constant, and place the result in a third register. Only load and store instructions access storage.
- A large "windowed" register file – The processor has access to a large number of registers configured into several overlapping sets. This scheme allows compilers to cache local values across subroutine calls, and provides a register–based parameter passing mechanism.
- Delayed control transfer – The processor always fetches the next instruction after a control transfer, and either executes it or annuls it, depending on the transfer's "annul" bit. Compilers can rearrange code to place a useful instruction after a delayed control transfer and thereby take better advantage of the processor's pipeline.
- One–cycle execution – To take maximum advantage of the SPARC architecture, the memory system should be able to fetch instructions at an average rate of one per processor cycle. This allows most instructions to execute in one cycle.
- Concurrent floating point – Floating–point operate instructions can execute concurrently with each other and with other non–floating–point instructions.
- Coprocessor interface – The architecture supports a simple coprocessor interface. The coprocessor instruction set is analogous to the floating–point instruction set.
- AI support – The tagged arithmetic instructions can be used by languages such as Lisp, Smalltalk and Prolog.
- multiprocessor support – SPARC has two special instructions to support tightly coupled multiprocessors.

6. SPARC Architecture Overview

A SPARC processor logically comprises an integer unit (IU), a floating–point unit (FPU), and an optional, implementation–defined coprocessor (CP), each with its own set of registers. This organization allows maximum concurrency between integer, floating–point, and coprocessor instructions. All of the registers, with the possible exception of the coprocessor's, are 32 bits wide. Instructions operate on single registers, register pairs, or register quads. See Figure 4.

A typical system that uses the SPARC architecture is organized around a 32–bit virtual address bus and a 32–bit instruction/data bus. Its storage subsystem consists of a memory management unit (MMU) and a large cache for both instructions and data. The cache is virtual–address–based. Depending on the storage subsystem's interpretation of the processor's address space identifier bits, I/O registers are either addressed directly, bypassing the MMU, or they are mapped by the MMU into virtual addresses.

The SPARC Architecture does not specify an I/O interface, a cache/memory architecture, or an MMU. Although the instruction set has no intrinsic bias favoring either virtual– or real–addressed caches, most systems are currently based on virtual–addressed caches in order to minimize the cycle time.

SPARC does not specify an MMU for the following reasons: An MMU definition is best established by the requirements of particular hardware and operating systems, is not visible to application–level programs, and is not a performance bottleneck with virtual–addressed caches because it is not in series between the processor and the cache. Sun has defined a "reference" MMU, which is included in the latest Sun Sparcstation products.

The SPARC reference memory management unit supports multiple contexts, 4 GByte virtual address space and upto 64 GByte physical address space.

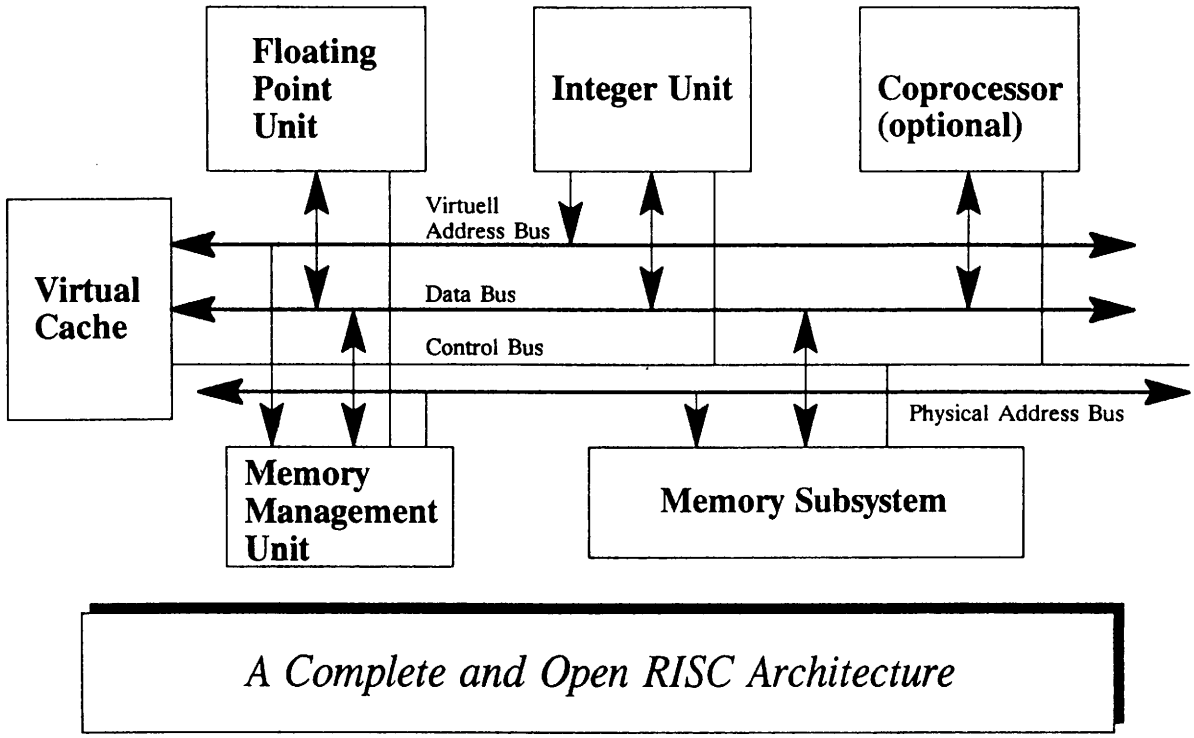


Figure 4: *SPARC Architecture Overview*

7. SPARC Integer Unit

The IU is the basic processing engine of the SPARC architecture. It executes all the instruction set except floating–point operate instructions and coprocessor instructions. See Figure 5.

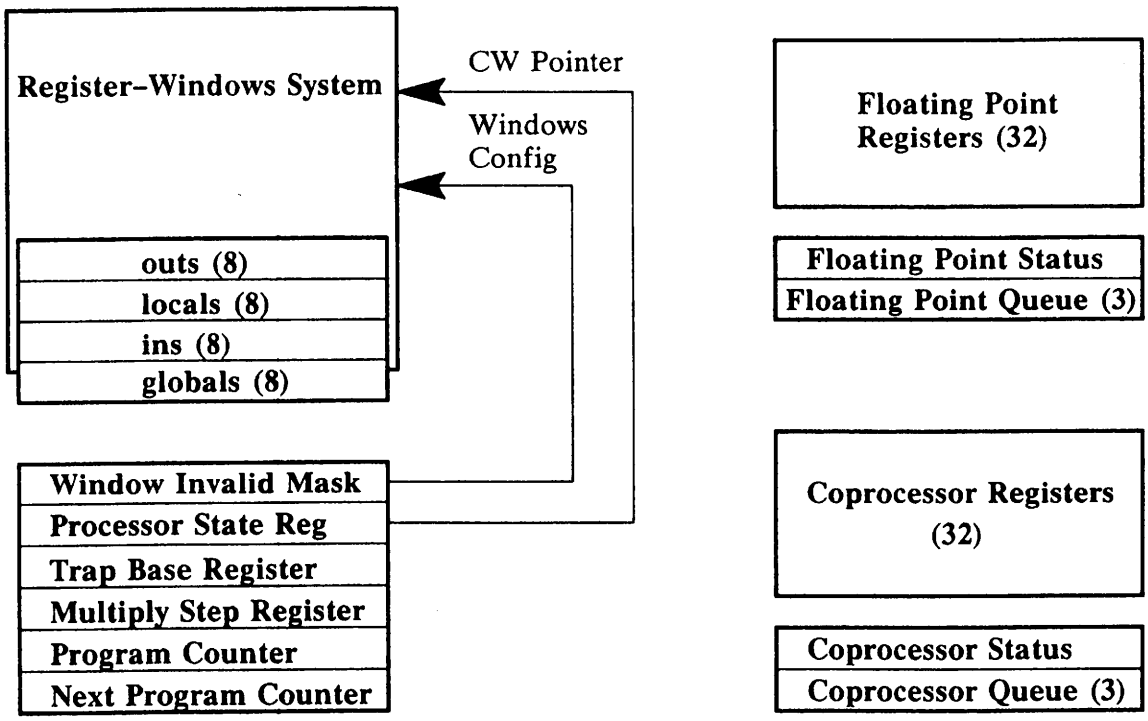


Figure 5: *SPARC Register Structure*

The register structure forms an important part of the overall architecture. The integer unit has two types of registers associated with it; working registers and control/status registers. Working registers are used for normal operations, and control/status registers keep track of and control the state of the IU. The IU's working registers are divided into several windows, each with twenty–four 32–bit working registers, and each having access to the same eight 32–bit global registers. The current window pointer (CWP) field in the processor state register (PSR) keeps track of which window is currently "active".

In addition to the window registers and global registers, the SPARC architecture provides several control and status registers, and a non–windowed working register file for the FPU and the optional coprocessor.

The IU's control/status registers are all 32–bit read/write registers. They include the program counters (PC and nPC), the Processor State Register (PSR), the Window Invalid Mask register (WIM), the Trap Base Register (TBR), and the multiply–step (Y) register.

The Processor State Register holds a user/supervisor bit, MC68000–compatible integer condition codes (negative, zero, overflow, and carry), the 4–bit processor interrupt level (PIL), FPU and CP disable bits, the CWP, and an 8–bit version/implementation number. The Trap Base Register (TBR) holds a programmable base address for the trap table and an 8–bit field that identifies the type of the current trap. Like the WIM, the PSR and TBR are only accessible by the operating system.

The Program Counter (PC) contains the address of the instruction currently being executed by the IU, and the nPC holds the address of the next instruction to be executed (assuming a trap does not occur).

In delayed control transfers, the instruction that immediately follows a control transfer may be executed before control is transferred to the target. The nPC is necessary to implement this feature.

8. SPARC Register Windows

The IU may contain from 40 to 520 general–purpose 32–bit registers. This range corresponds to a grouping of the registers into 2 to 32 overlapping register windows, where the actual number of registers and windows depends on the implementation. The number of windows in a particular chip is not visible by a compiler or application program.

At any one time, a program can address 32 integer registers: the 8 ins, 8 locals, and 8 outs of the active window and the 8 globals that are addressable from any window. The 8 outs of one window are also the 8 ins of the adjacent window. Note that global register 0 is always zero, making the most frequently used constant easily available at all times.

The active window is identified by the 5–bit Current Window Pointer (CWP) in the Processor Status Register (PSR) and is invisible to the user. The save instruction decrements the CWP, making the next window become active and, due to the overlapping, making the outs of the old window addressable as the ins of the new window. Incrementing the CWP with the restore instruction makes the previous window active. See Figure 6.

Programs nearly always use more windows than a particular chip provides. An overflow trap to the operating system occurs if all the windows are full before a save; an underflow trap occurs if they are empty before a restore. In SunOS, windows are saved in the memory stack.

Although the overlap of adjacent ins and outs provides an efficient way to pass parameters, the principal benefit of windows is their cachelike behavior. As a program calls and returns procedures, control moves up and down the execution stack but generally fluctuates around particular levels in the execution stack. The register windows are effective when the average size of these fluctuations is less than the number of windows.

Register windows have several advantages over a fixed set of registers. By acting like a cache, they reduce the number of load and store instructions issued by a compiler because register–allocated locals and return addresses need not be explicitly saved to and restored from the memory stack across procedure calls. A consequence is a decrease in the chip/cache and cache/memory operand bandwidth – that is, fewer loads and stores and fewer data cache misses. This benefits chips with multicycle load or store instructions by executing fewer of them and serves tightly coupled, cache–consistent multiprocessors by reducing the memory bus traffic.

Windows perform well in incremental compilation environments such as Lisp and in object–oriented languages such as Smalltalk. On the average, register windows are better than a fixed register set architecture because windows respond dynamically to the runtime behavior of programs.

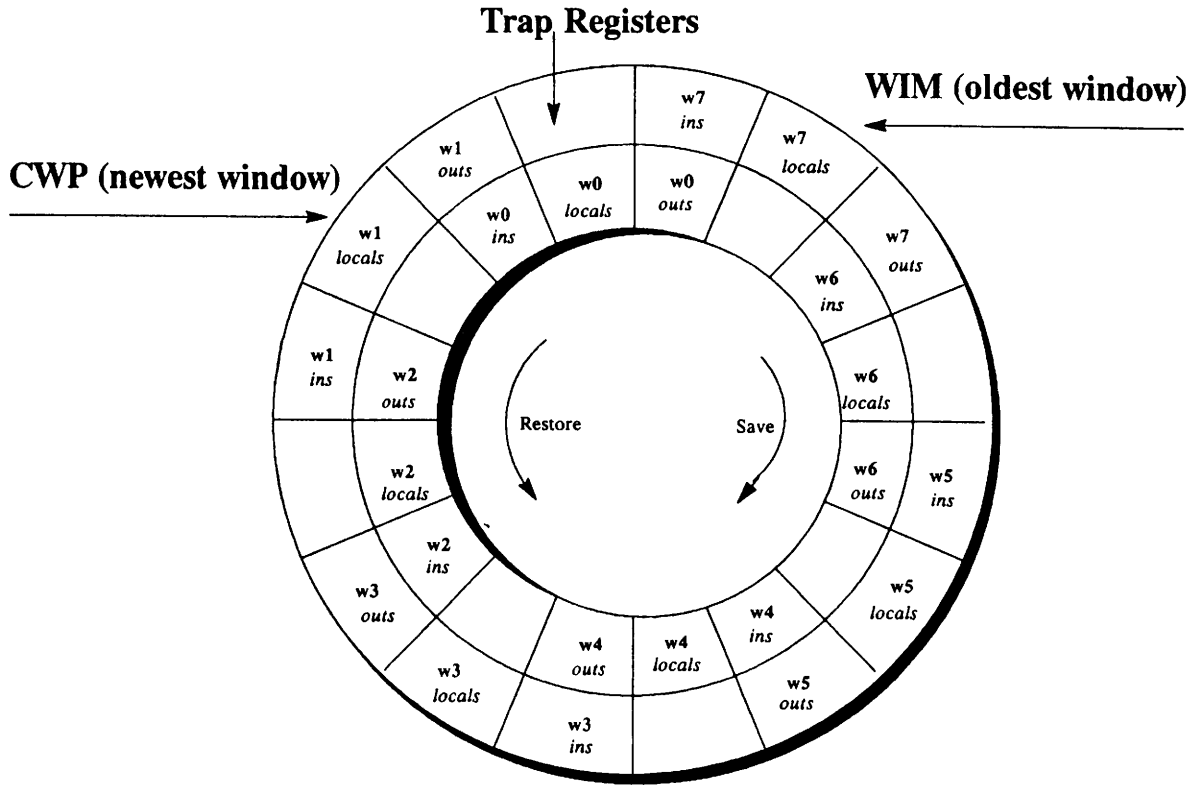


Figure 6: *SPARC Register Windows*

This representation of the register windows assumes eight windows. Assume W7 is the oldest window, and WO is the newest and corresponds to a procedure that attempted to execute a save instruction and generated a window–overflow trap. The trap handler cannot use W7's ins or W1's outs, but it is always guaranteed WO's locals.

The number of windows ranges from 2 to 32 depending on the implementation. Implemented windows must be contiguously numbered from 0 to NWINDOWS – 1.

The windows are addressed by the CWP, a field of the Processor State Register (PSR). The CWP is incremented by a RESTORE or RETT instruction and decremented by a SAVE instruction. The active window is defined as the window currently pointed to by the CWP.

The Window Invalid Mask (WIM) is a register which, under software control, detects the occurrence of IU register file overflows and underflows.

The registers in each window are divided into ins, outs, and locals. Note that the globals, while not really part of any particular window, can be addressed when any window is active. When any particular window is active, the registers are addressed as follows:

ins are r[24] to r[31]

locals are r[16] to r[23]

outs are r [8] to r[15]

globals are r [0] to r [7]

Each window shares its ins and outs with adjacent windows. The outs from a previous window (CWP + 1) are the ins of the current window, and the outs from the current window are the ins for the next window (CWP – 1). The globals are equally available from all windows, and the locals are unique to each window.

9. SPARC Instruction Set

SPARC defines 55 basic integer instructions, 14 floating–point instructions, and 2 coprocessor computational formats. All instructions are 32 bits wide.

Functionally, SPARC architecture instructions fall into 8 categories:

1. load and store
2. arithmetic/logical/shift
3. control transfer
4. read/write control register
5. floating–point operate
6. coprocessor operate
7. tagged arithmetic
8. atomic instructions for multiprocessor support.

Instructions may also be classified into three major formats, two of which include subformats.

The three instruction formats are called format 1, format 2, and format 3. Figure 7 shows each instruction format, with its fields and bit positions. It also lists the types of instructions that use that format.

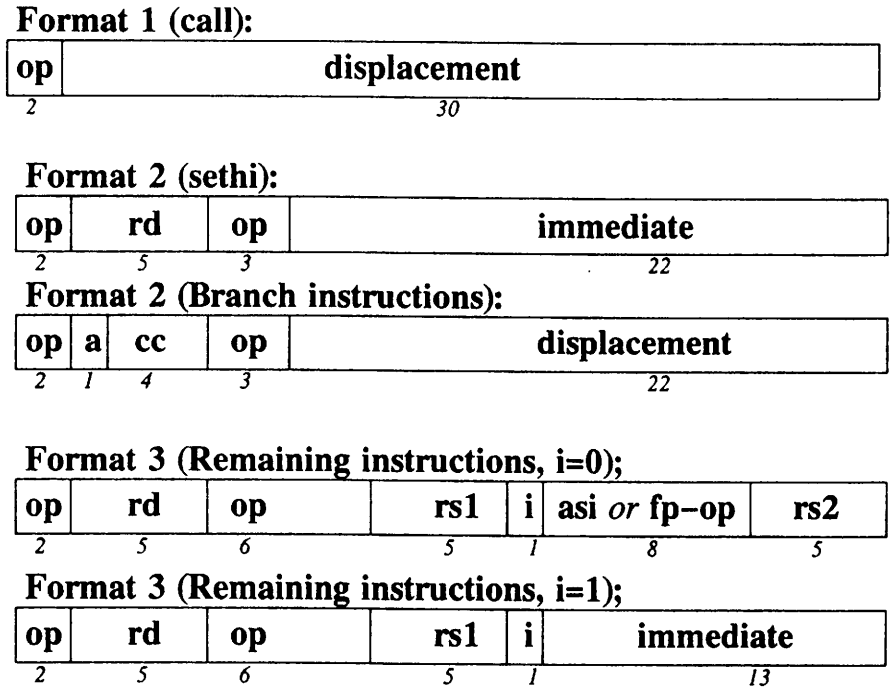


Figure 7: *SPARC Instruction Formats*

- Format 1 has a 30–bit word displacement for the call instruction. Thus, a call or branch can be made to an arbitrarily distant location in a single instruction.
- Format 2 defines two instruction types: sethi and branch. The 22–bit word displacement defines a +– 8–Mbyte distance for the PC–relative conditional branch instruction.
- Format 3, which has specifiers for two source registers and a destination register, encodes the remaining instructions. As in the Berkeley RISC architectures, when i = 1, a sign–extended 13–bit immediate value substitutes for the second register specifier. For the load/store instructions, the upper 8 bits of this field are used as the address space identifier (ASI) and, along with the i bit, as an opcode extension field for the floating–point and coprocessor instructions.

- Unused opcode space is reserved for future expansion. Unimplemented instructions trap, and reserved fields must be 0.

10. SPARC Floating Point Details

The FPU has 32 32–bit registers. Double–precision values occupy an even–odd pair of registers, and extended–precision values occupy an aligned group of four registers. The floating–point registers can hold maximum of either 32 single–, 16 double–, or 8 extended–precision values. The FPU's registers are accessed externally only via memory load and store instructions; there is no direct path between the IU and the FPU. Floating–point load and store double instructions improve the performance of double–precision programs.

Although the user–level floating–point architecture conforms to ANSI/IEEE 754/1985, some nuances of the standard, such as gradual underflow, may be handled by software. An implementation indicates that it cannot produce a correct ANSI/IEEE 754–1985 result by generating a floating–point unfinished or unimplemented trap. System software then simulates the missing functions. The operating system must also emulate the entire FPU if it is not present in a particular system.

SPARC allows floating–point arithmetic operations, such as multiply and add, to execute concurrently with each other and with integer instructions and floating– point loads and stores. For example, it is possible to preload a floating–point value while executing a floating–point multiply and a floating–point add. The degree of concurrency is implementation–dependent – for example, an FPU can have several multipliers and adders. In all current system implementations, floating–point arithmetic instructions execute in parallel with cache misses.

The FPU performs all the required register interlocks, such as not beginning another floating–point instruction until all its operands have been computed, so concurrency is hidden from the user. Programs generate the same results including the order of floating–point traps as if all instructions ran sequentially. To handle traps properly, the FPU maintains a floating–point queue (FQ). The first–in, first–out queue records all pending floating–point arithmetic instructions and their memory addresses at the time of a floating–point trap. The depth of the queue depends on the FPU microarchitecture. In the SunOS, software emulates all the instructions found in the queue at the time of a floating–point trap.

The user–accessible Floating–point State Register (FSR) contains mode and status information; in particular, there are trap–enable control bits, current–exception bits, and accrued–exception status bits for the five ANSI/IEEE 754–1985 trap types.

11. Load and Store Instructions

The load and store instructions move bytes (8 bits), halfwords (16 bits), words (32 bits), and doublewords (64 bits) between the memory and either the IU, FPU, or CP. These are the only instructions that access main memory, which, to user application programs, is a byte–addressable, 32–bit (4–gigabyte) memory space. Because the CP is implementation–dependent, it can load/store data of other sizes, such as quadwords.

For the floating–point and coprocessor loads and stores, the IU generates the memory address, and the FPU or CP sources or sinks the data. I/O device registers are accessed via load/store instructions.

As with most RISCs, the load and store halfword, word, and doubleword instructions trap if the addressed data are not aligned on corresponding boundaries. For example, a load or store word instruction traps if the low–order two address bits are not 0. If necessary, the operating system can emulate unaligned accesses.

The load and store instructions assume Motorola 68000– and IBM 370–compatible byte ordering: Byte 0 is the most significant byte, and byte 3 is the least significant byte in a datum. To preclude possible incompatibilities between SPARC application binaries that can access common data, only one byte ordering has been defined.

SPARC defines interlocked delayed loads: The instruction that immediately follows a load may use the loaded data, but if it does, the load may slow down, depending on the implementation.

Special load and store alternate instructions, usable only by the operating system, allow access to a number of 32–bit address spaces defined by a particular hardware system. An 8–bit address space identifier (ASI) is supplied by the load/store alternate instructions to the memory, along with the 32–bit data address. The architecture specifies four alternate spaces – user instruction, user data, supervisor instruction, and supervisor data – and leaves the remainder to be defined by the external hardware system. These ASIs can be used to access system resources that are invisible to the user, such as the MMU itself.

12. Tagged Instructions

The tagged arithmetic instructions can be used by languages such as Lisp, Smalltalk, and Prolog that benefit from tags. Tagged add (taddcc) and subtract (tsubcc) set the overflow bit if either of the operands has a nonzero tag or if a normal arithmetic overflow occurs.

The tag bits are the least significant two bits of an operand, so that integers are assumed to be 30 bits wide and left–justified in a word with a zero tag. The tsubcc instruction with global 0 as its destination is the tagged compare instruction.

Normally, a tagged add/subtract instruction is followed by a conditional branch, which, if the overflow bit has been set, transfers control to code that further deciphers the operand types. Two variants, taddcctv and tsubcctv, trap when the overflow bit has been set. These trapping versions can be used for error checking when the compiler knows the operand types.

13. Multiprocessor Instructions

Two special instructions support tightly coupled multiprocessors: swap and "atomic load and store unsigned byte" (ldstub).

The swap instructions exchange the r register identified by the rd field with the contents of the addressed memory location. This is performed atomically without allowing asynchronous traps. In a multiprocessor system, two or more processors issuing swap instructions simultaneously are guaranteed to get results corresponding to the executing the instructions serially, in same order.

These instructions cause a mem_address_not_aligned trap if the effective address is not word–aligned.

If a swap instruction traps, memory remains unchanged.

The atomic load–store instructions move a byte from memory into an r register identified by the rd field and then rewrite the same byte in memory to all ones without allowing intervening asynchronous traps. In a multiprocessor system, two or more processors executing atomic load–store instructions addressing the same byte simultaneously are guaranteed to execute them in same serial order.

14. Architectural Comparison

The comparison between SPARC–, 88000–, mips– and i860–RISC architecture is summarized in figure 8.

	SPARC	88000	mips	i860
Instruction Set				
64–bit load/store	yes	no	no	no
Multiprocessor supp.	atomic l/s instruction	xmem	no	lock/unlock next instruc.
Register Set	register windows	fixed register set	fixed register set	fixed register set
	register windows ease burden on compilers	sophisticated compilers required	sophisticated compilers required	complex compilers required to handle complexity of chip
	does not preclude interprocedural reg. allocation techniques	N/A	N/A	N/A

Figure 8a: *Architectural Comparison*

	SPARC	88000	mips	i860
Implementation				
integration	flexible	fixed	fixed	fixed
cache, MMU, FP, MP	flexible	fixed	fixed	fixed
bus structure	implement. issue (split/ comb. i/d bus)	split i/d bus	multiplexed i/d bus	split i/d bus
external coprocessor	yes	no	yes	no
Business				
Volume shipping	3 vendors	?	1 vendor	(alpha vers.)
MHz	16,20,25,33	?	20,25	15–20
Application SW	500+	?	50 (?)	?

Figure 8b: *Architectural Comparison (contd.)*

15. SPARC Vision today

If you look at the status of SPARC vision today, you will realize, that part of it has become already reality.

SPARC microprocessor chips are available from 4 different vendors using gate array, CMOS or ECL semiconductor process technology, as shown in Figure 9.

Following figure shows several semiconductor vendors, who licensed SPARC and related technology. First licensee was Fujitsu. Their Gate–Array implementation of the SPARC architecture was available in April 1986. A stable SunOS was running on a prototype system in June 1986, only two month after the first chips arrived. The first customer shipment of a Sun 4/200 was about 1 year later in August 1987.

LSI–Logic implemented the SPARC architecture in their HCMOS process. The performance is between 12.5 and 16 mips. The processors are used in the latest Sun 4 products, the Sun 4/60 and the Sun 4/300 series.

SPARC processors with 20 mips and 65 mips are available today from Cypress Semiconductor and Bipolar Integrated Technologies respectively.

Two other companies are working on single processor implementations reaching 100 or 200 mips.

Chips have been announced, but are not available today in volume.

The World of SPARC is not only the SPARC architecture itself but also a powerful operating system SunOS and optimizing compilers.

Among the SPARC system software products being offered by Sun, Phoenix, and Interactive Systems are optimizing compilers that allow the development of efficient application software in languages such as C and FORTRAN, with C++, Pascal, Lisp, and Ada soon to follow. There is also a growing list of software companies such as Wind River Systems, JMI, and Ready Systems who offer products that support SPARC in realtime markets.

For hardware and software design, Sun offers several useful tools. The SPARC Architectural Simulator (SPARCsim TM) eases the development task by allowing designers to emulate an entire SPARC system architecture before building a prototype. Information can be fed into other analysis tools such as the SPARC Trace Analyzer. The SPARC Remote Debugger has full source–code symbolic debugging capabilities for debugging processes on remote SPARC systems.

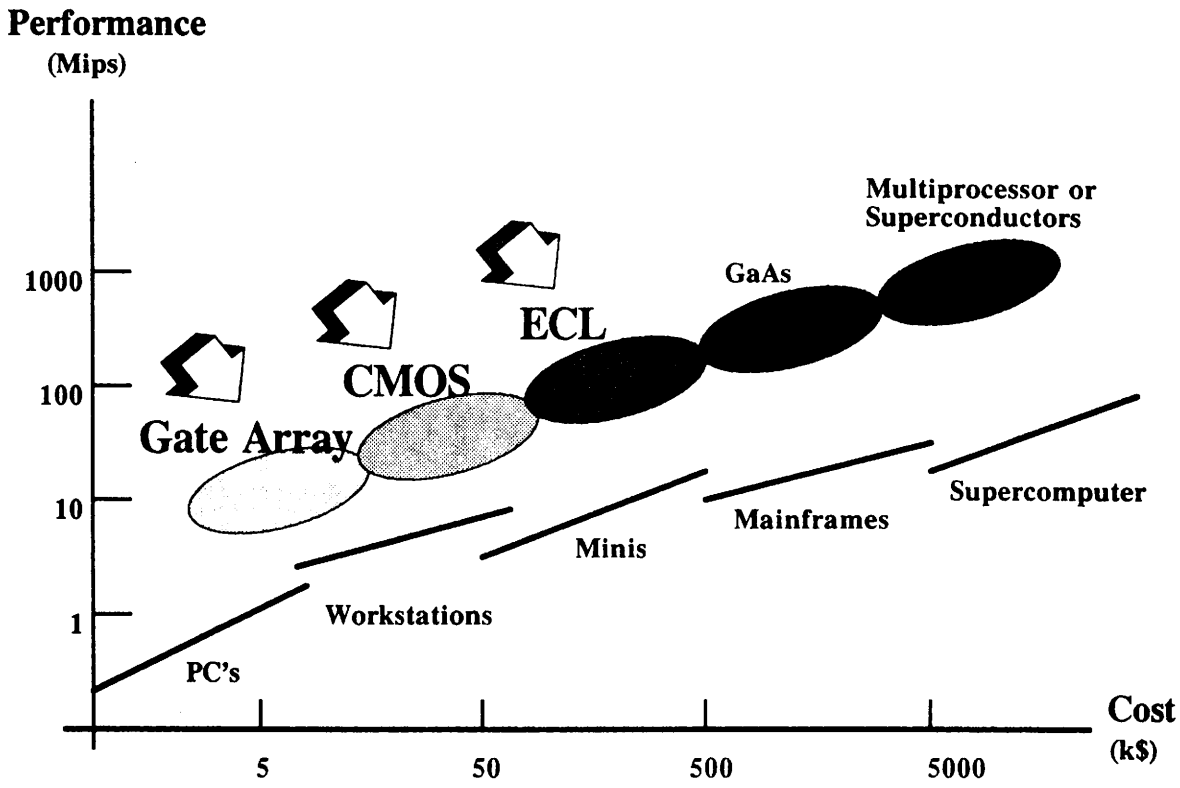


Figure 9: *SPARC-Vision and Reality Today*

- SPARC Architecture
- SunOS, Optimizing Compilers
- Real–Time Operating Systems
- SPARC Based Systems
- SPARC Development Tools
- 500+ Applications Programs
- Five Semiconductor Vendors
- Multiple Architecture Licensees
- Board–level Products
- SPARC International

A Complete Solution available today to support your product development needs for fast time to market

Figure 10: *The World of SPARC*

Application software is the fuel that enables a computer system to achieve broad success in the marketplace. Without it, any machine quickly descends into obscurity. For example, one of the chief factors helping the IBM PC standard to proliferate was an enormous base of compatible software that ran on any DOS PC.

SPARCware is the identical solution for the 1990s. It is the shrink–wrapped software approach for high–performance computer users. SPARCware is already the largest base of RISC software in the industry. There are now more than 500 SPARCware applications. More are being added daily as the packages in Sun's 2300–plus Catalyst program are ported to SPARC.

SPARC is the only RISC architecture that is openly licensed. Enhancing this openness is a rich base of development tools and system software that is beyond anything found in the UNIX industry.

Sun licenses its operating system, SunOS (a merge of UNIX System V and 4.3/4.2 BSD), as well as its other essential system software. Products designed around SunOS will automatically run under AT&T's UNIX System V Release 4 when it becomes available. Another approach is for developers to acquire these software elements through Phoenix Technologies Limited and Interactive Systems Corporation, the leaders in the UNIX systems porting business.

TeamSo: Team Software Development Support System

Eva Strausz & Janos Szel

Department of Electronics Computer and Automation Institute
Hungarian Academy of Sciences

ABSTRACT

A software tool is introduced to help software development in team-work. This paper provides an overview of this package called TeamSo, and describes some of the most important design and implementation decisions.

1. Introduction

Most software is too large to be built by one person, on the other hand many problems arise when it is developed by a team. The team work can cause inevitable confusion, like double maintenance, simultaneous update etc. To avoid these, standards and co-ordination is needed between staff members to avoid a fall-off in productivity.

TeamSo is a software tool designed to solve team productivity problems. It manages the software development process by taking over the communication between team members.

The operation of the TeamSo system is based on a so-called PROJECT_DATABASE which contains the design and implementational information for the given project. It is a shared project-database containing all the information necessary to produce the target software and the appropriate of documentation.

2. Short description

TeamSo leads the user in a compulsory direction; in this way it helps to find the best solution for the target software. This guide does not mean a necessity but rather a helpful control, where the choice among the possibilities are at the user's disposal.

During the work different images appear on the screen always having the structure shown in Figure 1.

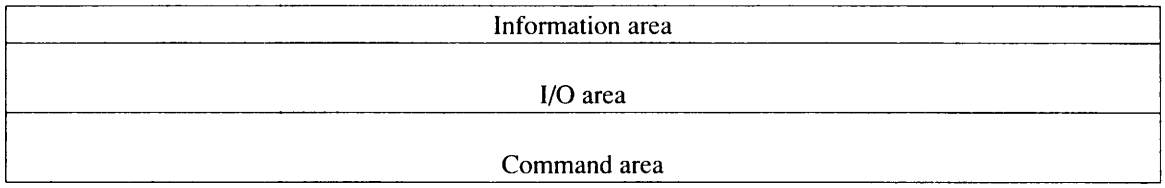


Figure 1

The information area lets the user know the type of manipulation and the level of competence.

Actually in the I/O area a questionnaire appears. After answering the questions TeamSo controls them and – if correct – maintains/updates the database. If there are any problems, the user is forced to repeat. The answers can be typed in or chosen from a menu.

The command area consists of menu items. The user can choose any element with the cursor-keys or by entering the first character of the command. To confirm the desired action the user presses the <ENTER> key. Changing between the I/O and command areas is possible whilst preserving the previous input state. A help facility is always amongst the available menu items. Having chosen it, all necessary information is available concerning the state appearing in the information area.

The rest of the paper gives a detailed description of TeamSo.

3. How to get started

Let us show the main milestones of the team productivity coordination.

From the UNIX Shell one can enter in the TeamsSo with the "tea" command. The greeting picture is displayed and the cursor is on the work menu item. See Figure 2.

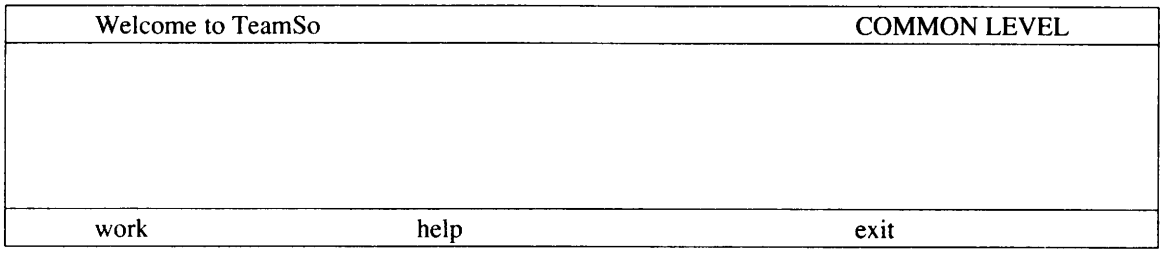


Figure 2

Approving the work menu item the image shown in Figure 3 appears.

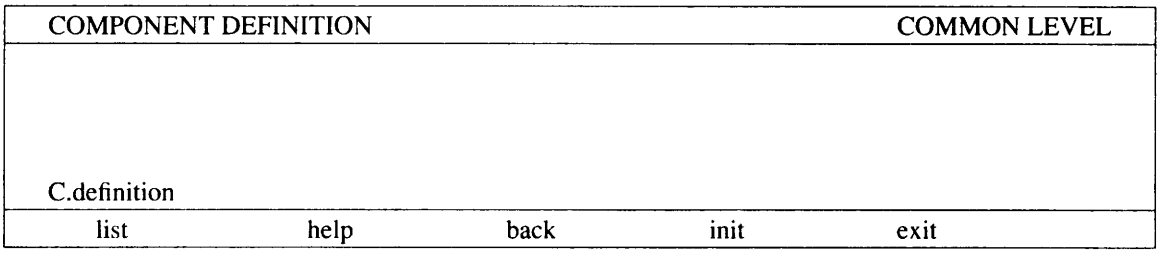


Figure 3

Now it is time to give the component definition. What does that involve? A component is a defined part of the task to be solved. TeamSo defines three types of components each of which represent a competence level in the team organization. The work is shared as follows:

- Project Manager level
- System Manager level
- Module Manager level

The Project Manager is responsible for the final acceptance of the product.

The System Manager maintains and controls a logically self-contained part of the project.

The Module Manager is the programmer; he writes the program (code) on the basis of the skeleton automatically generated from the System Manager's database realm.

Returning to the component definition, syntactically it is built of component names according to the following rules:

- Project definition: P-name
- System definition: P-name/S-name
- Module definition: P-name/S-name/M-name

Entering a component definition, the system determines its type and informs the user about the actual level and examines whether the component already exists. If so, it checks the validity by asking for the secret identifier. After typing the correct identifier the system opens. In case of any mistyping or illegal attempt the system rejects the user.

A new project-entry can be made at this level. The future Project Manager has to define his own identifier.

After having successfully logged into the system, the questionnaire appearing on the screen is the same concerning the three components. Therefore only a project definition process will be shown. See Figure 4.

Project 'P-name'- header questionnaire					PROJECT LEVEL
name full name version date author testing mode					
save	mail	help	back	init	exit

Figure 4

If it is a new project, the questionnaire is blank and the Project Manager is forced to fill each item. If not, the previous state (retrieving from the database) is viewable and can be modified. Finally the filled questionnaire is saved. If it is an existing project, the database is updated, otherwise a new database is created.

4. Project decomposition

The next step is the decomposition of the project into logical parts. The Project Manager determines the number of systems, assigns the System Managers, and defines their tasks.

Having defined the second level in the component-hierarchy TeamSo forces the Project Manager to determine the connections among the systems on a new screen image where the system-names are already defined and the possible connections are displayed. The Project Manager co-ordinates the systems by choosing from the following connection types:

- active data communication between processes
- passive function calling
- inclusive one system belongs to another in case one of them is a data file.

A manager at any time can modify all his statements. The TeamSo informs the team members concerned and ensures the integrity of the system. In addition a mail feature can be used for "verbal" messages.

5. System decomposition

However the System Manager's task is not yet finished. Besides the above mentioned general administrative actions he has to define the programming environment for the Module Managers. He can do it by filling the questionnaire shown in Figure 5.

'P-name/S-name/M-name' environment					SYSTEM LEVEL
Name Purpose Language Algorithm ... Debug/test Abnormal termination					
save	data	subr	mail	.....	exit

Figure 5

Saving the fulfilled questionnaire the program skeleton is automatically generated. It forms the comment part of the source code written in the given language.

Since each module is a single file of source code containing a number of related subroutines and data definitions, now the System Manager should describe them. The filling mechanism is similar in both cases. We show an example data definition in Figure 6.

'P-name/S-name/M-name' data structures							SYSTEM LEVEL
Name Type Number Storage ...							
Code(s)							
save	prev	next	crea	dele	type	....	exit

Figure 6

The data definition follows the rules of the previously defined programming environment. This mechanism is provided by TeamSo to avoid any semantical problems. It also takes into consideration their common usage by the co-operating modules.

The generated code – deriving from the given definition – is immediately displayed on the bottom of the I/O area. The definition can modified at any time; naturally the output follows the change.

Besides the default types of the given language the System Manager can construct his own ones too.

The result of the System Manager's action is not only a set of skeletons, but a "makefile" is generated also containing the dependencies and the compilation instructions.

The detailed design for a module is finished; it is now time to introduce the Module Manager level where the effective code is to be written.

6. Writing code

When a Module Manager (i.e. the programmer) successfully enters into TeamSo, the list of source file names assigned to him appear on the screen, See Figure 7.

'P-name/S-name/M-name' Source directory			MODULE LEVEL
prog.1 prog.2 ... prog.n			
mail	help	....	exit

Figure 7

This file contains comments concerning the task to be solved, data definitions and empty subroutines derived from the System Manager's decomposition action.

Having chosen one of this files, a mechanism becomes active to prevent simultaneous access of different programmers. This is the so-called "check-in" and "check-out" facility based on the UNIX SCCS feature. Technically a copy is made in his private workspace ("check out") where he can use the shell with restricted rights (rsh) and do his task (editing, compiling, debugging, etc). Having finished his work the result is returned to the source directory (heck in"). The System Manager can run his "makefile" at any time to check the status of the system, and to produce the output for the Project Manager.

7. Documentation

One of our goals in the design of TeamSo was to support the programming staff in producing software documentation in an easy way. Certain parts of the filled questionnaire are built in the future documentation, so the TeamSo user is freed from the tiring and boring but inevitable work.

The documentation of each level can be reached and enlarged optionally using an editor. The user can place metacharacters in his document, to mark arbitrary text parts for different kinds of documentation (e.g. user, designer, programmer, etc).

When certain documentation is needed, TeamSo collects the respective information stored on different levels and creates a comprehensive text. Naturally this text can be stored in a file, or on different media and can be printed out at any time.

8. Hardware/software environment

- CPU: IBM PC/AT

  1 Mb memory, FPP, 2 serial lines, 2 ST225 Winchester, 1.2 Mb floppy driver
- Op. sys: SCO XENIX System V Ver. 2.2.1
- Language: C
- Code size: 200 kbytes

9. Conclusion

The basic concept of TeamSo was developed two years ago in our Institute. F. Sipos and T. Sztano implemented their system on PDP/RSX, on PC-DOS and on VAX/VMS.

Our solution has partly adopted the main features of the previous realizations, however we took advantage of the possibilities provided by UNIX so we could produce a much more compact and faster software tool.

Concerning our future plans we are to implement TeamSo on a multiprocessor VME system. Since we made efforts to produce portable code, we hope the implementation will be easy and successful.

A SQL Programming Interface for the Relational Database System Db++

Ralph Zainlinger
Thomas Hadek

Technical University Vienna
Institut für Techn. Informatik
Treitlstr. 3/182/1
A-1040 Vienna
Austria
ralph@vmars.at
tom@vmars.at

ABSTRACT

Database systems in the UNIX world are still a controversial topic splitting the research community. Today's users can choose between systems such as ORACLE, well known, broadly accepted, and expensive, or typical UNIX databases, such as the relational database system db++, integrated into the UNIX framework at a reasonable price but with rather low capabilities compared with the first group. The major shortcomings of db++ are the low-level programming interface, the lack of mechanisms for concurrent multiple access (multi-user) and the neglect of transaction based processing. This paper describes various extensions of the relational database system db++ based upon a high-level SQL programming interface. The main purpose is to maintain the obvious advantages of db++ by simultaneously increasing its functionality and general applicability with standardised mechanisms and concepts.

1. Introduction

With Oracle, the UNIX world relies upon a very powerful and sophisticated relational database management system (RDBMS), so why should we spend efforts in extending a rather low-level database system named db++ [Agn86a]? The reasons are as follows: First, Oracle is a rather costly solution and thus very often not attractive, especially for universities and small software companies. Second, db++ fits extremely well into the UNIX framework, thus allowing the application of a number of standard UNIX tools for processing, e.g. cp, mv, diff. Third, db++ is centered around a very powerful query language.

Our work was originally motivated by the lack of a high-level C-programming interface for db++ which made it very difficult to build applications with reasonable efforts. Discussing the various alternatives, it turned out that building a SQL programming interface is the most useful and thus appropriate solution. Although SQL was used exclusively as an interactive query language in the past, recent trends show that SQL tends to become "the" standard for programming interfaces of RDBMSs.

The organisation of the paper is as follows: Section 2 evaluates the relational database system db++ with respect to a number of basic requirements that should be met by a RDBMS. Section 3 compares the traditional approaches for programming interfaces of database systems. In section 4 the major concepts of

our extensions are described. Implementational aspects are detailed in section 5.

2. Db++ – A Critical Review

2.1. Db++

Db++ is a relational database system especially developed for UNIX systems. In contrast to other UNIX database systems db++ does not impose a new working environment on the user. Db++ behaves like a standard UNIX tool and can thus be combined with a variety of other UNIX utilities.

Data storage is not transparent to the user. All information pertaining to a single relation is contained in a single UNIX file. Therefore, standard UNIX commands like *cp(1)*, *chmod(1)*, and *rm(1)* can be applied to db++ relations.

Db++ consists of a family of programs supporting the typical database activities (creation, modification, query and report generation). These programs are built around the "UNIX philosophy" thus allowing perfect cooperation with the existing UNIX environment.

Database queries can be formulated in two ways. First, db++ offers a command language interface based on an algebraic query language. This database interface is a high-level interface in the sense that it is not necessary to write complex programs in order to retrieve information. The second possibility to extract information out of the database is to write C programs which make use of the special db++ C library functions (*raccess, simple* [Agn88]). These library functions provide a rather low-level access mechanism; basic as well as sophisticated operations such as *union* and *join* are not available.

2.2. Db++ – A Database Management System?

The term "database management system" (DBMS) refers to a database system fulfilling a number of basic requirements as detailed in [Ans86a, Cod85a], and [XPG87a]. Since db++ neglects several of these requirements, we avoided speaking of a DBMS when we mentioned db++. We will now examine db++ with respect to the most important requirements.

Data Types

Db++ supports all important data types. The fact that db++ has been written in C leads to a broad correspondence between db++ data types and C data types [Con86a].

Indicators, NULL Values

Indicators are used to represent at the logical level the fact that the information is missing (i.e. not yet specified). Besides the logical representation, the DBMS must support manipulative functions for these indicators and these must also be independent of the data type of the missing information.

Db++ does not support indicators nor does any available programming interface support indicator variables.

Database User

The concept of a database user in db++ is realised through the traditional UNIX file permissions. The ownership facilities of database objects (i.e. relations) are thus equivalent to the well known UNIX mechanisms. It is for instance impossible, that a database object belongs to more than one group.

Views

Views are virtual (i.e. not physically existent) relations resulting from the evaluation of a query specification. The basic idea of views is that they can be treated as a logical unit behaving exactly like a conventional relation.

Db++ does not support views.

Transaction Processing

A transaction is a sequence of operations, including database operations, that is atomic with respect to recovery and concurrency [XPG87].

Although [Agn86] claim that db++ is capable of dealing with transactions, we found, that their notion of a transaction slightly differs from the definition given above. A db++ transaction is restricted to a single relation. The concept used is that of "flushing", i.e. all written information becomes visible to succeeding read operations after an explicit "flush" call (in fact flushing defines the point of commitment). Since only one relation can be flushed at a time this mechanism is inappropriate for transactions spanning more that one relation, because it cannot be guaranteed that a sequence of "flushes" is atomic. Rollback of such restricted transactions is done implicitly if the program crashes.

Explicit rollback can be achieved by a preserving mechanism which is applied to a relation before modifications are initiated. If the modifications are to be rolled back the previously frozen state of the relation can be restored. Again this mechanism is restricted to a single relation.

Multi-User Operation, Consistency, and Concurrency

The original version of db++ did not support multi-user operation. In the meantime an extension (*simple* provides additional services supporting multi-user applications. This enhancement is client/server based, i.e. the application (client) communicates with a server process which directly accesses the database via raccess library functions.

The system allows single tuple locking even in different relations, but most of the problems in connection with concurrent transaction processing are left to the application task:

(1) *Deadlocks* occurring due to tuple locking have to be resolved by the client process, there is, however, no hint how this could be effectively done, since the required knowledge about the competitive processes cannot be obtained.

(2) *Transaction anomalies* occurring due to interleaved transactions are not considered. The application has to handle these effects autonomously, i.e. serialisability issues as well as appropriate usage of shared and exclusive locks have to be carried out in the application domain.

3. Programming Interfaces for RDBMSs – A Comparison

This section compares the most important approaches of programming interfaces for RDBMSs. Shortcomings and advantages are discussed.

3.1. Self Contained Languages

Some databases are based upon a self contained database language used to program the database, e.g. *Cobol, dbase3 plus*. These languages additionally support the generation of masks, menus, etc., but the range of application areas is restricted due to the languages' inherent limitations. Building for example applications manipulating a high-level window-based user interface and concurrently accessing a relational database is impossible.

Among this class of languages we can distinguish between nonprocedural and procedural languages. Nonprocedural languages are capable of expressing *what* kind of information shall be obtained while procedural languages focus on the specification of a detailed retrieval algorithms, i.e. *how* information can be obtained. Nonprocedural languages are often termed *fourth-generation languages* (4GL), procedural languages are known as third-generation languages.

Practical experiments have shown that 4GL code tends to be smaller and can be obtained faster than 3GL code. However, the improvements are not significant [Mis88a]. Besides, many 4GLs suffer from the inefficiencies of the underlying retrieval algorithms, a serious problem, since the programmer has no influence on the performance [Cas89a].

3.2. Host Language Dependent Approach

In this approach the interface of the database is totally adapted to a certain host language. The programmer cannot distinguish between regular programming constructs (e.g. I/O calls) and calls representing the interface to the database. The currently available db++ C programming interface (*raccess, simple*) is a typical example.

Usually these interfaces are library extensions of a conventional programming language. From the conceptional point of view this approach is best suited because the declaration of database related and "regular" data as well as the treatment of database data and program data can be done uniformly.

However, these interfaces suffer from some major deficiencies. Since database operations have very special characteristics (e.g. the result of a query cannot be stored in a single variable) the mechanisms available through the host language are often insufficient or at least troublesome. For the same reason the readability of the programs is dramatically decreased thus making maintenance difficult and expensive. Another origin of the reduced readability stems from the procedural nature of these interfaces.

Figure 1 contains an illustrative example based on the raccess library: A database is used to handle hierarchically related graphical objects. To keep the example clear and simple we confine ourselves to rectangles represented by the relation *RECTANGLE (rec_id, x_coord, y_coord, width, height)*. The father-child relationship is reflected in the relation *IS_CHILD (father_id, child_id)*. The purpose of the sample procedure is to extract all the child rectangles (more precisely their position and dimension) of a given parent rectangle.

```
#include <stdio.h>
#include "raccess.h"

int get_children(father)
     long father;
{
     long x, y, w, h;

     relf *     rp_child, * rp_rec;
     tupleproto_child, proto_rec;
     tuple * result_child, * result_rec;
     fieldf_child, f_rec;
     attribute *     a_child, * a_father;
     attribute *     a_rec_id, * a_x, * a_y, * a_w, * a_h;

     rinit (0, 0);

     if ((rp_child = ropen ("is_child", 0, 0)) == (relf*)0)
          return db_err();
     if ((rp_rec = ropen ("rectangle", 0, 0)) == (relf*)0)
          return db_err();

     rtupinit (rp_child, &proto_child);
     rtupinit (rp_rec, &proto_rec);

     a_father = attno (rp_child, "father_id");
     a_child = attno (rp_child, "child_id");

     a_rec_id = attno (rp_rec, "rec_id");
     a_x = attno (rp_rec, "x_coord");
     a_y = attno (rp_rec, "y_coord");
     a_w = attno (rp_rec, "width");
     a_h = attno (rp_rec, "height");

     f_child.f_long = father;
     afput (a_father, &proto_child, &f_child, (char*)0);
     result_child = rfindtup (rp_child, 0, &proto_child, 1);
```

Figure 1a: *Sample procedure using raccess*

```
        while (result_child != (tuple*)0)
        {
                long r_father, r_child;

                r_father = AFGET (a_father, result_child) -> f_long;
                if (r_father != father)
                        break;

                r_child = AFGET (a_child, result_child) -> f_long;
                rfindall (rp_rec, 0);
                f_rec.f_long = r_child;
                afput (a_rec_id, &proto_rec, &f_rec, (char*)0);
                result_rec = rfindtup (rp_rec, 0, &proto_rec, 1);

                while (result_rec != (tuple*)0)
                {
                        long r_rec_id;

                        r_rec_id = AFGET (a_rec_id, result_rec) -> f_long;
                        if (r_rec_id != r_child)
                                break;

                        x = AFGET (a_x, result_rec) -> f_long;
                        y = AFGET (a_y, result_rec) -> f_long;
                        w = AFGET (a_w, result_rec) -> f_long;
                        h = AFGET (a_h, result_rec) -> f_long;

                        printf ("x = %ld, y = %ld, w = %ld, h = %ld\n",
                                x, y, w, h);
                        result_rec = rget (rp_rec);
                }

                result_child = rget (rp_child);
        }

        rclose (rp_rec);
        rclose (rp_child);
        return (0);
}
```

Figure 1b: *Sample procedure using raccess (cont'd)*

3.3. Host Language Independent Approach

The programming interface is totally independent of the host language, i.e. whatever language is used, the programming interface has the same appearance. In the literature these interfaces are referred to as *embedded* programming interfaces. The most popular representative of this kind of interface is "embedded SQL". The annex of [Ans86] contains formal definitions of embedded SQL for various programming languages (e.g. Cobol, Fortran, Pascal).

The obvious advantage of such embedded languages is that their syntax does not depend on the host language. If the concepts and syntax of the language are known to the programmer then applications can be written in different host languages without additional efforts for learning a new database programming language. In the special case of embedded SQL the major advantage is, that SQL is a well known and broadly accepted interactive query language. Embedding this language into a host language results in an effective combination of the advantages of a procedural with a non procedural language.

```
#include <stdio.h>

int get_children(f)
    long f;
{
    EXEC SQL BEGIN DECLARE SECTION;
        long x, y;
        long w, h;
        long father;
    EXEC SQL END DECLARE SECTION;

    father = f;

    EXEC SQL WHENEVER SQLERROR GOTO db_err;
    EXEC SQL CONNECT 'demo' IDENTIFIED BY 'secret';

    EXEC SQL DECLARE CURSOR rec_cursor FOR
        SELECT    x_coord, y_coord, width, height
        INTO :x, :y, :w, :h
        FROM rectangle, is_child
        WHERE father_id = :father
        AND       child_id = rec_id;

    EXEC SQL OPEN rec_cursor;
    EXEC SQL FETCH rec_cursor INTO :x, :y, :w, :h;

    while (SQLERROR != SQL_NO_DATA)
    {
        printf ("x = %ld, y = %ld, w = %ld, h = %ld\n", x, y, w, h);

        EXEC SQL FETCH rec_cursor INTO :x, :y, :w, :h;
    }

    EXEC SQL CLOSE rec_cursor;
    EXEC SQL COMMIT WORK RELEASE;

    return (0);
dberr:
    ...
    return (1);
}
```

Figure 2: *Sample procedure in embedded SQL*

From the conceptional point of view embedded systems reveal several problems. The uniformity of the language disappears, database and program variables are treated differently (e.g. declaration). Another conceptional deficiency is brought about by the fact that generally, the result of a SQL query is a relation (i.e. a sequence of rows in a table) and that in a conventional programming language only one item can be processed at a time. To combine these contrary principles, the concept of a *cursor* has been introduced. A cursor is declared in conjunction with a transaction and can be used to reference the single rows of a resulting relation.

Compared with the host language dependent approach as presented in the previous section, programs written in embedded SQL tend to become much shorter and easier to read than their host language dependent analogies. Figure 2 contains an embedded SQL procedure with the same functionality as the procedure described in Figure 1. Reduced program length and improved readability are evident.

4. SQL for Db++

In this section the major concepts of our extensions are presented. We realised the host language independent approach as detailed in section 3.3. The unique "db++ approach" reveals numerous problems. Thus fundamental decisions and compromises are discussed and justified.

Data Types

As mentioned in section 2.2 there is a high correspondence between db++ data types an data types in C. Since SQL is independent of any programming language, which also applies to the data types, we had to establish a mapping of SQL data types to db++ data types. This mapping is defined in Table 1. The upper half comprises all the SQL data types as specified in [XPG87], the lower half contains additional data types supported in our extension.

SQL data type	db++ data type
CHAR (number)	FT_STRING
SMALLINT	FT_SHORT
INTEGER	FT_LONG
DECIMAL(precision,scale)	FT_SHORT, FT_LONG, FT_FLOAT, FT_DOUBLE
FLOAT	FT_DOUBLE
DATE	FT_LONG
LONG	FT_STRING
NUMBER (precision [,scale])	FT_SHORT, FT_LONG, FT_FLOAT, FT_DOUBLE
RAW(number)	FT_STRING
VARCHAR(number)	FT_STRING

Table 1: *Mapping of SQL data types to db++ data types*

One of these special data types is the SQL data type *LONG* permitting storage of information with "unlimited" variable length. Since db++ does not support such data types we had to introduce a new mechanism. The basic idea was to combine db++ with the UNIX file system thus transforming db++ to a hybrid database [Pen87a]. An entry in a relation of type LONG (respectively FT_STRING) represents the name of a UNIX file (a unique name for each entry), whereby the UNIX file contains the essential information. This storage technique remains totally transparent to the user provided that he accesses the database through the SQL interface.

Inspecting Table 1 it becomes clear that the mapping defined is not reversible. In other words, it is impossible to determine the appropriate SQL data type if merely the db++ data type is given (e.g. FT_LONG can stand for INTEGER, DATE or NUMBER). But exactly this knowledge is required if data is extracted from the database and converted to a corresponding SQL data representation.

To overcome these problems we adopted the concept of a centralised data dictionary.

Data Dictionary

A data dictionary is part of the database and used to store information relevant for maintaining and managing the database. A data dictionary contains the names of all existing relations, views, and detailed descriptions of the relation's and view's columns. Besides, this dictionary could be used to define the access rights of each user to each relation.

Transactions, Multi-User Operation, and Locking

To allow bounded transactions (begin, commit and rollback) each relation is extended by default with two additional but hidden columns. This enhancement supports single tuple locking and thus transparent multi-user operation. The semantics of the two additional attributes is as follows:

(1) *OID* (owner identity) defines by which process the row is currently owned (zero indicates that the row can be accessed by any process).

(2) *MID* (modifier identity) determines the process currently modifying the row (zero indicates that the row can be modified by any process).

Figure 3 contains a symbolic excerpt of our sample database as it could appear during a transaction updating a single row. The snap shot is taken immediately before a transaction commit respectively rollback. A process *A* is updating the rectangle number 1(rec_id = 1), however the update has not yet been committed. Thus, the corresponding row is contained twice in the table (as indicated by the two right hands in the first column of the table). For each process except process A the original row (the one with OID equal to zero) is visible, only process A "sees" the modified one. As long as process A has not finished its transaction, the original row has the status read only for the remaining processes.

A detailed description of this algorithm is contained in [Had89a].

rectangle	rec_id	x_coord	y_coord	width	height	*OID*	*MID*
⇒	1	20	30	45	45	*0*	*A*
	2	13	18	22	16	*0*	*0*
	3	27	91	79	35	*0*	*0*
⇒	1	30	30	45	45	*A*	*0*

Figure 3: *A relation during update*

Database User

The question whether the UNIX file permissions are adequate to realise the database user concept is equivalent to the question whether our implementation should adhere to [XPG87] in that point or not. Since [XPG87] requires that granting privileges to other users can take place at the level of a single column (at least in the case of an UPDATE privilege), it becomes obvious that the conventional UNIX file permissions are insufficient.

As suggested above, the data dictionary is appropriate for the realisation of the more sophisticated user privilege mechanisms. There remains, however, one open question: How should the conflict between the UNIX file permissions and the newly introduced and more granular privilege mechanisms be resolved? In other words: The overall system still remains an *open* system in the sense that all the relations (represented by UNIX files) can be accessed using different UNIX tools. Can UNIX file permissions be abused to supersede privileges defined in the SQL context?

A reasonable solution is that all relations belong to a virtual UNIX user, say db++, and, that access to these relations is granted exclusively through the server. From the viewpoint of security this solution is best suited. The usage of the db++ tools is then restricted to the user db++, i.e. to some kind of database super user. A database object is still possessed by a database user but this database user does not correspond to the real UNIX user.

Server

Consistently managing a database in multi-user mode can be effectively achieved by establishing some kind of server. In our approach each database (i.e. a collection of interrelated relations stored in a UNIX directory) is associated with one database server. Each application operating on the database is connected with that server. The server is automatically generated by the first application that intends to access the database. After the last application has closed the connection to the server, the server exits.

The server is responsible for:

(1) Performing elementary database requests (i.e. read, write, update, etc.)

(2) Locking tuples and controlling multiple access

(3) Managing and updating the data dictionary

(4) Detecting and resolving deadlocks

Static vs. Dynamic Command Processing

The processing of SQL commands can either be carried out statically, i.e. the command is interpreted during compile time, or dynamically, i.e. the command can be generated at run time and is then interpreted by the application.

Our approach supports both methods. If the special SQL statements for dynamic commands are used the command is interpreted at run time otherwise the command is transformed to an internal statement tree at compile time.

5. Implementation

Conformance with the Standards

[Ans86] distinguishes between two levels of conformance. Level 2 conformance implies level 1 conformance. Thus level 1 conformance is much easier to achieve. Even [XPG87] has not adopted all requirements necessary to conform to level 2. One of the major problems at level 2 is concerned with the serialisability of transactions that guarantees avoidance of transaction anomalies. Thus [XPG87] excluded this requirement.

Our implementation is designed to fully conform to [XPG87]. There is only one discrepancy which is not yet eliminated: Db++ demands that the key fields determining an index file have to be unique for each tuple (i.e. it is not possible to insert two tuples with the same index), whereas [XPG87] is not that restrictive. In our extension indices are forced to be unique.

Development Process

Figure 4 describes the development process of a program written in embedded SQL. The precompiler converts the embedded SQL program (*prog.pc*) into a conventional C program (*prog.c*), i.e. all embedded SQL statements are reformulated to regular C procedures. The resulting C program is then compiled and linked with the special SQL library. The bold boxes in the figure indicate the newly introduced components.

The Precompiler

The principal functionality of the precompiler is as follows: Each line in *prog.pc* containing an embedded SQL statement is parsed and transformed to an internal statement tree. The parser has been developed with lex and yacc.

For each type of statement (e.g. SELECT, DROP) we provide one executing function. All these functions are part of the library (as depicted in Figure 4) and will be invoked in the resulting program *prog.c*. Parameters passed to these functions are the corresponding statement tree and the actual variables.

Building the statement tree at compile time permits syntax checking before run time.

At run time the statement tree is interpreted by the application, i.e. merely basic requests are passed to the database server which accesses the database through the raccess library. The main reason for this approach is to keep the server as simple as possible and to avoid complicated scheduling mechanisms which would be necessary if an application request consumed a larger amount of time.

6. Summary

This paper presented various extensions to the relational database system db++. The main goal was to combine sthe obvious advantages of sophisticated database management systems like ORACLE with the UNIX oriented and thus much better integrated database system db++.

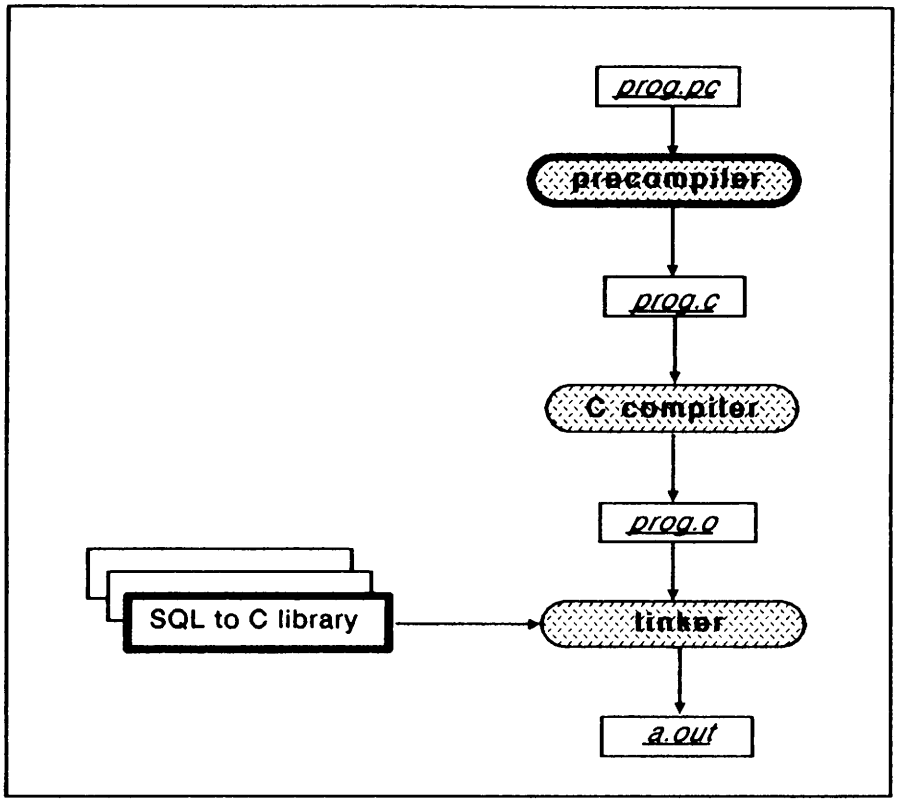


Figure 4: *Developing an embedded SQL program*

Db++ has been enriched with a variety of new features (transaction processing, data dictionary, multi-user operation). The central point is an embedded SQL programming interface for C, which closely adhere to the X/Open standard [XPG87] for embedded SQL and Oracle's PRO*C [Nev87a].

The major benefit of our extension is the reduced effort for program development and maintenance under db++.

7. Acknowledgements

The authors wish to thank Guenter Gruensteidl, Heinz Kantz, Gernot Kunz, Karin Schneider, and Werner Schuetz for valuable comments on an earlier version of this paper. We also want to acknowledge the support of Concept ASA providing us with useful informations about db++ internals.

References

[Con86a] Concept ASA, *Db++ Relational Database Management System – User Guide*, Frankfurt-Main, 1986. Second Edition

[XPG87a] *X/Open Portability Guide – Relational Database Language (SQL)*, January 1987.

[Ans86a] American National Standard for Information Systems, "Database Language – SQL," ANSI X3.135-1986.

[Agn86a] M. Agnew and R. Ward, "The Db++ Relational Database Management System," *Proc. of the European UNIX User Conference*, pp. 1–15, Florence, Italy, April 1986.

[Agna] M. Agnew and R. Ward, *Db++ Relational Database Management System – The Raccess Library Routines – The C Language Interface Reference*, Concept ASA, Frankfurt-Main $D April 1988. Third Edition

[Ber81a] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, vol. 13, no. 2, pp. 185–221, June 1981.

[Cas89a] R. J. Casimir, "Fourth Generation Problems," *Sigplan Notices*, vol. 24, no. 5, pp. 83–86, May 1989.

[Cod85a] E. F. Codd, "Is your DBMS really relational," *Computerworld*, October 14, October 21, 1985.

[Had89a] T. Hadek, "Eine hochwertige Programmierschnittstelle für das relationale Datenbanksystem Db++," Master Thesis (in German), Technical University Vienna, Vienna, Austria, to be published December 1989.

[Mis88a] S. K. Misra and P. J. Jalics, "Third-Generation versus Fourth-Generation Software Development," *IEEE Software*, vol. 5, no. 4, pp. 8–14, July 1988.

[Nev87a] D. Neville, *PRO*C User's Guide*, April 1987. ORACLE Part No. 3504-V1.1

[Pen87a] M. Penedo, "Prototyping a Project Master Database for Software Engineering Environments," *Proc. of the 2nd Software Engineering Symposium on Practical Software Engineering Environments, ACM Sigplan Notices*, vol. 22, no. 1, pp. 1–11, January 1987.

Processable Multimedia Document Interchange using ODA[1]

Jaap Akkerhuis
Ann Marks
Jonathan Rosenberg
Mark S. Sherman

Information Technology Center
Carnegie Mellon University
Pittsburgh, USA
jaap+@andrew.cmu.edu
annm+@andrew.cmu.edu
jr+@andrew.cmu.edu
mss+@andrew.cmu.edu

ABSTRACT

The EXPRES (Experimental Research in Electronic Submission) project promotes the electronic interchange of multi-media documents among the scientific research community. For this project we concentrate on the problem of effective interchange of processable multi-media documents. In particular, we are ignoring the transfer method. Instead we concern ourselves with the question of how a multi-media document created on one system can be viewed and edited on another system.

The obvious technique of performing translations between each pair of systems is impractical. In order to attack the problems efficiently, we make use of a standard representation. We have settled on the international standard Office Document Architecture (ODA) [ISO88a] as the intermediate format. This paper discusses how we implemented ODA for interchange.

Introduction

In the last decade there has been an explosion in the number of multimedia document processing systems. These range from simple batch text processors systems to fancy WYSIWYG multimedia editors; these are used by professional typesetters, computing professionals and administrative personnel. Most systems have "multimedia" facilities, which range from the ability to use different fonts, inclusion of drawings, and mathematical equations, up to sound and video. The term "multimedia document" is most of the time actually a misnomer. Basic facilities such as non-proportional spacing, different fonts, etc. have always been part of a document. We will use this term however to discriminate from the typewriter and lineprinter style documents.

The introduction of these systems has generated a new problem. To interchange a document from one system to the other is hardly possible given the number of systems in use.

The National Science Foundation (NSF) receives yearly a considerable number of proposals for research grants. Most of these are actually prepared on the above mentioned systems. This observation leads to the question of whether it would be possible to receive these proposals in electronic form, so that the amount a paper involved could decrease. Ideally, the electronic form should make it possible to process the documents, without requiring the submitters to standardise on a single system. This led to start of the Experimental Research in Electronic Submission (EXPRES) project, whose goal is to investigate the problems of document interchange for dissimilar systems. Main participants are the Information

[1] This work was funded by the US National Science Foundation under grant ASC-8617695. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the US Government.

Technology (ITC) Center at Carnegie Mellon University (CMU) and the Center for Technology Integration (CITI) at the University of Michigan (UM).

As part of the Andrew project [Mor86a], the ITC developed a multimedia toolkit, the Andrew Toolkit (ATK) [Pal88a], which supports objects such as multi-font text, line drawings, equations, spread sheets, and raster drawings. At the University of Michigan, the Diamond multimedia system [Tho85a] is the base for their collaboration in the project. Although Diamond and the Andrew systems are quite similar in their capabilities, they were independently developed and quite different in their underlining implementations. This provided an ideal environment for the EXPRES project.

1. Translation fidelity

When translating a document from one system to be viewed and edited on another, it is necessary to decided on the fidelity required. We distinguish three major fidelity categories for document interchange. These are *imagining fidelity, structural fidelity* and *editing fidelity*. Below we briefly discuss what these entail. An more thorough discussion may be found in [Ros89a].

1.1. Imaging fidelity

Imaging fidelity can defined by how close the translated document matches the original in appearance when printed or on the screen. For certain types of document this is a primary requirement, as for instance legal documents where changes in the layout are often unacceptable. This type of fidelity is also what naive users want from a translation system.

To achieve this, one can use a standard page description language, such as PostScript, Interpress or DVI, as produced by TeX. Of course this assumes that the implementation of these language on the receiving end will produce the the same result. This might not always be the case. For example, if the more or less standard Computer Modern Roman fonts for TeX are not used, this scheme will fail.

1.2. Structural fidelity

Normally a document is highly structured. A document consists of paragraphs, figures with legends, footnotes etc. Maintaining the structure of the document allows the receiver to format the document differently than the originator. This way one can retain the general appearance of an document.

1.3. Editing fidelity

Editing fidelity requires structural fidelity, but, in addition, the document must be editable in a way similar to the originating system. This is particularly important for the EXPRES project since we are concerned with allowing collaboration on multimedia documents from dissimilar systems.

A prime example of an editing feature to be retained during translation is *style sheet* or *property sheet* information. A lot of document systems provide a mechanism for defining styles. For example, one can define a quotation style where the right and left margin are indented and the font changed to italic. This style can then be applied to various parts of the text. The important fact is that when this style's definition is changed, this will take effect on the parts of the text where the style is applied. To elaborate, let's assume that we have a document on system A, and that the document includes a definition for a quotation style which is applied several places in the document. Let's assume that we now interchange the document to system B, and that the interchange preserves editing fidelity. On system B, an edit is made to change the quotation style. Now all applications of the quotation style will appear different in the document on system B. Of course it must be possible to interchange these changes to the quotation style when sending the document back to the original system or to any other system.

2. The Office Documentation Architecture

It is obvious that it is impractical to build $n \times (n-1)$ translators for n document processing systems. We decided to translate in and out a common format for all translators. For the this format we choose the Office Documentation Standard (ODA), an international standard designed for interchange of multimedia documents. One of the main reasons is that it doesn't only specify the logical structure of a document but also includes full semantics to specify the layout of a document. We felt that this was necessary in the interchance, since users would insist on the ability to specify the appearance of the document. In addition, ODA is an international standard that has a following in Europe. This offered us the possibility of extending our interchange to others.

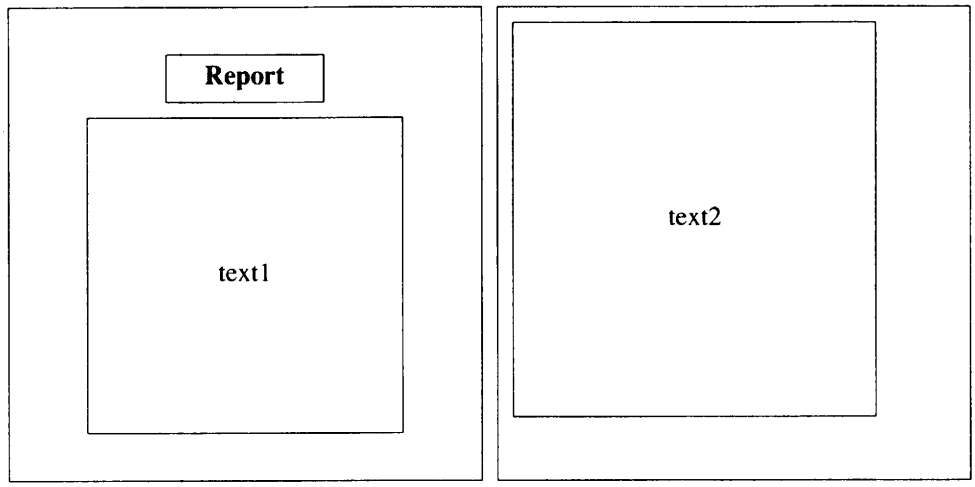


Figure 1: *The sample document*

ODA defines a document architecture, several content architectures and two data stream formats. The *document architecture* is the means by which the structure of a document, independent of its actual content, is represented. In general, an ODA document is represented using two sets of structures.† The *logical structure* is based on the meaning of various divisions of the document. For example, the logical structure of a document might consist of chapters, sections and paragraphs. In the *layout structure*, the document is structured on the basis of presentation. For example, the layout structure of a document might consist of pages and, within the pages, frames and blocks that define headers, footers and paragraphs.

In addition, each structure may exist in two forms: *generic* and *specific*. A generic structure may be thought of as a template or macro that allows structure information to be collected and referenced. For example, the *generic logical structure* of a document might indicate that the document consists of a title, followed by one or more sections, followed by a set of references. Correspondingly, a *generic layout structure* for the same document might indicate that the title is a block that appears two inches from the top of the first page and is centered.

If the generic structures of a document can be thought of as macros, then the specific structures represent invocations of those macros. The *specific logical structure* is, thus, the actual structure of a document. For example, the specific logical structure might show that a particular document consists of a title, five sections and a set of seven references. There is a *specific layout structure*, corresponding to the generic layout structure, but it is used only for the representation of a final form document (one that may be imaged). Since we are concerned only with editable documents, our translation schemes do not use any specific layout structures. The actual content of an ODA document consists of instances of *content architectures*. Each content architecture defines its own internal structure, which may consist of logical and layout structures. There are currently three content architectures defined within ODA. *Character content architecture* defines the presentation and processing of characters and allows the specification of graphic character sets, multiple fonts, ligatures and formatting directives such as indentation and justification. *Raster graphics content architecture* defines pictorial information represented by an array of picture elements. *Geometric graphics content architecture* defines picture description information such as arcs and lines.

A *data stream* is an out-of-memory representation for a document that is suitable for storage in a file or transmission over a network. The ODA standard defines an ASN.1 binary data stream format known as the Office Document Interchange Format (ODIF).‡

† It is possible for an ODA document to consist of only one of these sets of structures. For our purposes, this is immaterial and we will only consider documents containing both sets of structures.

‡ ODA defines another data stream representation, the Office Document Language (ODL), which is a clear text representation that conforms to the Standard Generalized Markup Language standard (SGML). Note that this does not imply that there is a direct relationship between an ODA document and the equivalent document marked up using SGML.

Documents represented in ODA are graphs, the nodes of which are known as *constituents*. Each constituent has a set of *attribute-value* pairs. The values of attributes are used to represent the structure of the document. Attributes have values that control the presentation and layout of the document. For example, the value of the attribute "Separation" at a constituent will control the distance between blocks of text when the document is displayed or imaged.

Figure 1 displays a small example of a two page document. It consists of two pages. The first page contains a title "**report**" that is centered and bold-faced, and a centered paragraph, "text1". The second page contains a left justified paragraph "text2".

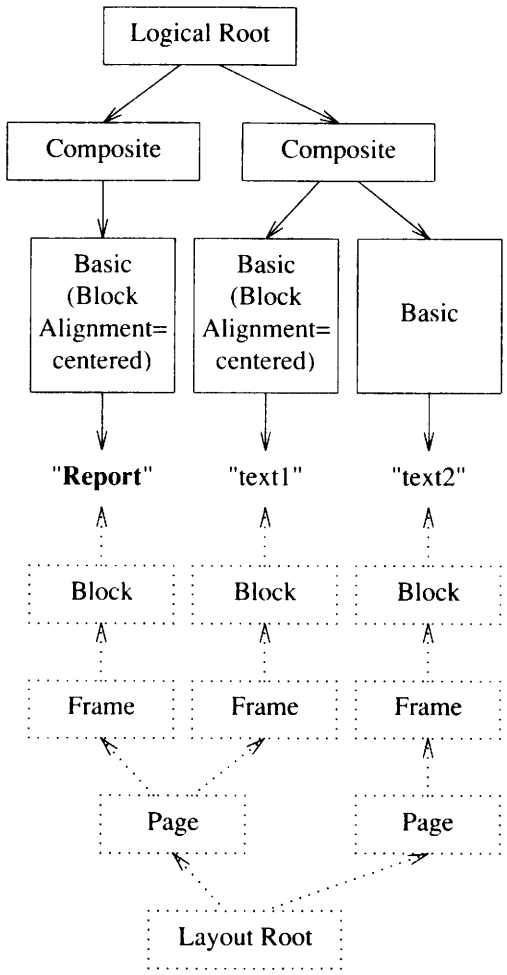


Figure 2: *The ODA specific structure*

The corresponding specific logical and layout structures are shown in Figure 2. The logical structure consists of a *composite* object for each section and a *basic* object for each paragraph. The centering of the text is accomplished by attaching the attribute "Block Alignment" with the value "centered" to two of the basic objects. The specific layout structure consists of two *page* objects each of which contains a *frame* and a *block* object to position the paragraph.

3. The CMU ODA Tool Kit

To make it easier to build translators we developed a subroutine library for manipulating ODA documents. The Tool Kit [Rosa], written in C, includes C language definitions for the objects that occur in ODA, such as constituents and sequences. The Tool Kit also includes data type definitions of all the data used in ODA, for example construction expressions and font information. The Tool Kit provides subroutines for manipulating ODA structures. For example, the Tool Kit permits the creation of documents and components; it allows the user to associate an attribute value with a constituent; subroutines for reading and writting the binary ODIF interchange format are also included.

The Tool Kit performs a number of useful functions for dealing with ODA. When setting an attribute value, the Tool Kit performs full semantic checking to ensure that the attribute can be associated with the given constituent, and that the value specified for the attribute is legal. This is extremely useful because some attributes are only allowed on certain kinds of constituents, so the Tool Kit will prevent the creation of illegal combinations. The ODA standard also specifies a complicated scheme for defaulting of attribute values. These defaulting rules are fully supported by the Tool Kit. The Tool Kit includes subroutines for reading and writing of the ODIF data stream, operations which are complex given that the ODIF stream is a context sensitive binary representation. For debugging purposes it is possible to create a human readable representation of the binary data stream. Service routines are also included to support the ISO 9541 standard for fonts which is used by ODA.

The subroutine library is designed to be highly portable, therefore it is written in a subset of C which we carefully specified with portability in mind. The Tool Kit will compile and run on various operating systems and hardware platforms. By carefully separating the operating system dependencies, such as I/O, into separate modules, it is easy to port to other systems. In addition, hardware dependencies are localised to a short set of type definitions. Currently there is support for:

1 UNIX (System V and BSD flavors) on Vaxes, IBM-RT, Sun

2 VAX VMS

3 Macintosh under MPW

4 IBM PC running MS-DOS

We estimate that it is less than a day's work to bring it up on a new machine.

Although the Tool Kit is very useful, there are many capabilities that it does not currently include. For example, there is no capability for interpreting content. The actual content of the document, be it text or a raster, is a sequence of bytes. Translator implementors must examine content sequences to extract formating information such as font changes. The Tool Kit does not include any of the conventional document notions such a paragraph or left margin. Such higher level document constructs must be built by creating the appropriate ODA structure and attaching the required attributes. There are some document operations that would make nice additions to the Tool Kit. For example, it would be convenient to be able to have a single operation for instantiating a generic object when constructing an ODA document. Some of the information required by ODA is very cumbersome, e.g. construction expressions and font information. It would be nice to be able to specify these concisely. The Tool Kit does not perform the layout or imaging processes included in ODA's document reference model. Both could be built on top of the Tool Kit. Finally, there is no support for the ODL SGML based interchange format.†

4. Examples of Tool Kit Use

In this section, we present four examples illustrating the use of the Tool Kit to construct translators. These four examples are paired to show similar processing operations when translating from a native document format to ODA and when translating from ODA back into a native format. The first pair of examples shows how a part of the specific logical structure is created or examined. In the second pair, we show how to associate attribute values with constituents or how to retrieve attribute values. Throughout, we use a C-like notation to present program segments. We omit checking of Tool Kit return values to keep the examples uncluttered.

4.1. Example of Document Structure

In this example, we assume that the document being interchanged is to contain processable information. In ODA, this is represented using the logical structure. We further limit ourselves to the specific logical structure to illustrate how the structure is built up when translating to ODA, and how the structure is interpreted when translating from ODA. Figure 3 depicts the ODA structure. Here we assume that there is a parent component that is a composite logical object. The children of this parent are also to be composite logical objects.

† Although the binary ODIF representation of a document is cryptic and unreadable by a human, it is also much easier to parse and unparse than ODL. In addition, all other ODA implementations of which we were aware were using ODIF and we would, thus, have some chance of interchanging with other systems. For these reasons, we have only implemented reading and writing of ODIF within the ODA tool Kit.

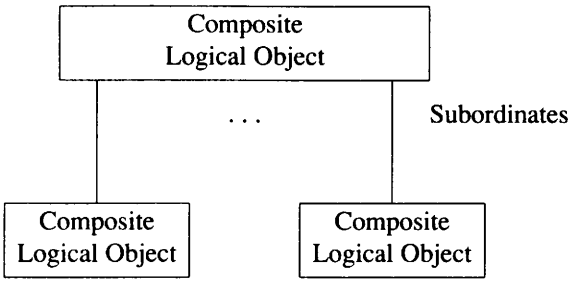


Figure 3: *ODA Structure for Examples 1 and 2*

```
/*
 * At this point, a parent composite logical object is
 *  to be created.
 * The children, also composite logical objects, are also
 *  to be created.
 * The subordinates attributes is to connect the parent to
 *  the children.
 * document is assumed to be of DOCUMENT_type and created by
 *  a call to the Tool Kit MakeDocument routine.
 */

INT_type ReturnCode;     /* for Tool Kit returns */
CONSTITUENT parent;      /* the parent */
CONSTITUENT child;       /* one of the children */
SEQUENCE_CONSTITUENT_type Subordinates;
                         /* the value for the subordinates attribute */

    /* create the parent */
    Parent = MakeComponent( document, SPECIFIC_COMPONENT,
              at_OBJECT_TYPE_comp_logical_obj );

    /* create the empty sequence for the subordinates attribute */
    Subordinates = MakeSequence( SEQUENCE_CONSTITUENT_tag, (INT_type) 0 );

    /* loop to create the children and add to the subordinates */
    for( each child needed ) {
        /* make the child component */
        Child = MakeComponent( document, SPECIFIC_COMPONENT,
                  at_OBJECT_TYPE_comp_logical_obj );

        /* expand the subordinates sequence */
        ReturnCode = ExpandSequence( Subordinates, (INT_type) 1 );

        /* add the Child at the end of Subordinates */
        Subordinates->sequence_value.constituents[Subordinate->length-1]
                                                              = Child;
    }

    /* now set the subordinates attribute */
    ReturnCode = SetAttr( parent, at_SUBORDINATES,
                    (POINTER_type) Subordinates,
                    (PARM_MASK_type) 0 );
```

Example 1: *Creating Components and adding the Subordinates Attribute*

4.1.1. Translating into ODA

To translate into ODA, the native document must be traversed, and the appropriate ODA structures constructed. We assume in this example that the traversal has reached a point where the parent component is

to be built, the children are to be created, and the subordinates attribute is to be associated with the parent. The code for accomplishing this is given in Example 1.

In this example, we see calls to the Tool Kit MakeComponent routine which creates a component. MakeComponent creates the component in the given document; the component will be of a given type, a specific component in this example; the component will be of a given kind, a composite logical object in this example. The Tool Kit MakeSequence routine is used to create a sequence of constituents to hold the value of the subordinates attribute. Initially, this sequence has length zero, but the length is increased by one as each child is created. Finally, the SetAttr routine is used to set the value of the parent's subordinate attribute. Because the subordinates attribute does not have parameters (as does the offset attribute, for example), a null value is passed as the last parameter.

4.1.2. Translating out of ODA

When translating out of ODA, the ODIF data stream will first be read from a file by calling the ReadODIF Tool Kit routine. This will result in the creation of a document with type DOCUMENT_type. This example shows how the data stream is read, and how to examine the children encountered during the traversal. As the traversal is performed, the native form of the document is constructed; we omit code for doing this. The code is shown in Example 2.

In this example, we see that the entire ODIF data stream is read by a single call to the Tool Kit routine ReadODIF. This creates a DOCUMENT_type object that is the document contained in the ODIF data stream. To locate the root of the specific logical structure, the FindDocumentRoot is called. At this point the recursive traversal begins. To traverse each constituent in the specific logical structure, each constituent is processed; this processing will entail the creation of the appropriate native document format piece. To continue the traversal, the value of each constituent's subordinate attribute is obtained, and traverse is called for each subordinate.

The Tool Kit provides an alternate way to traverse document structures using the Tool Kit ITERATOR_type object. In Example 3, we outline how an ITERATOR_type can be used for this purpose.

Note that this example begins like the previous one with the reading of the data stream and the locating of the document's specific logical root. The iterator is then created, and the iteration is performed. This iteration will result in the entire specific logical structure being traversed. Note that the traversal will be parent first, like the previous example, but, here the traversal is breadth first where the previous example is depth first. Finally, this example illustrates an iterative way to traverse document structure in contrast to the previous example which used recursion.

4.2. Example including an Object Class and a Style

The next pair of examples is based on the ODA structure shown in Figure 4. Here we have three constituents: a basic logical object, a basic logical object class and a presentation style. The basic logical object indicates that it is an instance of the basic logical object class by the object class attribute. The basic logical object class has an associated presentation style as indicated by the presentation style attribute. The presentation style has one attribute associated with it, the character content architecture attribute indentation. The indentation attribute has value 5 which, according to ODA semantics, is in standard measurement units.

4.2.1. Translating into ODA

This example illustrates how the structure shown in Figure 4 can be created. The native format document is being traversed. At some point, in this traversal the structure shown in Figure 4 needs to be created to represent the native format document. The code for doing this is shown in Example 4.

Note that each constituent is created by a Tool Kit call. The two components are created using MakeComponent, but the presentation style must be created using MakeStyle. The attribute values are set using various flavors of the SetAttr routine. To set the object class attribute for the basic logical object, and to set the presentation style attribute for the basic logical object class, the SetAttr routine is used. To set the value of the indentation attribute on the presentation style, we have used the SetIntAttr routine. This permits us to pass the value of the attribute rather than the address of an INT_type variable with value 5 which SetAttr would require.

```
/*
 * A data stream is read.
 *
 * The document specific logical root is located.
 *
 * A depth first, parent first traversal is performed on
 *  the specific logical structure.
 */

INT_type ReturnCode;        /* Tool Kit return value */
DOCUMENT_type document;     /* the document */
CONSTITUENT LogicalRoot;    /* the root of the specific logical structure */
                    .
    /* first read in the document */
    document = ReadODIF( fileno(stdin) );
    /*
     * Here we assume that this is running on UNIX
     *  and that the data stream is on the standard input.
     */
    /* now locate the document logical root */
    LogicalRoot = FindDocumentRoot( document, SPECIFIC_DOC_LOGICAL_ROOT );

    /* call subroutine traverse to examine the root */
    traverse( LogicalRoot );

void traverse( constituent )
CONSTITUENT constituent;
{
    /*
     * Traverse the given constituent.
     *
     * The appropriate part of the native format would
     *  be created but this is not shown.
     */

    INT_type i;      /* for looping through the children */
                     /* the constituent's subordinates */
    SEQUENCE_CONSTITUENT_type Subordinates;
    INT_type ReturnCode;     /* return code from the Tool Kit */

    process the parent;

    /* now get the parent's subordinates attribute */
    ReturnCode = GetAttr( constituent,
                    at_SUBORDINATES,
                    (POINTER_type) &Subordinates,
                    BOOL_false,    /* do not use the ODA defaulting rules */
                    (PARM_MASK_type *) 0 ) ;

    /* now start the iteration over the children */
    for( i = (INT_type) 0; i < Subordinates->length; i++ ){
            /* recursively traverse the child */
        traverse( Subordinates->sequence_value.constituents[i] );
    }
```

Example 2: *Reading a Data Stream and Traversing the Document*

4.2.2. Translating Out Of ODA

To translate out of ODA, the ODA document is traversed. Presumably, at some point the basic logical object is encountered, and the value of the indentation attribute is needed. Example 5 shows how to obtain the value of the indentation attribute.

```
DOCUMENT_type document;          /* the document */
ITERATOR_type iterator;          /* the iterator */
CONSTITUENT constituent;         /* a constituent */

    /* first read in the document */
    document = ReadODIF( fileno(stdin) );
    /*
     * Here we assume that this is running on UNIX
     *  and that the data stream is on the standard input.
     */

    /* now locate the document logical root */
    constituent = FindDocumentRoot( document, SPECIFIC_DOC_LOGICAL_ROOT );

    /* make the iterator */
    iterator = MakeSubgraphIterator( constituent,
                  PARENTS_FIRST,     /* the parent goes before the children */
                  BREADTH_FIRST );   /* the traversal is to be breadth first */

    /* now begin the iteration */
    for( constituent = NextConstituent( iterator );
            constituent != ERROR_CONSTITUENT
                           && constituent != NULL_CONSTITUENT;
            constituent = NextConstituent( iterator ) ){

                process constituent;
    }
```

Example 3: *Reading a Data Stream and Traversing the Document using an Iterator*

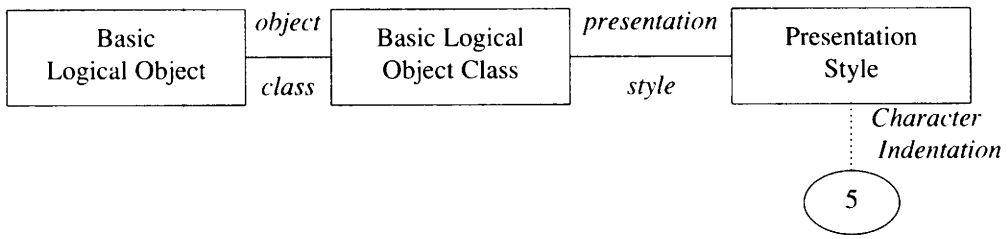


Figure 4: *ODA Structure for Examples 4 and 5*

We see that only one Tool Kit routine is called, GetAttr. The value of indentation for the BasicLogicalObject can be obtained easily by using the ODA defaulting mechanism which is implemented by the Tool Kit. Without Tool Kit support for ODA defaulting, it would be necessary to look for styles, object classes, resource documents, default value lists, document application profile defaults, and to know the ISO 8613 default values for all attributes that have default values.

5. Conclusions and Status of the CMU ODA Tool Kit

The ODA Tool Kit enabled us to interchange documents between different platforms and between different document processing systems in timely fashion. Having the ODA Tool Kit as a common base permitted us to interchange much sooner than we would outwise have been able done otherwise. In addition, many difficulties were eliminated because we were all using the same Tool Kit.

We plan to release the Tool Kit on the next MIT X tape, which is currently scheduled for release in December 1989, although MIT is controlling the date of the release. We are also investigating the possibility of releasing the Tool Kit through other publically available channels, possible the ISODE distribution. On release the Tool Kit will be largely complete. The functionality that we presently expect to be missing or limited includes: limited support for ODIF, most notably the document profile and specific layout structure will be largely incomplete or missing; the Tool Kit will include no support for swapping of

```
/*
 * First create the constituents, then set the appropriate
 *  attributes.
 *
 * document is a DOCUMENT_type object created by a
 *  call to the Tool Kit routine MakeDocument.
 */
CONSTITUENT BasicLogicalObject;
CONSTITUENT BasicLogicalObjectClass;
CONSTITUENT PresentationStyle;
INT_type ReturnCode;    /* Tool Kit return code */

    /* make the basic logical object */
    BasicLogicalObject = MakeComponent( document,
                SPECIFIC_COMPONENT,    /* in the specific structure */
                at_OBJECT_TYPE_bas_logical_obj );

    /* make the basic logical object class */
    BasicLogicalObjectClass = MakeComponent( document,
                GENERIC_COMPONENT,     /* in the generic structure */
                at_OBJECT_TYPE_bas_logical_obj );

    /* make the presentation style */
    PresentationStyle = MakeStyle( document,
                PRESENTATION_STYLE );

    /* now associate the basic logical object with the object class */
    ReturnCode = SetAttr( BasicLogicalObject,
            at_OBJECT_CLASS,    /* the attribute to be set */
            (POINTER_type) BasicLogicalObjectClass,
                                /* the value of the attribute */
            (PARM_MASK_type) 0 );
                /* the object class attribute does not have parameters */

    /* now associate the basic logical object class with the style */
    ReturnCode = SetAttr( BasicLogicalObjectClass,
            at_PRESENTATION_STYLE,    /* the attribute to be set */
            (POINTER_type) PresentationStyle,
                                /* the value of the attribute */
            (PARM_MASK_type) 0 );
                /* the presentation style attribute does not have parameters */

    /* now add the indentation style value to the style */
    ReturnCode = SetIntAttr( PresentationStyle,
            cc_INDENTATION,     /* the attribute to be set */
            (INT_type) 5,       /* the value of the attribute */
            (PARM_MASK_type) 0 );
                /* the indentation attribute does not have parameters */
```

Example 4: *Building the ODA Structure shown in Figure 4*

parts of the ODA document, a feature important for machines with limited memory, or when working with huge documents; no ability to evaluate the expressions included in ODA, e.g. string expressions, numeric expressions etc.; the Tool Kit only supports text and raster content. At present, the Tool Kit is about 80,000 lines of C.

References

[ISO88a] ISO, *Office Document Architecture (ODA) and Interchange Format (ISO 8613)*, International Organization for Standardization (ISO), 1988.

[Mor86a] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, vol. 29, no. 3, pp. 184–201, March 1986.

```
/*
 * BasicLogicalObject is the basic logical object as
 *  shown in Figure 4.
 */
INT_type ReturnCode;     /* Tool Kit return value */
INT_type Indentation;    /* value of indentation */

    ReturnCode = GetAttr( BasicLogicalObject,
                          /* the attribute whose value is sought */
            cc_INDENTATION,
                          /* where to return the value of indentation */
            (POINTER_type) &Indentation,
            BOOL_true,    /* use the ODA defaulting rules */
                          /* do not return the parm mask */
            (PARM_MASK_type *) 0 );
```

Example 5: *Extracting the Value of Indentation for the BasicLogicalObject*

[Pal88a] Andrew. J. Palay, Wilfred J. Hansen, Mark Sherman, Maria G. Wadlow, Thomas P. Neuendorffer, Zalman Stern, Miles Bader, and Thom Peters, "The Andrew Toolkit—An Overview," *Proceedings of the USENIX Winter Conference*, pp. 9–21, USENIX Association, Berkeley, CA, February, 1988.

[Ros89a] Jonathan Rosenberg, Mark S. Sherman, Ann Marks, and Frank Giuffrida, "Translating Among Processable Multi-media Document Formats Using ODA," *Proceedings of the ACN Conferenrence on Document Processing Systems*, pp. 61–70, ACM, New York, December 5-9, 1989.

[Rosa] Jonathan Rosenberg, Ann Marks, Mark Sherman, Paul Crumley, and Maria Wadlow, "The CMU ODA Tool Kit: Site Installation Guide & Application Programmer's Interface," Technical Report CMU-ITC-071, Information Technology Center, Carnegie Mellon.

[Tho85a] Robert H. Thomas, Harry C. Forsdick, Terrence R. Crowley, Richard W. Schaaf, Raymond S. Tomlinson, and Virginia M. Travers, "Diamond: A Multimedia Message System Build on a Distributed Architecture," *IEEE Computer*, vol. 18, no. 12, pp. 65–78, December 1985.

TCP/UDP Performance as Experienced by User-Level Processes

Josef Matulka

Department of Applied Computer Science
Institute of Information Processing and Information Economics
Vienna University of Economics and Business Administration
Augasse 2-6,
A-1090 Vienna,
Austria
matulka@awiwuw11.earn

ABSTRACT

This paper is concerned with the presentation of empirical data on the performance of the internet protocols TCP and UDP as experienced by a distributed application. After a short summary of the main services provided by the internet protocols IP, UDP, and TCP, an abstract data type built on top of the well-known Berkeley IPC sockets is presented, which has been used to implement a benchmark experiment. The experimental design of this experiment is described and performance data are given for UDP and TCP under two different environments ("loaded" hosts and "unloaded" hosts).

1. Introduction

The era of the dominance of big mainframes with all the computing power concentrated into one computing center is over. The PC revolution has brought local computing power on to each desktop, workstations have found their way into business. At the company level, local area networks (LANs) connecting all computing devices become more and more common while national and international networks enable the communication between the local communities and are themselves brought together by the internetworking approach.

These decentralized hardware configurations demand distributed applications, and to a very large extent computer science research reflects this need. Apart from artificial intelligence distributed and parallel computing are probably dominating more conferences and journals than any other area of computer science. And indeed, there is a lot to discover. The subject of designing and implementing effective parallel and distributed algorithms on real machines is far from being "cut and dried", and the question of how to make the best use of the often impressing amount of MIPS scattered over numerous workstations is all but settled.

One line of research at the Department of Applied Computer Science is concerned with applying statistical and decision theoretical methods to the problems of designing (optimally) distributed algorithms and of adaptive task allocation in distributed systems. If one adopts this framework, basically three components are needed in order to succeed:

1. *a mathematical model*
2. *an objective function* (minimization of run time, etc.)
3. *empirical data* (to quantify the cost of communication)

Other research is concerned with more theoretical issues (cf. [Tau89a]), this paper sets out to deliver some of the empirical data needed for filling in communcation cost by benchmarking a small server-client-type distributed application. Section 2 first provides a short summary of the communication protocols used, namely the internet protocols TCP and UDP†, and the interface used to access them (Berkeley IPC sockets). Secondly, a simple abstract data type is described, which simplifies distributed programming, and is used to implement the benchmark. Section 3 describes the experimental design of the benchmark in

† For a thorough discussion of these protocols see [Com88a].

detail: programs used, messages exchanged, environment conditions, measurements taken and performance indices calculated. Section 4 reports on the results of the experiments, the conclusions of which are listed in section 5.

A similar experiment has been performed by [Cab88a] but in general network benchmarks are far more frequent for lower layers. They allow a "cleaner" experimental design and an easier identification of factors influencing performance. Unfortunately, the figures relevant for distributed applications are those measured at the user-level.

2. Distributed Computing Under HP-UX

The computing facilities of the Department of Applied Computer Science consist of 16 HP 9000/330 and one HP 9000/350 workstations. All of them are running HP-UX 6.2, a System-V.2-type UNIX version with ARPA and Berkeley extensions, and are connected by a 10 megabit/second Ethernet.

Distributed applications in this environment usually make use of the internet protocols TCP and UDP. These two transport-level protocols are often referred to as TCP/IP and UDP/IP respectively, in order to stress that both make use of the underlying (network-level) IP protocol which in turn is built on Ethernet services‡.

The *Internet Protocol (IP)* enables the upper-layer protocols to send Internet datagrams, which may be lost or duplicated, from one host to another which may belong to another physical net. In order to shield the upper layers from the different maximum transfer units (MTUs) imposed by different nets the protocol has built into it the capability of dividing large datagrams into smaller fragments which – in most cases – travel independently from source to sink where they are reassembled.

This connectionless and unreliable host-to-host protocol is only slightly enriched by the *User Datagram Protocol* Adding individual process identification, UDP datagrams are sent from process to process instead of host to host. UDP still remains unreliable and connectionless. TCP, on the contrary, builds up a virtual connection between the communicating processes, includes flow control, and adds reliability using a sliding window technique.

In order to access the services provided by these protocols in a program, an interface is needed. The Berkeley IPC sockets† are probably the most well known interface, suitable for TCP as well as UDP. From a programmer's point of view, they are just a set of additional system calls, allowing him to open a connection with a remote process, send and receive messages, and finally shut down the communication. This communication is defined uniquely by a five-tuple, called *association* = (protocol, local host address, local port, remote host address, remote port). Port refers to an address within a host which is used to distinguish different processes from each other. Frequently, *service names* are assigned to special ports, thus making them more comfortable to handle. Hosts have names besides their addresses, anyhow.

This paper introduces a simple abstract data type (ADT) built on the Berkeley system calls which has been used in the implementation of the benchmark. The main advantages of this ADT are the acceptance of names instead of addresses for identifying hosts as well as ports and the optional availability of exhaustive logging of all events useful for debugging purposes. The key idea of the implementation is to summarize all information identifying an association in a C structure, the so-called **ComBox** structure.

Two versions of this ADT exist, for TCP and UDP respectively; the TCP functions are shown in Figure 1. The Berkeley calls used by it are given in brackets below each function.

With the aid of this ADT a TCP communication between a server and a client‡ is simple: The server calls **TCPcomListen** to indicate that it is ready to accept communication wishes. Then it calls **TCPcomAccept** and waits until a client invokes **TCPcomInit** and the virtual circuit can be established. Now both processes can send (**TCPcomSend** and receive (**TCPcomReceive**) an arbitrary number of bytes until finally the communication is closed (**TCPcomClose**).

---

‡ This describes the actual situation at the department. Being *inter*net protocols, TCP and UDP are used with non-Ethernet technologies as well.

† IPC denotes *I*nter *P*rocess *C*ommunication.

‡ Only this special case is described, as this is the only one needed for the benchmark.

```
int TCPcomInit (comBoxPtr, localService, remoteService, remoteHost)
[socket, bind, connect]
int TCPcomListen (comBoxPtr, localService)
[socket, bind, listen]
int TCPcomAccept (listenComBoxPtr,comBoxPtr)
[accept]
int TCPcomSend (comBoxPtr,message,len)
[send]
int TCPcomReceive (comBoxPtr,message,len)
[recvfrom]
int TCPcomClose (comBoxPtr)
[shutdown, close]
```

Figure 1: *ADT ComBox (TCP version)*

With UDP, the same communciation is more symmetrical. Each process calls **UDPcomInit** to begin communication†, then packets are exchanged by **UDPcomSend** and **UDPcomReceive** and eventually the communication is ended by **UDPcomClose**. Thus, in the UDP version the functions listed in Figure 2 are needed.

```
int UDPcomInit (comBoxPtr, localService, remoteService, remoteHost)(5)
[socket, bind]
int UDPcomSend (comBoxPtr,message,len)
[sendto]
int UDPcomReceive (comBoxPtr,message,len)
[recvfrom]
int UDPcomClose (comBoxPtr)
[shutdown, close]
```

Figure 2: *ADT ComBox (UDP version)*

The use of this ADT in the benchmark definitely adds some overhead, but if the sockets are used in a distributed program, they are likely to be isolated in specific procedures, thus adding very much the same kind of overhead.

3. Experimental Design

3.1. Environment

The benchmark experiments were performed on two of the Department's HP-UX 9000/330 workstations. Two different scenarios were analysed: "unloaded" hosts and "loaded" hosts.

The term "unloaded" host refers to a workstation on which no other user program except the benchmark process is running. There was no attempt made to reduce the "ordinary" system processes running on the UNIX system save that the **crontab** entry which would have invoked the **syncer** in 15 minute intervals was removed for the duration of the experiments.

In the "loaded" hosts scenario each of the two hosts was compiling a C program as a background process. In fact, it was the UDP echo server used for the benchmark, a C program of 30 lines source code, which was compiled and linked again and again.

As the tests were performed during night hours, the net was considered "unloaded", this means relatively free of user traffic.

3.2. Test Programs Used

Four small programs were written for the benchmark, all of them making use of the functions provided by the ADT ComBox. Two of these programs are just very simple echo servers, for TCP and UDP respectively. They are started before the experiments begin and are continually waiting for incoming messages. Each message is received (this entails a copying in main storage) and sent back immediately (meaning a second copying of the message).

† No UDP pseudo-connecitions are used.

The remaining two programs, called clients for short, are concerned with the collection of data by exchanging messages with the above described echo servers.

```
for (j = 0; j < 40; j++) {
    TCPcomInit(&comBox,"bench","bench",rhost);
    elapsedTime[0] = times(&timeVal[0]);
    for (i = 0; i < 100; i++) {
        bytesToRec = len;
        if (TCPcomSend(&comBox,message,len) != len)
            exit(-2);
        do {
            bytesRec = TCPcomReceive(&comBox,
                                &message[len - bytesToRec],
                                bytesToRec);
        } while ((bytesToRec -= bytesRec) > 0);
    }
    elapsedTime[1] = times(&timeVal[1]);
    TCPcomClose(&comBox);
}
```

Figure 3: *TCP Client*

The code given in Figure 3 illustrates how the benchmark works. A TCP connection is opened 40 times and 100 messages of a certain length are exchanged before the connection is closed again. The message length **len** is specified in an outer loop, executing the code given for different message sizes. The reason for the selection of these two repetition counts 40 and 100 as well as the actual message lengths chosen will be given in the next section.

The loop around **TCPcomReceive** is required as TCP uses a *stream concept* without any record boundaries. Several receive calls might thus be necessary to get all the bytes sent at once.

The timing results are obtained by invoking the **times** system call twice: before entering the inner **for-loop** and after leaving it. Thus, what is actually measured is how long it takes for the client to exchange 40 messages with the receiver in terms *user time*, *system time*, and *elapsed time*.

As the loop almost entirely consists of system calls, user time will turn out to be neglectable compared with system time (typically at least fifty times smaller than system time). System time and user time are measured for test purposes only and are not reported in this paper.

The client part of the UDP benchmark program given in Figure 4 is very similar to the TCP version. It uses the UDP version of the ADT ComBox. A loop around **UDPcomReceive** is not needed, as all packets sent at once are bound to be received at once. (Of course, IP fragmentation takes place for message sizes above 1469, but the protocol hands over to the user process only the reassembled packets.)

As UDP contains neither secured transmission nor flow control, it is possible for the packets to be lost due to failure of the underlying layers or due to desynchronisation (when a process sends data before its peer has arranged for reception, the packet is lost). Furthermore, it is possible that **UDPcomSend** fails, this means that UDP is not ready to accept a packet at a that point of time. Therefore a timeout, set via an **alarm** system call, is necessary. If the client does not receive a packet within the timeout period (4 seconds in the experiments) the receive call is interrupted by the alarm signal, the packet simply is retransmitted and this round is cancelled. When the attempt to send fails, the client waits for 4 seconds before trying to send again.

Of course, this timeout mechanism makes a correction of elapsed time necessary. For each send and receive error 4 seconds are subtracted from the time obtained via **times** system call. It is this adjusted elapsed time which is used in the calculation of the performance indices.

The provisions for possible send and receive problems are taken at the client side only. The UDP echo server has neither of them. It just consists of a simple receive-send loop. If it cannot send, it just goes on and waits for the next datagram. If a datagram sent to it is lost, it has not even a chance to take notice of this event. **UDPcomReceive** just waits till the next datagram arrives.

```
for (j = 0; j < 40; j++) {
    UDPcomInit(&comBox,"bench","bench",rhost);
    elapsedTime[0] = times(&timeVal[0]);
    for (i = 0; i < 100; i++) {
        if (UDPcomSend(&comBox,message,len) != len) {
            errno = 0;
            sendError++;
            i--;
            sleep(delay);
            continue;
        }
        alarm(delay);
        if (UDPcomReceive(&comBox, message, len) != len)
            if (errno == EINTR) {
                errno = 0;
                receiveError++;
                i--;
            } else
                exit(-1);
        alarm(0);
    }
    elapsedTime[1] = times(&timeVal[1]);
    UDPcomClose(&comBox);
}
```

Figure 4: *UDP Client*

3.3. Message Length and Repetition Count

Both pieces of code presented in the previous section include two loops. Though we are interested in round trip time for one message, not this value is measured, but the time needed for 100 round trips. This has been done because it is somewhat meaningless to measure elapsed time for just one round trip with the aid of the **times** system call which reports the elapsed time in units of 20 milliseconds in our environment, whereas the values of interest are typically even below this number for smaller message sizes.

The second loop, responsible for another 40 repetitions of these 100 repetitions, has been established in order to estimate a standard deviation for round trip time.

Both loops have been executed for a number of different message lengths. 1 byte is a fairly obvious message length, it is the smallest user message one can possibly transmit. The other extreme is given by 9216 bytes, being the largest UDP message allowed by the specific implementation.

Though the basic endeavour of the paper is not to present a detailed analysis of TCP and UDP, but to provide empirical data useful when designing distributed algorithms, an attempt was made to use knowledge of IP fragmentation for explaining performance. The values chosen between the boundaries 1 and 9216 partly reflect this attempt. The largest number of UDP user bytes that can be transmitted with one link-level packet is 1469. This value is calculated by subtracting the lengths of link-level, IP, and UDP headers (17, 20, and 8 bytes) from the maximum Ethernet packet length used by the implementation (1514 bytes). Therefore, 1469 and 1470 bytes were reasonable choices. In order to get the other values representing fragmentation boundaries one has to consider two facts: First, 1514 bytes as maximum length of link-level packets are somewhat an exception. All other fragments travel inside packets with a maximum length of 1509 bytes. Second, only the first IP fragment has to include a UDP header. With this knowledge, one gets, with calculations such as 1464 + (1464 + 8) = 2936, the message sizes 2936/2937, 4408/4409, 5880/5881, 7352/7353, and 8824/8825. The values in the middle between two fragmentation boundaries namely 2200, 3672, 5144, 6616, and 8088 were included as well.

Added to these sizes were the lengths 112, 256, 512 in order to obtain increased information for smaller messages. Besides, 112 happens to be the smallest message that can be stored in one mbuf (a data structure used for memory management inside the kernel), and 512 the common size for a UNIX disk block. Eventually 1015, 1017, 1020, 1022 and 1024, 1026, 1028, 1030 were included to test whether the singularity uncovered by [Cab88a] for messages of size 1024 can be reproduced.

Though the fragmentation boundaries reported above are valid for UDP only, these message sizes were used for TCP as well, in order to be able to compare between UDP and TCP.

3.4. Performance Indices Calculated

Two performance indices are calculated for the different message sizes used: round trip time and round trip throughput.

In the diagrams of the next section, *round trip time* is given in milliseconds and denotes the time needed for the exchange of one message of a given size. What one actually gets from the experiment, however, are 40 values of elapsed time, each of them representing the time needed for the exchange of 100 messages. By taking minimum, mean, and maximum of these 40 values and deviding them by 100 one gets the values of minimum, mean, and maximum round trip time reported. The variance of these 40 values is devided by 100 as well and used as an estimate for the variance of one round trip.

Round trip throughput is calculated from round trip time by deviding message length by round trip time and multiplying by 8, as throughput will be given in kilobit per second. We are defining round trip troughput, a value of for examples 1000 kilobit/sec means that 1000 kilobit are sent to and fro in one second. Thus, if one wants to compare the figures given with the 10000 kilobit/sec valid for raw bits, one has to double them.

4. The Results of the Benchmark

In Figure 5 four functions are charted, all showing throughput as varying with message size. The top line shows UDP throughput as observed in case of unloaded hosts. The next line shows the corresponding TCP throughput, and the last two lines are the throughput functions for the loaded hosts scenario.

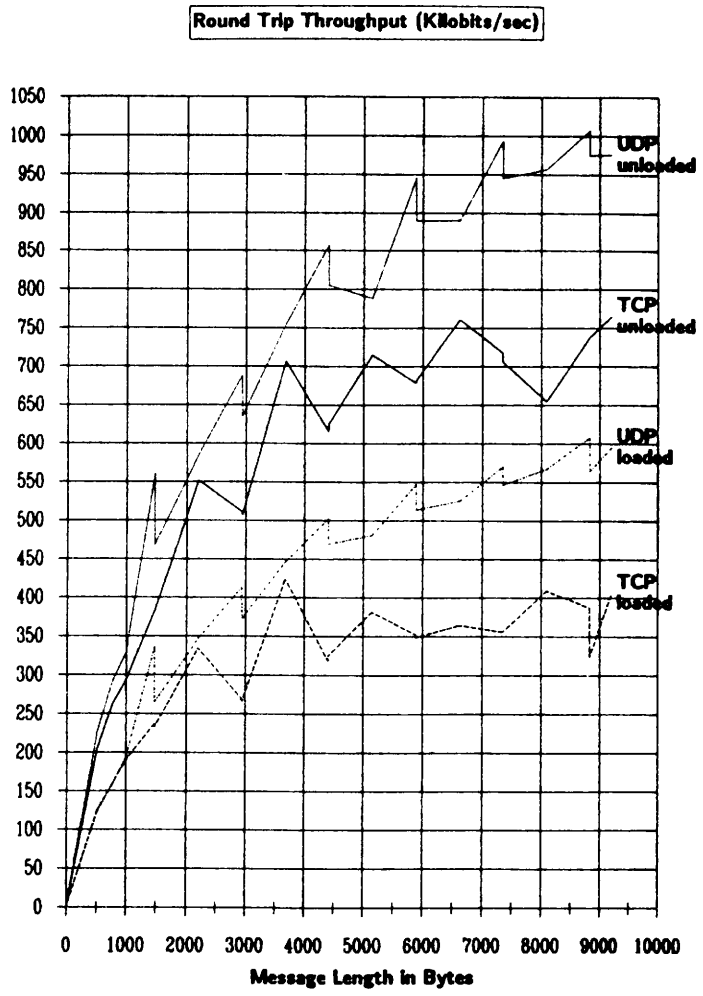


Figure 5

In principle, if one disregards local low and high points, all functions can be regarded as convex functions of message length. In general it therefore holds that the larger the message size the higher the throughput. The highest throughput, namely 1000 kilobit/sec, was reached when exchanging UDP packets of 8824 bytes between two idle hosts.

The major decrease in throughput from 1469 to 1470, from 2936 to 2937, from 4408 to 4409, from 5880 to 5881, from 7352 to 7353, and from 8824 to 8825 obviously is accounted for by IP fragmentation. 5880, for example, needs only four Ethernet frames for transmission, whereas the single additional byte of 5881 requires an additional frame which causes the decrease in throughput.

The measured throughput for TCP (also in Figure 5) shows that the same influence of IP fragmentation cannot be observed when the TCP protocol is used. In most cases, the two values are too close together to be distinguished as different points. This is not surprising, as the message sizes were carefully chosen to reflect the fragmentation points valid for UDP, and different message sizes would be the relevant IP fragmentation boundaries for TCP. Yet, the paper of [Cab88a] as well as own experiments confirm that IP fragmentation is not suitable for explaining TCP performance anyhow. Flow control and reliability entail the exchange of small packets containing control information; compared to this overhead, IP fragmentation is neglectable.

It is this overhead that causes, not surprisingly, a slower functioning of TCP in comparison to UDP. In general, TCP throughput ranges between 70 and 90 percent to that of UDP for message sizes above 2200 bytes, with the exception of 3672 where TCP throughput reaches 94 percent of the corresponding UDP value. The discussion of smaller message sizes is deferred to the discussion of Figure 6. There, the throughput values for the loaded hosts scenario, shown in the last two lines of Figure 5, will be commented on as well.

Before turning to Figure 6, however, some words on two special points: The TCP throughput values for 8088 (unloaded scenario) and for 8825 (loaded scenario) reflect the impossibility to guarantee an idle net even during night hours. As communication with other network users indicate, this effect is most likely to have been generated by a backup daemon. Other runs have shown a much higher troughput for these message sizes, namely 820 instead of 654 kilobit/sec for 8088 (unloaded scenario) and 389 instead of 323 kilobit/sec for 8825 (loaded scenario).

The data provided by Figure 6 are TCP and UDP round trip times for small message sizes only (the smallest message size charted being 1 byte, of course, though the chart might give the impression of it being 0 byte). In the UDP functions, one finds the expected fragmentation gap between message sizes 1469 and 1470. The most eminent impression, however, is the singularity observed between message size 1022 and message size 1024 for TCP as well as UDP. There, a decrease in round trip time from 25 ms to 22 ms has been observed in the UDP case, and an even larger decrease from 28 ms to 23 ms in the TCP case (unloaded scenario). Additional data collected for 1015, 1017, 1020, and 1026, 1028, 1030 support that 1024 is not just an outlier; round trip times for the former are typically close to 25 ms (UDP) and 28 ms (TCP), whereas for the latter they range at about 22 ms (UDP) and 23 ms (TCP).

Interestingly enough, the data published by [Cab88a], show a singularity at the same very point 1024. The explanation given there is [Cab88a, p.45]:

> The throughput "singularity" observed in the TCP/IP curve for the 1024 byte message size is due to savings in copy operations and to the better use of mbufs. As datagrams have well known boundaries, UDP/IP always allocates mbufs in optimal way. In UDP/IP the only savings in data copying are for 1024 byte datagrams, which are sent using trailer protocols.

As this paper does not investigate into the matter of network buffer management, it is beyond its scope to determine whether the same applies to the obervations made here. It can be stated, however, that 1024 bytes are transmitted by TCP in a very efficient way. For that message size, TCP reaches 97% of UDP performance (unloaded scenario) and is even faster in the loaded scenario (103%).

In general, for the small messages charted in Figure 6 the difference between TCP and UDP performance is reduced when there is extra load on the hosts; for message sizes below 1040, in particular, TCP is frequently even faster than UDP. The same, however, is not true for message sizes beyond 2200 for which the gap tends to become broader rather than smaller when extra load is imposed on the hosts.

Up to now, only mean values have been reported. The most eminent differences between loaded and unloaded scenario, however, are to be found in the variances of the performance indices. The majority of estimated standard deviations for round trip time are below 16 ms in the loaded scenario. When there is load on the hosts the deviations are about 5 to 20 times larger. It seems that the standard deviations for TCP are larger than those for UDP in cases of message sizes above 4000.

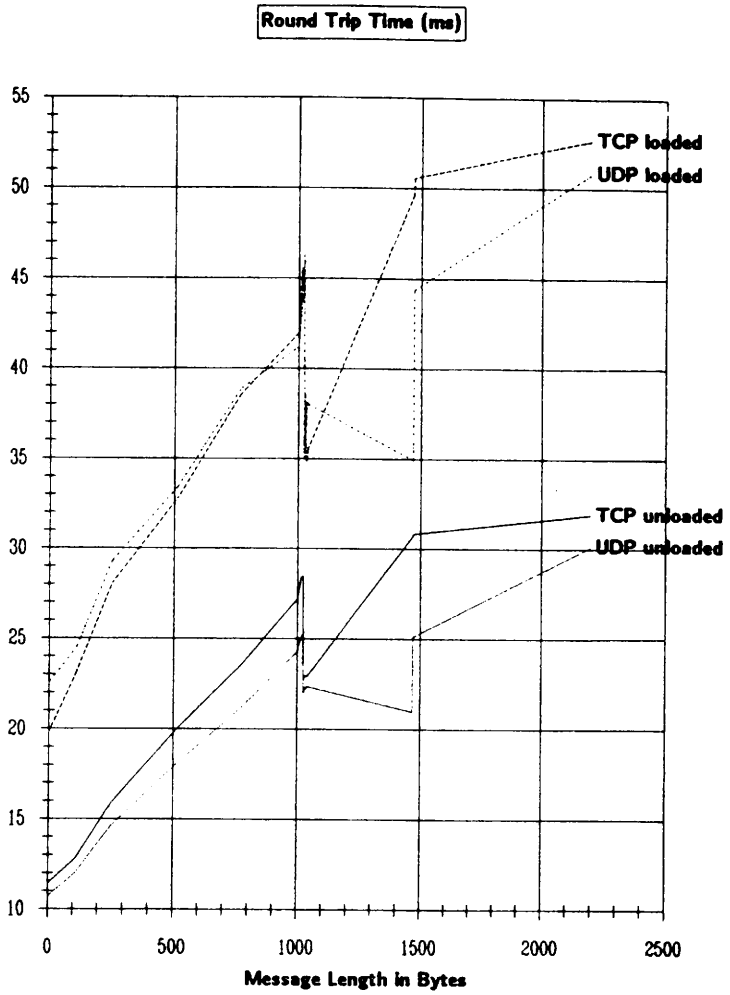


Figure 6

The extraordinarily high standard deviations for the two message lengths 8088 (TCP, unloaded scenario) and 8825 (TCP, loaded scenario), namely 258 ms and 256 ms support the statement already made above: most likely they are outliers.

Figure 7 shows, as an example, minimum, mean and maximum values as well as standard deviations for UDP round trip times in the unloaded scenario. The mean value tends to be very close to the minimum.

5. Conclusions

The paper set out to deliver empirical data useful in the context of distributed programming. Empirical data showing user level performance for TCP and UDP in two different environments (loaded and unloaded hosts) were presented and compared. The following facts and rules are confirmed by the data:

1. *Sending small messages is inefficient.* In general, throughput increases with message size. Try to wait and send larger blocks of data whenever possible. If the nature of the problem inevitably seems to require sending only a few bytes at a time, consider sending some bytes more, even if the probability that they are of any use for the receiver is a very low one. Their transmission does induce hardly any extra cost.
2. *Anomalies might be used for improving performance.* As can be seen from the tables presented, sending 1022 bytes is very different from sending 1024 bytes. In a time-critical application it makes sense to look for such anomalies – which can be detected for example with the aid of the programs presented in this paper‡ – and make use of them.

‡ The programs are available from the author at request.

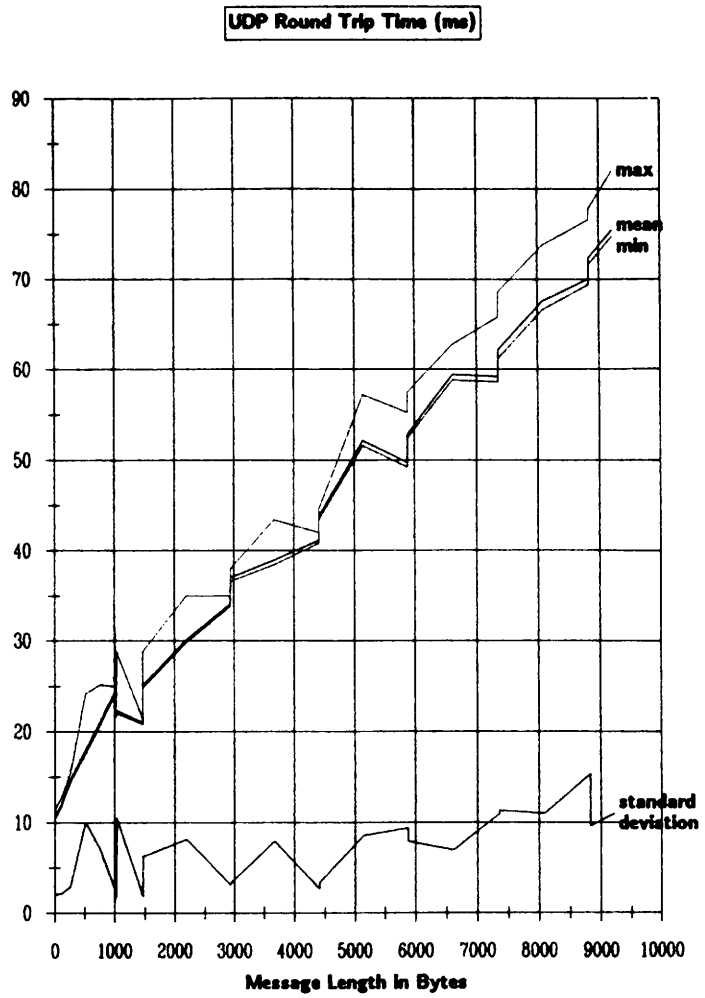


Figure 7

3. *UDP allows faster communication than TCP.* If one considers that the time needed for establishing communication (which is typically much higher for TCP) is not included in the figures given, the gap is even broader. Thus, if there is no need for a session-like connection, for example if only a few messages are sent to the same receiver, UDP should be chosen. It might pay to implement an application-specific protocol upon UDP. This is especially true, when UDP/IP is built on Ethernet technology which is known to provide relatively secure transmission (cf. [Nem88a]).
4. *Busy hosts mean slow communication.* There is an important influence of host load on network throughput. Just one additional user doing C compilations, not too unusual a case in a UNIX environment, might cause a 40 percent decrease in throughput. Thus, in many cases not network capacity but host processing power will be the bottleneck.
5. *Busy hosts mean higher variance of round trip time.* The delivery of single messages might be delayed by even more than 40 percent, as the additional load results in major increases of the standard deviation of round trip time.

Whereas it was possible to explain a good deal of UDP performance variation by IP fragmentation, the same was not possible for TCP. As the paper of [Cab88a] indicates, an analysis of network memory management is probably needed for a detailed explanation of TCP performance. Furthermore, it would be interesting to collect data for the case of loaded network (cf. [Cab88a]) and for hosts running not only one C compilation but many of them. Stastical estimation and test procedures could then be used to estimate and test the influence of different factors on performance.

References

[RFC768a] J. Postel (ed.), *User Datagram Protocol*, 1980. Request for Comment (RFC) 768

[RFC791a] J. Postel (ed.), *Internet Protocol*, 1981. Request for Comment (RFC) 791

[Cab88a] Luis-Felipe Cabrera, et al., "User-Process Communication in Networks of Computers," *IEEE Transactions on Software Engineering*, vol. 14, no. 1, pp. 38–53, January 1988.

[Com88a] Douglas E. Comer, *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, Prentice Hall, Englewood Cliffs, 1988.

[Nem88a] Martin A. W. Nemzow, *Keeping the Link: Ethernet Installation and Management*, p. 222, McGraw-Hill, New York, 1988.

[Tau89a] Alfred Taudes, *Optimal and Adaptive Communication Strategies in Distributed Decision Tree Problems*, To appear in Proceedings of the International Symposium on Approximation, Optimization and Computing, Dalian, China, 1989.

Interconnection of LANs, using ISDN, in a TCP/IP architecture

Philippe Blusseau

OST-DRD/CMC
CNET LAA/ITP/RIC
Route de Trégastel
B.P. 40
22301 LANNION CEDEX
blusseau@cnetlu.uucp

ABSTRACT

Network interconnection is not really a recent problem: network architectures currently used, or planned, specify a level structure, inside which one can insert new wide or local area networks. Of course, ISDN is one of these networks that can be included in those architectures.

In this paper, we will point out which interconnections can be done between LANs and ISDN. We will then focus on the new services offered by the ISDN, especially the basic "S0" access, and how they can be efficiently used in this context.

1. Introduction

Since 1983 France Telecom (The French Telecommunication Administration) has been involved in ISDN implementation, with the RENAN (Réseau des Entreprises pour de Nouvelles Applications Numériques) project. This experiment led to a commercial offering, called NUMERIS, with more than 300 accesses at the end of 1988. The covering of the whole country is planned by 1990.

OST, founded in 1980, well known in data communication field, owing to its packet switchers, PADs, and multiplexers, has set up a communication boards department, whose main products are X25, X21, and ISDN "S0" boards. This department, with the support of the CNET (the French PTT's research center), initialized a study about LAN's interconnection through ISDN.

In the UNIX environment, TCP/IP is currently the standard for local data communication. This network architecture was designed to allow interconnection between several local and wide area networks, the whole then forming a global internetwork.

As far as wide area communications are concerned, the TCP/IP architecture mainly uses leased lines, or X25 networks. Network interconnection is carried out by "IP-routers", which act as gateways between an Ethernet network, and (for example) an X25 network.

The interface currently offered by NUMERIS, is the basic "S0" access [Dic87a], which provides two B-channels (64Kbit/s), and one D-channel (16KBit/s). Therefore, the main interest of ISDN, is to allow faster communications, in an internetwork including WANs. Moreover, ISDN offers "supplementary services", primarily designed in a telephone point of view, that can be very useful in the design of an "ISDN IP Router".

In the first place, we will remind the main TCP/IP features, especially those dealing with network interconnection (§ 2). Afterwards, we will underline the ISDN characteristics likely to be of interest in our context (§ 3). At last, we will describe software and hardware implementation of our router (§ 4). In conclusion, we will mention the possible future developments of this study (§ 5).

2. TCP/IP and network interconnection

2.1. The TCP/IP architecture

TCP/IP [DDN82a] is the common name of a set of protocols, defined within the context of ARPA project. Those protocols, specified in "RFC" (Request For Comments), are parts of a four-levels architecture (Figure 1).

The **A level** represents the real networks used in the communication. As far as Local Area Networks are concerned, TCP/IP mainly uses the Ethernet technology. Wide Area communications take generally either leased lines, or X25 networks.

This level fits with the Physical, Link layers, and the lower part of the Network layer, of the OSI reference model [ISO84a].

The **B level** corresponds to the upper part (sub-level "3c") of the OSI Network layer. This level specifies the Internet Protocol [RFC791a], which have, on the one hand, to standardize the services offered by the real networks, and on the other hand, to manage routing. Those functions will be described on the next chapter.

The **C level** fits with the Transport layer of the OSI reference model. The two main protocols specified at this level are:

- **TCP** (Transmission Control Protocol), which performs end-to-end flow and error controls, and offers a connection-oriented service.
- **UDP** (User Datagram Protocol), which offers a connection-less service.

The **D level**, specifies several protocols, which offer user's services. This level corresponds to the three upper layers of the OSI reference model. The main application protocols are **FTP** (File Transfer Protocol), **Telnet** (Virtual Terminal), **SMTP** (Simple Mail Transfer Protocol), and **TFTP** (Trivial File Transfer Protocol).

NFS (Network File System) is not an "ARPA" protocol. However, this set of protocols was designed to operate on the top of a connection-less transport level. The current implementation of NFS is based on top of the service offered by UDP.

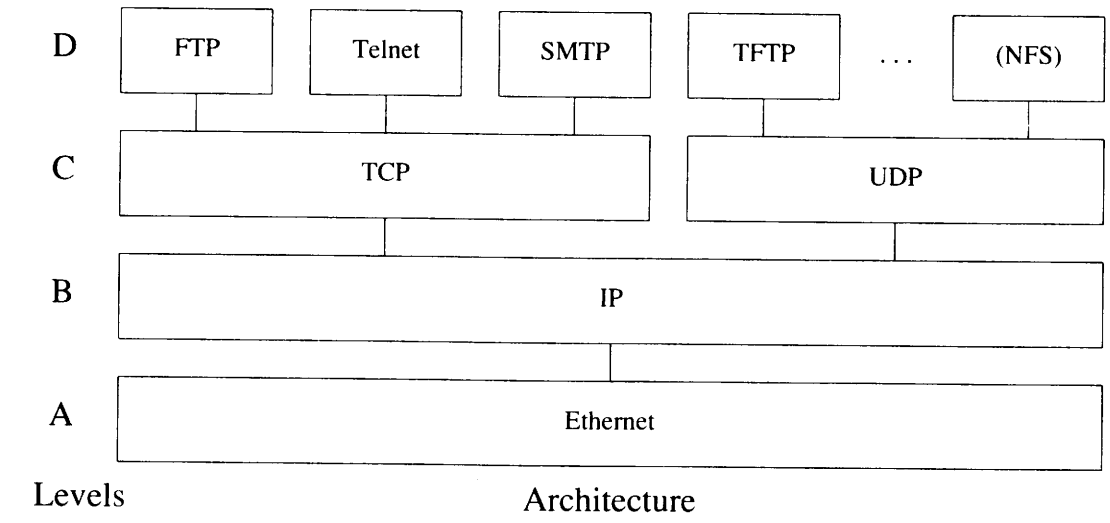


Figure 1: *The TCP/IP architecture*

2.2. The Internet Protocol

As we have already said, the B level manages routing between the real networks. The C and D levels deal with end-to-end communications, and thus are implemented only in end-systems. Intermediate systems, called **IP-routers** in the TCP/IP architecture, may only implement the A and B levels (Figure 2).

The Internet Protocol is connection-less: the C-level entities don't have to open a connection, before exchanging data. Each data message, called "datagram", is supposed to be independent from the others. Therefore, this datagram has a header, which includes the necessary control information for routing functions, and especially source and destination addresses.

Internet Protocol entities, implemented in each (end and intermediate) system, when receiving a datagram, analyse the datagram header. Information in this header helps the entity to find the next real network to use, and the "real address" (in this real network) of the next system to pass the datagram through.

The Internet Protocol also manages fragmentation and reassembling of datagrams. A datagram larger than the maximum size of a real network unit, will be fragmented by the IP entity. Fragments will be reassembled on the destination end system.

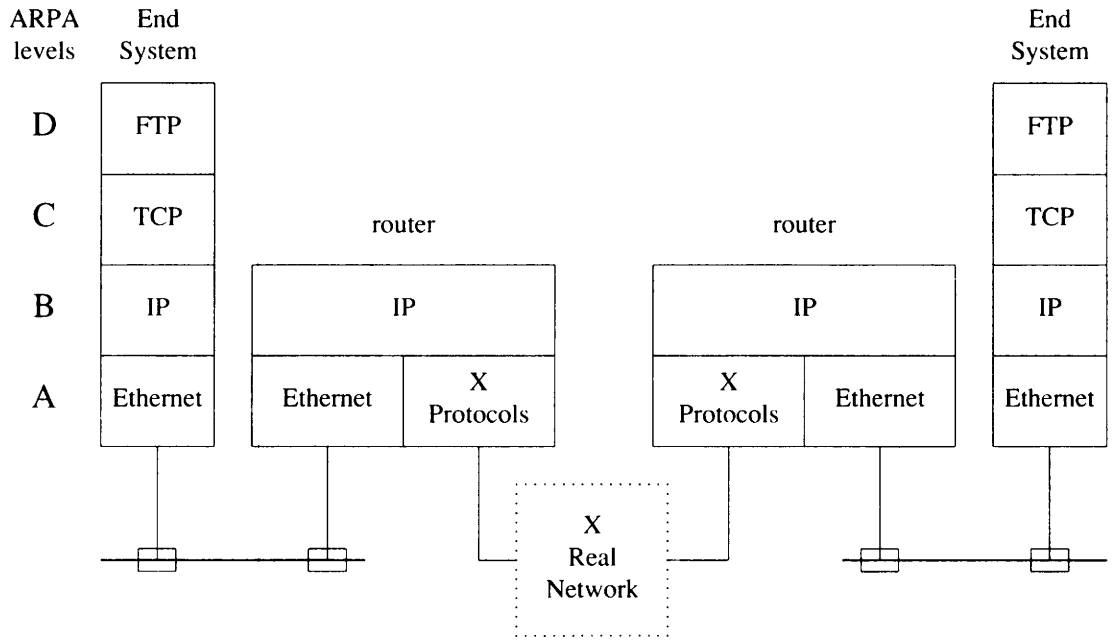


Figure 2: *IP routers*

2.3. Convergence Function

When the real network, chosen by the IP entity, offers a connectionless service, the datagram is merely fitted into a real network unit (for example, an Ethernet frame). The next system will then have to extract the datagram, and to analyse its header, in order to continue the B-level's work.

On the other hand, if the real network offers a connection-oriented service, as in X25, X21, or ISDN networks, one must implements a function, whose aim is to offers the connection-less service to the IP entity, while the service used is connection-oriented. Such a function is generally called a "Convergence Function".

The first convergence function specified in the ARPA project, was "A standard for the Transmission of IP Datagrams Over Public Data Networks" (in accordance with the X25 protocol) [RFC877a]. Some points in this document, give indications dealing with the connection-oriented aspects of X25. Therefore, those points will be naturally suitable to all connection-oriented networks.

When receiving a datagram, the convergence function must verify if there is already some open connection to the next system. If not, the convergence function calls for a connection set-up.

This connection may be cleared, when it has been idle for some period of time. The value of this period may be a compromise between many parameters, among which we can quote:

- the connection set-up and clearing delays;
- the connection set-up cost;
- the connection duration cost.

In order to allow protocol demultiplexing, the convergence function defines a specific value, that can be set in the first byte of the call user data field.

The X25 convergence function also uses X25-specific aspects,such as using the M-bit, in order to manage datagrams whose length is greater than the X25 packet-size, accepted by the real network.

3. Using ISDN

3.1. The ISDN convergence function: basic principles

As far as we are concerned, ISDN offers a connection-oriented service: before any data transmission, one must open a B-channel. Therefore, an "ISDN-Router" (Figure 3) must implement a convergence function, that we will call "S-DCF" ($S_{[0,2]}$ Dependent Convergence Function), described as follows.

The basic principles of our S-DCF are identical to those explained in the previous chapter. A B-channel will be opened, only when necessary, then closed after an idle-period time.

The first difference occurs in this idle-period choice, because of the ISDN-specific parameters. In France, ISDN costs essentially depends on B-channel's open-time duration. The implementation of X25 Convergence Functions, using timers of about five minutes, is not suitable here.

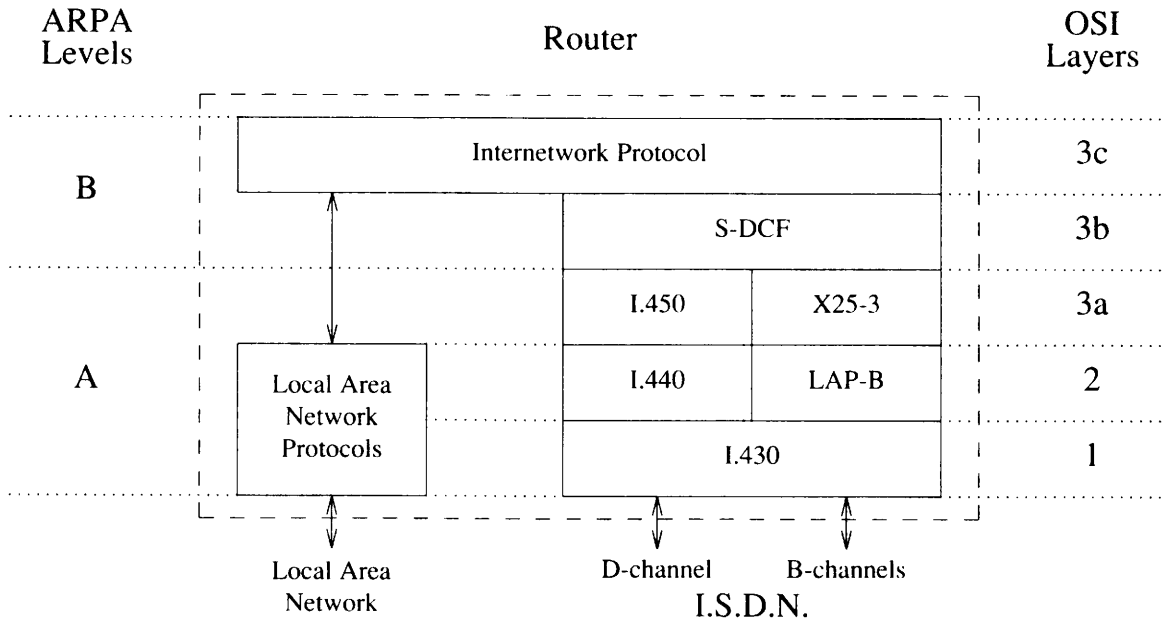


Figure 3: *S0 Router*

The S-DCF also uses more than one B-channel, if others are available, and tries to make a compromise between communication rates, and ISDN costs. When communication needs an ISDN path between two routers, only one B-channel is open first. Other B-channels will then be opened, if it seems that one 64 Kbit/s connection is not sufficient. This decision is taken, according to waiting-datagrams file length.

As we can see in Figure 3, data transmissions use X25 Packet Level Protocol. X25 Packets will be transported in B-channel. But the S-DCF also manages ISDN signaling (D-channel), in order to ask for B-channel set-up, and to use supplementary services described below. Figure 3 shows a "S0-Router", with its D-channel signaling protocols I.430, I.440, I.450 (the S2-Router layers should conform respectively to I.431, I.441, I.451 recommendations).

3.2. Supplementary Services

ISDN, and NUMERIS's implementation of it, offer supplementary services (SS), which can be seen as facilities in current communication services. Many of these SS have been primarily specified in a telephone point of view, but we can see that some of those may allow simplification of the router specifications; other SS may be useful in security or network management areas.

In the basic specifications, we will use the **User-to-User Signaling** Supplementary Service, to allow control data to be exchanged between two systems, before opening a B-channel. These data are defined at the S-DCF layer, and has information giving reasons why the system should accept, or refuse, a communication (priority of the communication, etc ...).

In the communication area, security aspects are more and more important. ISDN supplementary services facilitate the implementation of security at the S-DCF sublayer:

- the **Calling Access Identification** will be required by the router. Only incoming calls, coming from well-known ISDN subscribers (that is to say, declared in an "incoming" table) will be accepted.
- the **Identification of Malicious Call** should be asked, if someone often tries to have non-authorized access to the router.

Network management implies many functions, among which are those dealing with communication-costs control. ISDN offers the following supplementary services, that should be very useful:

- the **Charge Advice** Supplementary Service regularly informs the router, during the communication, of the cost of the latter.
- the **Indication of Total Cost** Supplementary Service gives information about the total cost of a communication, at the end of the latter.
- the **Detailed Invoicing** Supplementary Service may also be used by the router manager.

Some applications require high availability and "fault-tolerant" networks. In this case, one must consider to make use of two (or more) routers, taking a similar action. When some problem occurs on one router, one can ask for:

- the **Call Forwarding** Supplementary Service; The router is considered as a terminal, from an ISDN point of view. Therefore ISDN will re-route future communications to another router.
- The **Portability** Supplementary Service allows the router to suspend a current communication, and to transfer it to another router. This SS may also be used, for the same reasons as the previous supplementary service.

4. Implementation

4.1. The Prototype

The different aspects pointed out in this paper have been implemented in a prototype, shown in Figure 4. This prototype allows interconnection between TCP/IP-based LANs, using basic rate "S0" ISDN access. The TCP/IP architecture has been chosen, because of its availability both in OST company and CNET research center of Lannion.

The prototype is based on an IBM PC or compatible, with one Ethernet board, (3 COM 505), and one ISDN board (OST PCSNET), whose characteristics are discussed further. It was more advisable to choose a multitasking operating system. The operating system is SCO-Xenix 286, because of the availability of both drivers for the two communication boards. In order not to decrease efficiency, there are only two main processes. Those latters may however create other processes for some reason.

The first process deals with user-interface facilities, such as:

- starting and stopping router;
- routing tables view and/or modification;
- access control filters view and/or modification;
- statistics view and/or reset.

The second process manages:

- protocols not handled by PC boards, that is to say the S-DCF and the Internet Protocol;
- Statistics collection;
- Access Control verification.

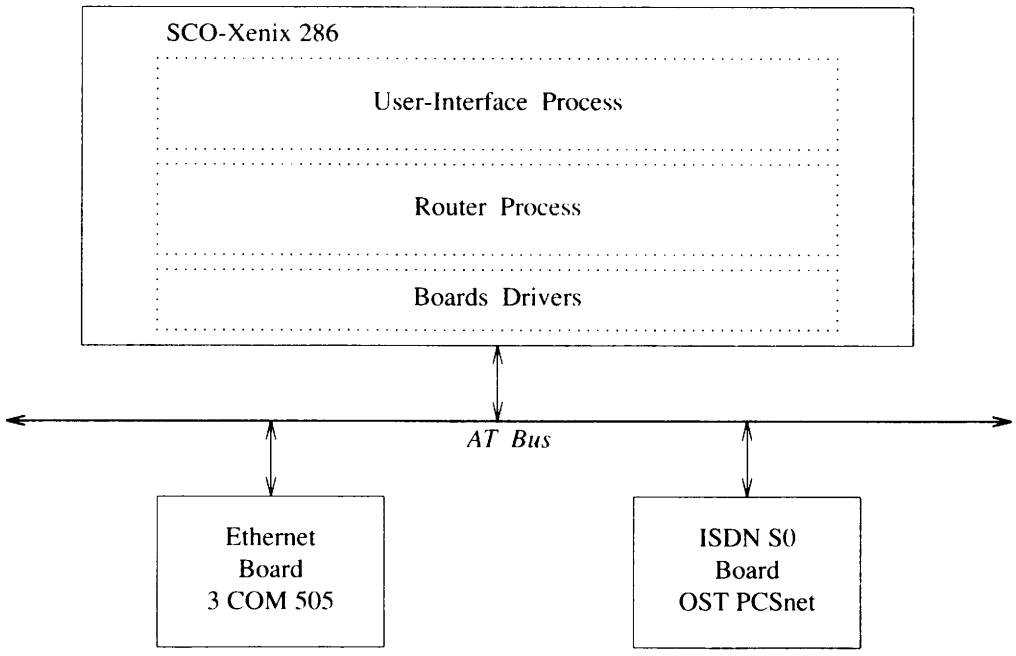


Figure 4: *Router Prototype*

4.2. The OST ISDN PC board

The OST ISDN PC board [Dav88a] conforms to the CCITT recommendations I.430 (layer 1), I.440 (layer 2), and I.450 (layer 3). The D-channel signaling is handled as required for the NUMERIS implementation of those recommendations.

The hardware design is built around a 68000 Motorola µP, with up to 512 KBytes of dynamic RAM. The S0 interface is managed by the AMD 79C30 DSC. Five interface connectors are provided for:

- S0 interface;
- telephone handset (or microphone plus speaker);
- V24 interface;
- external bus access, allowing specific-use peripheral board connection;
- internal PC bus access.

The software is read from the PC hard disk, and manages:

- full management of ISDN signaling;
- full X25 protocol support (level 1 to 3);
- optionally OSI Transport and Session levels;
- "transparent" service, allowing data transmission over one of the two B-channels.

5. Conclusion

With this study, we proved that using ISDN to interconnect LANs was, not only possible, but also interesting from a cost point of view. Now, we have to continue both study and implementation.

The first implementation handles only basic S0-ISDN services. Further studies and developments may include:

- supplementary services management;
- specification, in the OSI "Estelle" language [Bud87a], of the S-DCF, allowing it to be validated with the VEDA tools set, available in CNET-Lannion;
- implementation of an OSI router, as required by MAP/TOP specifications [GMT84a]: those OSI profiles use the ConnectionLess Network Protocol [ISO87a], which is similar to the Internet protocol;
- the European S2 Interface, and its 30 B-channels, seems to be an interesting way to increase global internetwork communications. This consideration requires a more accurate study of the dynamic allocation of those channels.

References

[RFC791a] *Internet Protocol Specification*, DDN Network Information Center, September 1981. RFC 791

[DDN82a] DDN Network Information Center – SRI International, *Internet Protocol Transition Workbook*, Menlo Park, CA 94025, March 1982.

[RFC877a] *A Standard for the Transmission of IP Datagrams Over Public Data Networks*, DDN Network Information Center, September 1983. RFC 877

[ISO84a] ISO/TC97/SC6, *Information Processing Systems – Open Systems Interconnection – Basic Reference Model*, 1984. ISO/IS 7498

[GMT84a] *MAP specifications*, Manufacturing Engineering & Development – General Motors Technical Center `, Warren, MI 48090-9040, September 1984.

[ISO87a] ISO/TC97/SC6, *Information Processing Systems – Data Communications – Protocol for Providing the Connectionless-Mode Network service*, May 1987. ISO/DIS 8473

[Bud87a] S. Budkowski and P. Dembinski, "An Introduction to ESTELLE: a Specification Language for Distributed Systems," *Computer Networks and ISDN Systems*, vol. 14, pp. 3–23, 1987.

[Dav88a] Y. David, "Interface Board for PCs, providing S0 Interface with voice and data communications capabilities," *ICCC '88 Proceedings: Computer Communication Technologies for the 90's*, Elsevier Science Publishers B.V. (North-Holland), 1988.

[Dic87a] G. Dicenet, "Design and Prospects for the ISDN," *Artech House Telecommunication Library*, 1987. ISBN 0-89006-269-2

System Administration of UNIX Networks: Two Approaches to Supporting the Management of Large, Distributed, Multi-Vendor Networks

Georg-Michael Raabe
Carol J. Rosenstock

Apollo Computer GmbH
Hahnstrasse 37–39
D-6000 Frankfurt 71
West Germany
raabe_g@apollo.com

ABSTRACT

This paper discusses two aspects of the support of large, distributed, multi-vendor networks. First we will address user account management and show how a single system can manage a heterogeneous network. Second we will address file system backup and restore and present a system designed to coordinate backup for an entire network.

1. Introduction

The UNIX operating system has undergone many changes. Equally signification have been the changes to the networks incorporating UNIX-based machines. They have changed in size, from small single-user establishments to large networks. They have changed in terms of geography, from centralized networks to dispersed networks. And they have evolved in terms of composition, from predominantly single vendor networks to collections of heterogeneous computers.

To accommodate these new, distributed networks methods were needed to support easy data and resource sharing. While UNIX provided individual users with significant performance improvements over time-sharing systems it also removed some of the benefits, most notably in the area of cooperation and sharing. Distributed file systems like Apollo's DOMAIN and file-sharing protocols like Sun's NFS provided the first step by allowing the easy sharing of data around the network. A significant breakthrough came with the introduction of Apollo's Network Computing System (NCS), which allows for the sharing of compute resources around the network.

With these facilities users can now take full advantage of the resources located around their multi-vendor networks. Users and applications can access data located anywhere on the network and the most appropriate computer resource can be applied to each part of an application. This change in the nature of UNIX networks can result in more powerful and robust applications and products.

But this change to UNIX networks can also result in an unexpected burden on the system administrator. Advances have not been as great in the area of system administration. As a rule, the tools were designed for single workstations. Few recognize or accommodate the distributed, heterogeneous nature of the networks. Some can be massaged to handle distributed networks, but they often break down in large networks and rarely work in multi-vendor networks.

To manage their networks, system administrators are often forced to: operate different programs for each hardware platform, coordinate their own activities for the different platforms, and coordinate their activities with other system administrators managing linked networks. New products and approaches are clearly needed to address system administration functions in these new networks.

Some of the key areas for consideration include: file system backup, data archival, software distribution, user account management, software installation, and software licensing. In this paper we will consider unique approaches to two problems: user account management and file system backup.

A user account management system coordinates login and password information. This task is complicated in large, distributed, multi-vendor networks because it requires a unified naming system to ensure assignment of unique names across network. In the interests of data integrity there needs to be a way to coordinate and possibly limit update privileges. And for ease of use there should be a single system across all platforms. With a single system a user could choose to have the same or different login and password on each platform. System administrators would no longer need to explicitly coordinate and unify their activities.

File system backup enables the backup and restore of files resident on disks around the network, using storage devices located around the network. This task is also complicated by the introduction of heterogeneous machines. For expediency, a single program should be able to backup from or restore to any disk around the network, regardless of the type of computer to which it is attached. In addition, the system should be able to utilize any storage device on the network, again regardless of the associated computer. To minimize administrative effort and training, the backup system should be consistent across all platforms.

Apollo has taken a network perspective to the needs of heterogeneous system administration and has developed new approaches to specific problems. This paper will address two of these new approaches. Passwd Etc, a network-wide user account management system, maintains a single logical database of all login and password information, ensuring uniqueness and consistency of information at all machines. OmniBack, a network-oriented backup and restore system, enables the backup of an entire network with single command, supports the easy restoration of files and directories, and allows the utilization of storage devices located around the network.

2. User Account Control

2.1. Introduction

A major requirement for networked computers is the assignment of unique user identifiers. A secondary requirement is the coordinated management of account data associated with these users and user identifiers. This account data is ultimately used to provide and deny access rights to users and to set up users' environments. User account management systems have been developed to provide a single, consistent representation of user identification information (user names and user ID's).

When networks are small and handled by a single administrator, user account management is a manageable tasks. However, as networks evolve from small work groups into larger, distributed networks these tasks become increasingly cumbersome and difficult to coordinate. Multiple administrators, each responsible for a portion of the user community, are required to share overall control of network-wide user account management. They are forced to either: create artificial naming conventions (and trust that people adhere to them), institute unnecessarily strict communication procedures, or accept conflicts and deal with them as they arise. These options are frequently not acceptable. To further complicate the problem, users can also get involved in user account management. Standard UNIX facilities allow individual manipulation of account files. Therefore, work and coordination done by administrators can still be negated by individual activity.

2.2. Summary of Existing UNIX Account Management Facilities

The standard method for managing user account information on UNIX machines is through the /etc/passwd and /etc/group files. They are independent, private, non-replicated text files that reside on each machine. These files are simple lists that associate user names with passwords (in the case of /etc/passwd) and group names with membership lists (in the case of /etc/group). Both administrators and users are free to make changes to local copies of the files.

To achieve network-wide coordination of the information in these files users must either coordinate their changes or relinquish control to administrators. These administrators must then develop a formal management procedure. In most cases this involves: designating one machine as the administrative center, identifying the /etc files on that machine as the master copies, making all changes to the master copies, and manually overwriting all files around the network with the newly changed master copy. Since each update requires administrative intervention and imposes a burden on the network (by forcing a complete copy to go out to every machine), administrators frequently choose to collect requests for changes and periodically execute one batch of changes. While this minimizes administrative effort and network traffic it also means that user account information can be out of date (which could represent a security infraction).

Sun Microsystems attempted to address user account management with the development of Yellow Pages (YP). YP is a simple data lookup facility that provides replicated data service for data sets (including /etc/passwd and /etc/group). It collects information into 'maps' that run on designated YP servers around the network. These maps are meant to serve as focal points for queries and updates. However, both queries and updates still, in some cases, go through the local /etc files, negating the benefits of centralization. YP also recommends, but does not enforce, the designation of a master map. Without the introduction of additional controls, updates can be made and then copied to different combinations of /etc files and YP maps around the network. When maps are updated they are completely overwritten by new copies. Since this, again, requires administrative effort and adds significant traffic to the network, administrators are likely to save up changes, rendering the information in the maps obsolete.

2.3. High Level Requirements for User Account Management

For a user account management system to be truly effective and administrable it must meet certain requirements:

- Manage a single set of information for networks of any size
- Guarantee assignment of unique user identifiers
- Ensure consistency of information and updates
- Be highly accessible for queries and updates from all machines around the network
- Provide security over information and updates
- Support real time updates
- Enable management of independent administrative domains by different administrators

Only by addressing these requirements can a management system be useful in supporting the administration of user account information.

2.4. Passwd Etc Architecture

Apollo has developed a user account registry system, Passwd Etc, that manages user account data (login, password, home directory, default shell, etc.) as well as general policy information. It utilizes what is logically a single database, encompassing all data for a heterogeneous set of computers. Passwd Etc is built on NCS, which provides the foundation for interoperation between all machines in the multi-vendor network. Passwd Etc consists of the registry database and two sets of routines, server software and client software.

2.4.1. Database Structure

While the registry is conceptually one logical database, it is actually implemented as replicated databases located around network. This improves performance, reliability, and response time. Passwd Etc uses a master/slave model, with the requirement that one database server be designated as the master. While queries can be handled by any server, updates are automatically routed to the master. All slave servers are automatically amended as part of the update process. This protects against inconsistent changes and security infractions through delayed updates and ensures that all servers contain accurate, up-to-date information. Passwd Etc utilizes a weakly consistent replication scheme. To minimize the information being transmitted over the network it propagates only the incremental changes to the database.

The database consists of three types of data: naming information for people, groups, and organizations; login information for people; and general system properties and policies. Groups and organizations are collections of people. Groups maintain the conventional UNIX semantics, and provide the means for a set of people to share privileges to system objects Organizations provide another means for sharing privileges, and would typically be used to divide the user community into administrative groups. The property and policy information sets guidelines for system usage (e.g. minimum password length, account lifespans).

2.4.2. Database Contents

The database consists of two types of entities, PGOitems and accounts. A PGOitem (for person-group-organization) relates to the naming information and establishes a binding between a name and a set of credentials (consisting of a unique identifier and a UNIX id). Each PGOitem has a full name field, an owner field, and properties. Group and organization PGOitems also have associated membership lists. An account provides login information for a particular user (including login name, password, home directory, default login shell, creator of account, account expiration date, etc.). Property and policy information can be associated with PGOitems and accounts.

The use of owner fields in PGOitems and registry policies allows the partitioning of the database into separate administrative divisions. Only the person designated as owner can manipulate that entity, e.g. create accounts for people, alter group and organization membership lists, modify properties, etc. By the assignment of ownership rights, the database can be logically partitioned and securely administered by mutually suspicious system administrators.

2.4.3. Database Security

Update and server administration operations must pass through an authentication interface. The process involves a series of authentication challenges that require the use of multiple secret keys in the encryption and decryption of bit patterns. This mechanism ensures that the database is not altered by an unauthorized user.

2.4.4. Registry Server Software

The server software consists of a database editing tool, a server administration tool, and the server itself that, together, support the maintenance of the registry database. The editing tool is an interactive editor that is used to manage the naming, account, and policy information in the database. It provides users and administrators with a structured interface to the data. The editor is aware of the semantic constraints of the database entries and the registry policies in effect. It uses this information to ensure that the changes are consistent, syntactically and semantically correct.

The server administration tool allows an administrator to control and monitor server activities. These include: changing which server is acting as the master, stopping servers, starting replica reinitializations, listing server sites, and checking the state of any or all servers.

The servers handle three types of activities: database operations (query and update), management of server sites, and database update propagation. The database is kept in virtual memory as a forest of balanced binary trees, resulting in efficient query and update operations. Updates are first applied to the in-memory data structures and are then recorded in a stable storage log. Periodic checkpoints are made of the in-memory data structures. In the event of a system crash the system automatically recovers the database by reloading the last checkpoint state and reexecuting each operation recorded in the stable storage log.

The master registry server manages the initialization of new slave replicas; the tracking of added, deleted, and moved replicas; and the propagation of updates to the replicas. When a slave site first starts running it locates the master site through the NCS Global Location Broker (which maintains information about NCS services available around the network) and announces its existence. For new sites, the master initializes the slave and informs all other slaves of its existence. For existing sites with new locations, the master records the change of address and again alerts the other slaves. When a server site receives a decommission request it purges its database and terminates execution.

During update propagation, the master applies a monotonically increasing time stamp to each update it records. In transmission of an update, the master sends the incremental change along with the current and the previous update time stamps. In this way slaves can detect when they are out of date and will then request to be reinitialized. Database reinitialization is accomplished through a series of propagations of the database to the target slave replica. Reinitializations are also performed on slaves that return to operation after having been out of communication for too long.

2.4.5. Registry Client Software

The client software is composed of a subroutine library of remote procedure calls that enable remote access to the database. This software runs on each machine in the network. The first time a registry operation is performed on any machine the client software contacts the Global Location Broker, which provides a list of the registry servers on the network. The client selects a database server for that operation and directs all subsequent operations to the same server. This alleviates the need to access the location broker more than once. If that server becomes unavailable the client references the list already received from the Global Location Broker and chooses another server.

2.4.6. Local Controls over Operations

A local registry, resident on each machine, provides a cache of user account data. It is used in the event that all database servers are unavailable. This mechanism can also be used to manage a Yellow Pages installation.

Individual machines can be configured to override information coming from the central registry. This can be used to provide stricter access control to a particular machine (e.g. exclude people, groups, or organizations from getting access) or tailor account attributes for a user of a different machine (e.g. provide a local home directory).

3. Network Wide Backup

3.1. Introduction

Every computer installation should execute frequent backups to guarantee the availability and reliability of one of its most valuable assets – its electronic data.

The UNIX operating system contains numerous backup and restore commands that enable a user or administrator to write copies of files to and retrieve files from some storage medium. Unfortunately, this has traditionally been a time-consuming and costly task. The standard UNIX tools (cpio, dd, dump, and tar) are relatively primitive, and require work to tailor them to the needs of the local site.

It is the responsibility of the system administrator to decide which of the commands best addresses the needs of the site, taking into account the number of users, amount of data to be backed up, storage devices available, and frequency and type of restore requests. The administrator will usually write shell scripts around the backup commands to automate backup and to coordinate backup for multiple machines.

As networks get larger and more distributed, backup becomes more difficult. First, if backup is to be automated for many machines across the network, scripts must know about and reflect all the many file systems partitions. Second, this causes the backup scripts to become longer and more complex, to address the needs of all users across several machines. Finally, most installation have a limited number of mass storage devices suited for backups. Unfortunately, they are not always attached to the type of machine being backed up. As networks become heterogeneous new problems are introduced, especially when system administrators are obligated to learn and operate different backup systems.

Due to these difficulties some organizations do not perform backups at all or else do not run them frequently enough, thereby taking significant risks with their data. These problems are further compounded by difficulties in retrieving data from storage media. UNIX tools tend to be even less robust on the restore side, adding to the administrative burden.

3.2. A Summary of UNIX Backup Tools

Many books and articles [Kolstada] have been written on how to manage backups on UNIX systems. We therefore will not add another lesson about UNIX backups. We do think, however, that a quick comparison of UNIX tools available for backups should be made here.

The table below lists some of the advantages and disadvantages of UNIX commands available for backup purposes:

tar

+ simple syntax

- dangerous simplicity (e.g. tar x)
- \+ recursive descent through file system
- does not allow incremental backup
- can not handle multiple tapes

dump/restore

- \+ up to nine levels of full/incremental backups
- may become obsolete/unsupported
- doesn't recognize end of tape

cpio

- \+ most versatile
- knows only about files – not directories
- System V only

dd

- \+ copies whole file system partitions (including sysboot)
- \+ allows data conversion
- knows only about files or devices
- restores only whole file system partitions

3.3. Higher Level Requirements for Backup

We will specify requirements for a comprehensive backup system where backups and restores are performed by the system administrator. We differentiate administrative backups from user backups, data archiving and software distribution, because of different needs, different key players, different time schedules, etc.

We define backups as being time driven, generally nonselective and performed primarily for disaster recovery. This is very distinct from data archiving, which is event driven and may be used to recover space from the primary storage media. In a heterogeneous environment data archiving might also require data conversion to a canonical form, for later restoration to a foreign system.

A backup system should provide the following features:

- A method which allows a user to specify (describe) exactly which objects to backup. It should allow for distinguishing:
  - file system partitions on individual machines
  - directory trees in particular file systems
  - files in particular directories.
- A scheduling system which contains a sufficiently fine level of granularity for scheduling (based on days, weeks, hours, etc.). Also the ability to define different schedules for different sets of machines is critical.
- A complementary facility that supports the three key stages of restore:
  - an easy method of defining objects (i.e. files, directories and/or entire file systems) to be restored
  - means to identify the backup media where such objects are kept
  - efficient restore mechanism.
- A journaling facility to automatically track all results of backup procedures:
  - generate and maintain journals of status of backup sessions
  - generate and maintain journals of backup media contents.

- A robust mechanism for detecting and correcting for various types of errors:
  - check labels of backup media, to prevent administrator from using the wrong media
  - automatically reschedule filesystems that were not available for backup
  - recover from error situations during a backup session
  - retry locked files.
- Support for backup storage devices:
  - support for a variety of devices
  - ability to easily add support for new devices.
- Support for unattended backup:
  - start backup procedure automatically
  - report to system administrator on the success of the backup process.
- Ability to work in a heterogeneous network of workstations and other computers

Note that one should not require that the machines which are networked together share a common file system via NFS or, for example, Apollo's DOMAIN. Neither should it be assumed that these machines use the same methods for naming objects.

3.4. OmniBack Architecture

Apollo's product for file system backup and restore, OmniBack, was designed to address the above requirements. OmniBack is made up of three cooperating programs (see Figure 1): the Operator Interface, the Disk Agent, and the Media Agent. The Operator Interface acts on the part of the user to control the backup and restore procedures across the network. The Disk Agent manages the disk during backup and restore, and the Media Agent manages the backup storage devices. These three components can simultaneously execute on the same or different machines on the network. With this modular architecture, the work performed by OmniBack can be distributed across multiple machines on the network for increased performance. The scheduling and journaling facilities are other important components of the OmniBack architecture. The scheduling facility eases the administrative job of controlling backup scheduling in a network. The journaling facility provides extensive feedback to the user and system administrator on the current and previous backup runs. Both of these features contribute to OmniBack's ease of use.

3.4.1. Scheduling Facility

OmniBack uses a simple ASCII text file called the work-list (an example of which is shown in Figure 2) to determine which volumes to back up and when. The system administrator is responsible for setting up the work-list, which is written once and is easily maintained thereafter. Once a user or administrator initiates a backup session OmniBack automatically references the work-list to determine the files and directories due for backup.

In its simplest form, the work-list need only specify the names of the volumes to be backed up. If nothing more is specified, the system defaults dictate that full backups be performed once a week, incremental backups be performed every day, and the entire volume be backed up. The administrator may choose to specify alternate backup schedules for one or more volumes listed in the work-list. This can be done by specifying days of the week or a relative frequency (e.g. every 3 days). The work-list can also be tailored in terms of the contents of the backup. Files and directories can be included in backup (with the -trees option) or excluded (with the -exclude option). Thus, the work-list has the flexibility to be used in either a simple default way, or it can be tailored to the needs of a particular computing facility.

In some sites, it is the individual machine owner, rather than the administrator, who determines what should be backed up or what should be excluded from backup. OmniBack supports this by providing a simple way for the administrator to selectively delegate these decisions to machine owners. Using work-list capabilities, the lists specifying what is included or excluded from backup can be redirected to other files (using the '<' symbol). These files can then be written by the user to control what he/she wants backed up. Only the file name of the redirection files need to be shared between the user and administrator.

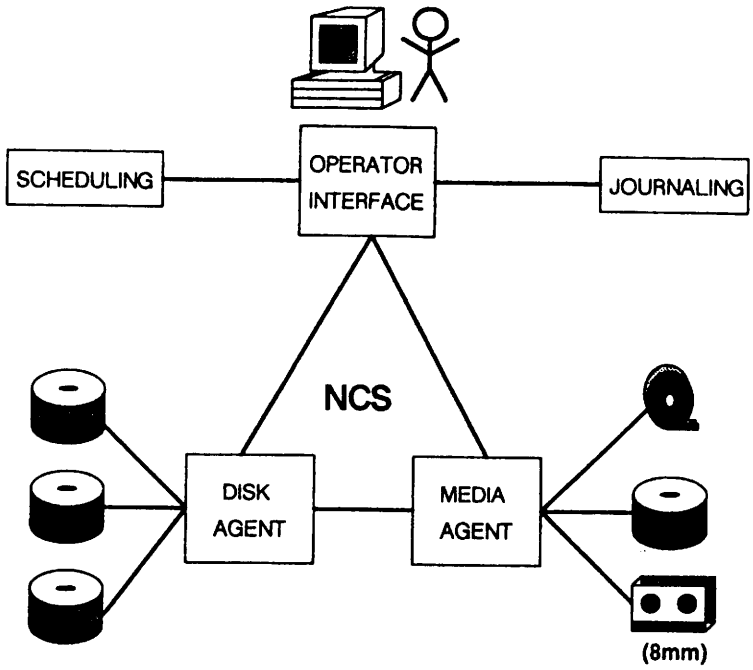


Figure 1: *OmniBack Architecture*

```
default   -full_sched  monday
          -incr_sched  wednesday friday

volume    //maple     { -trees //maple/user }

volume    //elm       { -trees //elm/demo
                         -exclude //elm/demo/*.bak }

default   -full_sched  friday
          -incr_sched  every 4 hours

volume    //ash       { -trees < //admin_node/tree_list
                         -exclude < //ash/user/exclude_list }
          -incr_sched  tuesday thursday
```

Figure 2: *OmniBack Work-List Example*

The administrator maintaining the primary work-list has ultimate control, but can share control with users as desired.

As each volume is successfully backed up, OmniBack records the date and time in a 'dates' file that it maintains. The next time OmniBack runs, it uses information from this file and the work-list to determine which volumes are currently due for backup, and whether a full or incremental backup is needed on each. Exactly those backups are then performed.

3.4.2. Journaling Facility

The OmniBack journaling facility offers administrators and users detailed information about backup runs. Log files are used to track the backup media and their contents. They serve as a mechanism for checking on the result of backup sessions and for identifying specific media required for file restores.

The 'detail log' documents, in detail, the backup information for a particular volume. One log is produced for each volume backed up in each OmniBack session. The level of detail is controlled by the administrator. The 'volume history log' contains cumulative historical information about all backups of a particular volume. These logs list the date and type of each backup performed on that volume. A 'summary log' is produced for each backup session, providing an overview of all aspects of the session. An 'error log' is generated if any errors occur with the storage device or in backing up an individual volume.

3.4.3. Operator Interface

The Operator Interface manages the overall session, acting as an interface between the user and the other OmniBack components during backup or restore. It controls backup and restore operations, initiates the Media Agent and Disk Agent programs, records information in log files, and monitors backup and restore progress. These programs run on the machine from which the backup or restore command is issued.

Once a work-list has been created for a network, execution of the OmniBack backup operation is initiated from the shell command line. Command line options can be specified to control: schedule adjustments, alternate log and work-list files, output listings, etc. OmniBack can be run with or without the window-based user interface.

Likewise the restore options allow the user to specify: the volume being restored, specific directories or files to be restored, alternate restore destinations, etc. If omitted from the command line, restore prompts for the required information. Other options allow the user to monitor ongoing progress as files are restored.

3.4.4. Disk Agent

The Disk Agent manages the reading and writing of files from and to disk. The Disk Agent program usually runs on each machine being backed up or restored, for efficiency and reliability. However, data can be pulled from or pushed to the disk from another machine. Typically there will be multiple Data Agents running simultaneously during an OmniBack backup session for improved performance.

The Disk Agent design was driven by performance and operation constraints of large network computing environments. It was designed to operate through the file system rather than the raw disk. Because of this, no system shutdown is required during backup.

3.4.5. Media Agent

The Media Agent manages the backup storage media, be it magnetic tape, disk, or 8 mm cassette tape. The Media Agent program runs on the node where the storage device is resident. The Media Agent does not look at the actual data sent by the Disk Agent. No conversion or interpretation is performed, so this data is private to the Disk Agent. The Media Agent only prepends a header and adds some bookkeeping information for its own purposes.

3.5. Performance

The most logical way to maximize the performance of the network backup operation is to exploit the parallel computing resources of the network and the individual machines. In most backup products running on networks, performance is limited by the transfer time in getting the data off the disk and over the network to the machine controlling the storage media, since these are sequential operations. In these systems, the utilization of the tape drive is a very low percentage of the backup time. Thus, the tape is idle for most of the backup procedure.

By exploiting the parallelism inherent in a network, OmniBack can run multiple Disk Agents simultaneously. The Media Agent program interleaves data coming from the multiple Disk Agents, and can thus keep the media device running a much greater percentage of time during the backup procedure.

Since the OmniBack user interface can execute on a separate system from the Disk Agents and Media Agent, setup and control operations are done in parallel with reading data from the disks and writing data to the backup media. In this way, data writing is running more regularly to maximize performance. OmniBack has been designed to also exploit the concurrency available on multiprocessing systems; all three components run multiple processes in parallel on their host machines. By capitalizing on the multiprocessing and parallelism of individual machines and the network, OmniBack enables disk access, network transfers, and media I/O to run in parallel, thereby pushing the performance of the entire system.

3.6. Heterogeneous Operation

OmniBack was designed to accommodate heterogeneous operation. Built on NCS, OmniBack has the mechanism at the lowest level to operate in multi-vendor networks. NCS, a de facto industry standard, allows processes on one machine to execute procedures on other machines.

The OmniBack framework supports the scheduling, initiation and management of backups for a heterogeneous set of machines. In such an environment, some operating systems may have a different way of naming files and directories than is used in the UNIX operating system. They may even have some file system concepts that do not have an equivalent in UNIX. Because of this, OmniBack makes no assumptions about the format of object path names, wildcard conventions, or even what options are required. Specification of backup parameters can be different for various heterogeneous systems without affecting the base OmniBack implementation. This approach allows OmniBack to easily operate in a heterogeneous network of systems, with a minimal amount of additional development work.

3.7. Reliability

System backups represent a large investment in computer, financial, and human resources. A backup system must be persistent in the face of network errors, since networks are prone to temporary failures. NCS provides the framework for detecting errors during communication between systems. If noise on the network disrupts communication, for example, OmniBack detects it and retries the disrupted operation.

If a machine is not available at the time a backup request is made by the user interface, an error message is printed on the screen and entered in the appropriate logs. This backup request will then be automatically rerun during the next backup session, without a specific request from the operator.

If a system goes down during the backup operation, OmniBack will detect it. It will attempt to regain communication with the failed system for a reasonable period of time. If the system responds within the time period, the backup operation is continued. If not, error messages are entered into the log files and OmniBack continues with the next system due for backup. The failed backup request will then be automatically rerun during the next backup session.

If a particular file is locked during backup, OmniBack will retry the backup operation at the end of the Disk Agent session. If it still unavailable OmniBack will flag this is in a log file, providing the name of the file not backed up.

4. Conclusion

Passwd Etc provides a powerful facility for addressing the issues involved in user account management. It is the first product specifically designed to accommodate large, distributed, heterogeneous networks. It provides a single, scalable system for consolidating and managing user account information. Passwd Etc ensures the assignment and use of unique user names and passwords across all platforms, guarantees the accuracy and consistency of this information at all sites, and provides security for updates and changes. In addition, it allows the management to be centralized or decentralized amongst administrators with complete control over their user communities. Passwd Etc offers a robust and easy way to manage user account information across heterogeneous platforms.

OmniBack is the first truly network-based backup system. Its modular architecture allows it to take full advantage of the computing features inherent in a network of machines. It goes beyond the standard backup and restore functionality by offering sophisticated scheduling and journaling capabilities. It enables an administrator to coordinate backup and restore operations for an entire network. It allows flexibility in the location of components, taking advantage of the full compute power of the network. It offers a robust user interface geared towards minimizing the administrative nightmare a network of systems often presents, yet simple enough for novice users. OmniBack provides a complete backup management system for computer networks.

References

[Kolstada] Rob Kolstad, "Daemons and Dragons: Backing up files," *UNIX Review*, vol. 7, no. 3, 4.

Resource Management System for UNIX Networks

D. Schenk, G. Reichelt, A. Pikhard

UNISYS Austria
Mariahilferstrasse 20
A-1070 Vienna
Austria

ABSTRACT

Although there are comfortable tools for UNIX administration on single, i.e. non-networked systems, there is a lack of such tools for (large) UNIX networks.

File sharing utilities such as RFS or NFS offer file-sharing and, in the case of NFS, automated updating of distributed data. A tool for central resource administration is missing.

UNISYS RMS (Resource Management System) allows central maintenance of distributed resources. A resource is any part of any component of the network that may be (re)configured, e.g. a user-id, a terminal port or a software product.

The "heart" of RMS is a relational data base which runs on one specific node in the network called the RMS Server. The database application generates requests which are forwarded to the other systems in the network called the RMS Clients. Every update to the data base causes a chain of actions to be started on the clients.

A prototype of RMS is currently evaluated; it runs both on LAN (Ethernet) and WAN (X.25). The full RMS is currently under development at UNISYS Vienna.

1. Introduction

In the beginning of UNIX there were no tools for system operation. UNIX was "the system designed for programmers by programmers", so there was no use for an operator.

In the meantime, UNIX has become a platform for a lot of end-user applications. UNIX systems no longer are programmer's systems only but are used anywhere.

This situation made operating tools necessary. Such tools offer easy ways for backup, restore, adding or deleting user id's, printers, etc.

Examples of such tools are *sysadm* or *sa* (which is an extension by UNISYS). While these tools are convenient for single installations, they have (at least) two severe restrictions:

(a) The only *documentation* on the state of the system is the system itself, so you have to look up in the system tables, e.g. /etc/inittab or /etc/passwd if you know where you have to look for what.

(b) They fail completely in UNIX *networks* as they are not designed for that purpose.

Decreasing hardware costs have, on the other hand, lead to distributed solutions. Networked systems are used more frequently instead of larger single systems for cost and environmental reasons.

A printer, for example, may be used by any user independent of the system the user works on, although the device is physically attached to one specific computer.

The Yellow Pages mechanism of NFS is a first step towards maintenance of distributed UNIX systems. But, as discussed under (a) above, there is no explicit documentation on the state of the network and this mechanism can only handle automated file updates. So the Yellow Pages can only handle items which are completely bound to file updates where the configuration files are identical on all systems in the network.

This may be sufficient for adding or deleting user id's or groups, but it is not sufficient for peripheral devices like terminals or printers, as they are described in a different way on different systems.

Therefore, RMS has been designed. RMS offers

(a) full documentation on the state of the network and also on the state history

(b) dynamic distributed resource management which allows orthogonal assignment of resources to systems.

This paper describes the principles of RMS operation together with the existing prototype. It will not discuss implementation details of RMS.

2. Definitions

2.1. Resource

Any software part on the network, i.e. of any system in the network, that may be (re)configured.

In the current prototype, resources are

- user and group entries
- user profiles
- terminal definitions
- printer definitions
- software products (add on to the operating system)

2.2. RMS Server

One dedicated system in the network that keeps the global state (see below) and which triggers changes to this state initiated by some network superuser. The RMS database is residing on the RMS Server. Currently, there may be only one Server in the network.

2.3. RMS Client

Any other system in the network. The client's configuration may be changed **only** by the RMS Server.

2.4. RMS Source Node

A system (client or server) that keeps a base of software products which may be installed on any other client. Source Node does no mean that the software is stored in source code. The file structure of the software base is defined by the installation process. The software base cares for different processor architectures in the network. There may be several source nodes in the net.

2.5. RMS Subserver

A system which accepts network changes from the server and forwards them to several clients. This allows a tree-like network management structure. Subservers are not implemented in the prototype.

2.6. Request

A request is a set of environment variables with associated values which are forwarded from the server to the client an which define an action.

2.7. Action

The actual change of a local system configuration on a client. The action together with its parameters is defined in the request.

2.8. RMS Trigger

A program running on the RMS Server which is invoked by the DBMS on changes to the global system configuration and which initiated actions on the clients by building and sending a request.

2.9. RMS Monitor

A program which runs on each client and which initiates local system changes on receipt of a request by starting an action.

2.10. Global system state/change

The state/any change to the state of the entire network, i.e. of any client, from the view of the RMS Server.

2.11. Local system state/change

The state/any change to the state of one client from the view of the client's local operating system.

3. RMS Structure

3.1. The RMS database

The heart of RMS is a relational database which stores the global state of the network. Network operation is accessed via a user oriented database application.

It is important to mention that the network administrator has not to know much about UNIX internals. The DB application is screen oriented. There exist entry/update screens for every resource type (e.g. user, terminal, printer, etc.) and also for inter-resource relations.

Appendix A shows some of the entry/update screens of the RMS database application.

The RMS database contains three types of data:

(a) tables describing the resources

(b) tables describing inter-resource relations

(c) auxiliary tables

Tables describing resources are for example the user table or the printer table. They contain information which is related only to the resource described.

Tables describing inter-resource relations contain information about combination of resources, e.g. a user's access to a certain software product (both the user and the software product are resources) or a host emulation (software product) via a terminal line (peripheral resource).

Auxiliary tables contain standards or defaults such as printer model names, default stty parameters, .profile entries, etc.

The RMS database is used in three ways:

(1) The global state of the network can be retrieved easily from the tables; several report facilities offer a fine system documentation.

(2) The history of all changes is also logged as every change to the global state must be done via the database.

(3) All changes to the global network state are actually performed by the DBMS on correct entry of all items.

Any change must be given an activation timestamp. This timestamp defines when a change is due to become active. Imagine installation of a new release of some software component. For operational reasons this changes is scheduled for Sept. 12, 1989, 12:00 p.m. This is the activation timestamp.

If the current date and time indicate that a change is due to become active, a request is generated and the required actions are invoked.

3.2. RMS Operation

3.2.1. RMS Trigger

As soon as the user (the network administrator) has entered all data needed for a change, RMS runs without user interaction.

The database application generates a request that consists of a set of data.

- The first item of the set contains a list of all clients affected by the change.
- The second item contains some update sequence to the database. At the time the request is initiated, this is only a template. It will be completed and then applied to the database on termination of all actions associated with the request.
- The remaining items contain parameters for the action associated with the request and data for RMS trigger.

For all clients affected by the change, RMS trigger build the request data set and sends it to the client.

As the request may contain a lot of data (e.g. in the case of installation of a software product), it is planned in a future version of RMS to send bulk request to RMS Subservers which forward copies of the request to several clients.

3.2.2. RMS Monitor

As soon as the request has arrived at the client, RMS trigger activates RMS Monitor on the client (only in the case of correct delivery of the request).

RMS Monitor interprets the request and starts an action. The action itself together with its parameters are stored in the request.

The RMS Client Package, this is the part of RMS which must be installed on every client, contains a set of programs which apply actions th the real UNIX system (e.g., for adding a user, update /etc/passwd, make $HOME, chown ... $HOME, create .profile, etc.).

These programs return an exitstatus which is passed back to the server via RMS Monitor and RMS Trigger. Depending on this state, the change is marked as

- OK, if it could be applied successfully to all clients
- FT (failed totally) if it could not be applied to any client due to possible network problems
- FL (failed) if it could not be applied to some clients (maybe because the were not up at the time of the attempt). In this case, a simple recovery is tried for those clients on which the action failed.

4. Sample RMS actions

For the following examples, assume a network consisting of four systems:

(a) The RMS Server, named "s".

(b) Three clients named "c1", "c2", and "c3". The structure of the network is sketched in Figure 1.

4.1. Adding a user

A user 'user1' has to be added to client c2. Adding a user is an operation which has affect only to one client. Therefore, RMS Trigger invokes the action "add user 'user1'" only on client c2. There is no action on s, c1, and c3 (note that the RMS Server may also be client at the same time).

4.2. Adding a printer

A printer named 'lp2' which is physically attached to tty100 on c3 is to be generated. Note that all physical (hardware) changes have to be done *before* the (software) change is done via RMS.

Contrarily to adding a user, definition of the printer shall lead to a (virtual) device which is accessible on all clients. Therefore, RMS Trigger send requests to all clients (lp -dlp2 shall work on all systems).

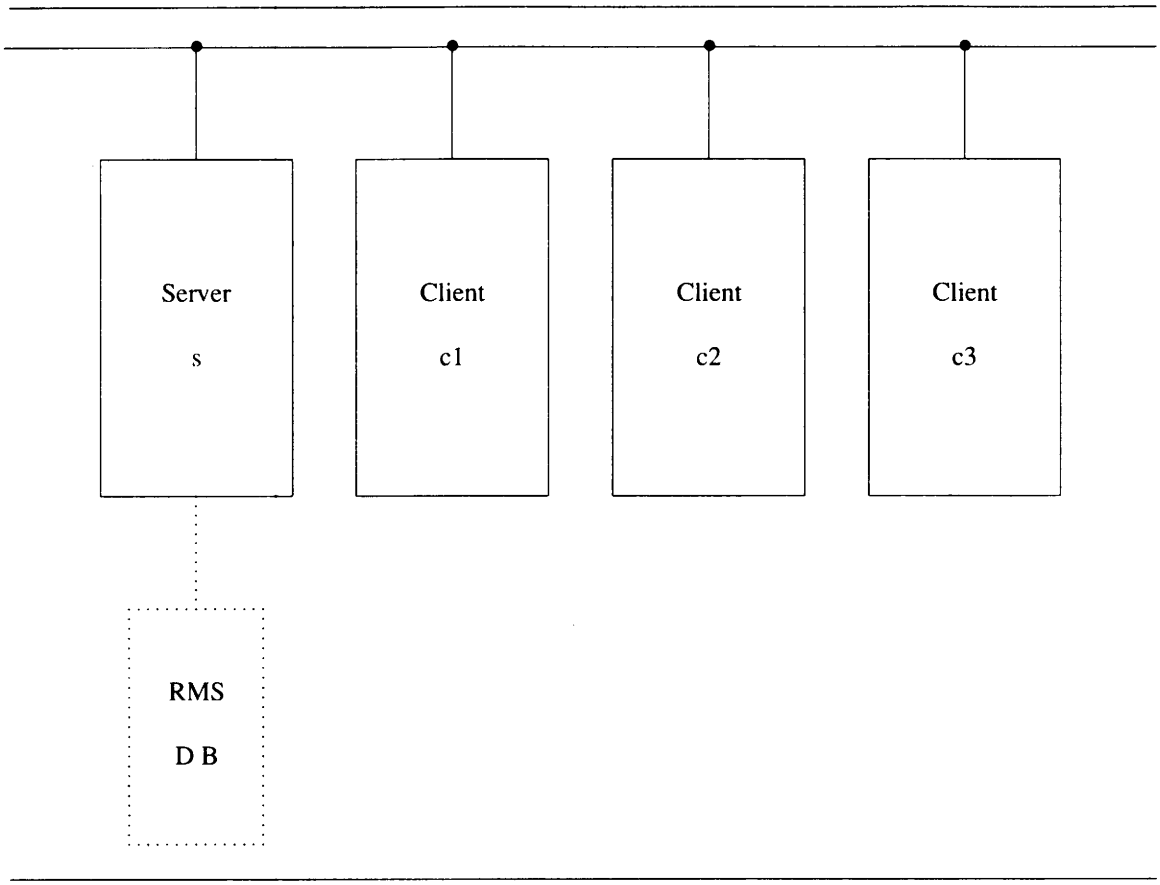


Figure 1: *Sample RMS network*

The action "add printer" invoked by RMS Monitor (on the client) runs in two different ways depending on the actual nodename of the client:

- If the client's nodename matches the destination nodename in the request, it performs "add local printer". The local printer is added to the system via standard lp tools using the parameters defined in the request.
- If the client's nodename does not match the destination, the client performs "add remote printer"; this installs a printer which does not use a physical device but uses the printer on a remote system by network mechanisms.

4.3. Installation of a software product

Usually, software add ons are installed by tape using a special tape format and an easy-to-use utility for installation. The tape does not only contain the pure data but also some programs for auto-installations, checksums, etc.

RMS installation uses a similar mechanism, but instead of real tapes is uses "virtual tapes", i.e. a bytestream which is transferred over the net.

Figure 2 shows how installation of a software product works; it shows the function of server, client and sourcenode.

Put Product is an action which runs on the sourcenode (which is a client) and which is invoked by a client. The data volume transferred via the Virtual Tape may be large so the use of subservers should bring an improvement.

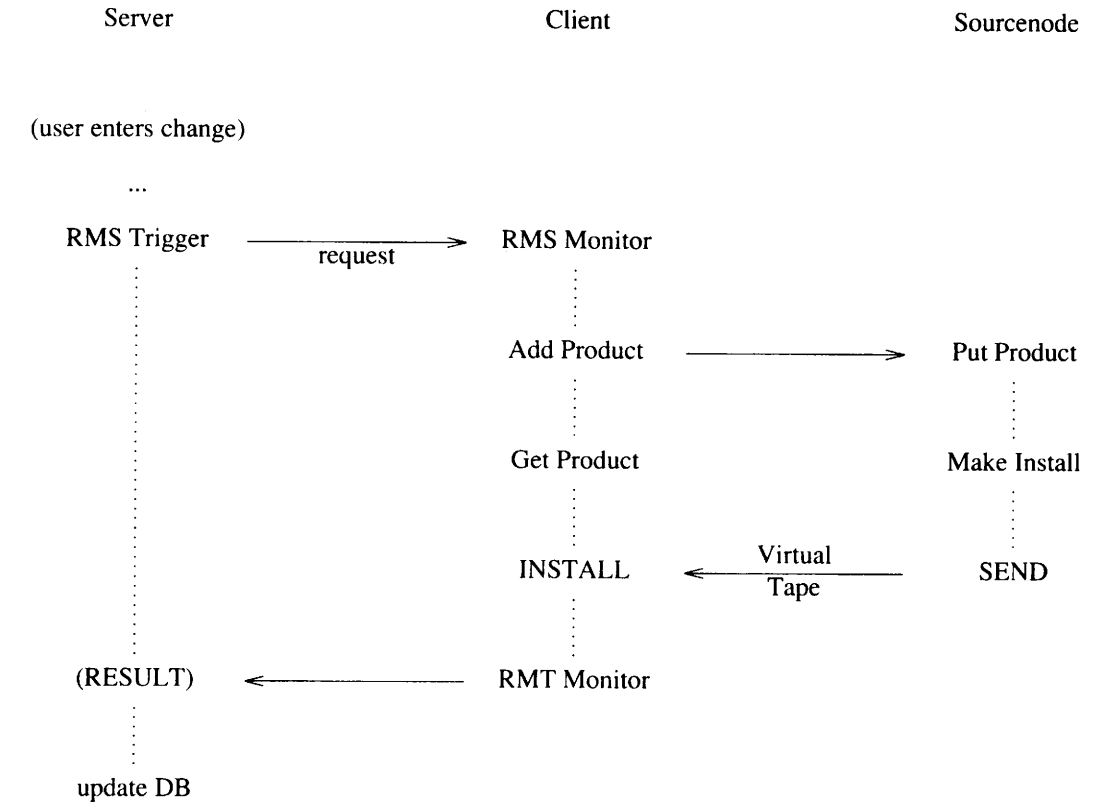


Figure 2: *Software product installation*

5. RMS implementation aspects

5.1. What you need from UNIX

All RMS programs are, as far as possible, written in Bourne Shell. Therefore, RMS is highly portable and runs on different processor types without change.

C programs are used only for performance and copy protection reasons, but they make only 5% of the entire RMS code.

5.2. What you need from the network

Currently, RMS uses the "r"-commands of BSD for data interchange and invocation of processes. Further releases should tend to uucp for several reasons:

- Better recovery
- Store and forward
- More flexibility in network architecture

5.3. What you need further

RMS requires a relational database management system. In the current version, this is ORACLE(TM). The user interface is an ORACLE application (SQL*FORMS) that starts requests by the trigger function of ORACLE.

6. Conclusion

In UNIX networks administration requires documentation on the state of the system. Current administrative tools, if they can handle networks at all, do not contain documentation support.

As a network with all aspects of local information is a complex structure, a relational database has been chosen for description of the state of the network.

The database offers the possibility of automatic invocation of change procedures to the systems of the network. Therefore RMS combines documentation with actual distributed operation.

A client-server model has been chosen to concentrate responsibility in the network. Although there are distributed database systems, this may have not only technical but also administrative advantages (security).

This paper described the design and some implementation aspects of RMS. It showed how RMS works, using examples of common administrative tasks. RMS is currently under evaluation in a prototype version. The restrictions of this prototype were also discussed in the paper.

Appendix A – Sample RMS entry/update screens

This appendix shows sample entry/update screens of the RMS database application. These screens are filled by the user and then committed. Upon commitment, the change request will be stored and activated when the activation time is reached.

As RMS is designed in German in the current version, input prompts are also in German.

Please note the bottom part of the user/printer/terminal entry screens; it contains activation timestamp and return status (updated automatically by RMS and not by the user).

RMS main menu

```
ttyp01 RMS                    H A U P T M E N U                    11/07/89

                Anfangsbuchstabe oder NXTFLD / NXTREC
                   Berechtigungen verwalten
                   Systeme
                   Drucker
                   Terminals
                   Benutzer
                   Produkterklärung
                   Produkte
                   Produkt / System
                   Produkt / System / Benutzer
                   Installation der geplanten Änderungen
                   Übersicht
                   Hilfstabellenwartung
                   Ausdruck der Konfiguration

             Wählen Sie mit <NXTBLK> den Menüpunkt!

        Einfügen:  NEIN     Bild 1                    Anzahl *0
```

Terminal entry/update screen

```
================================================================================
ttyp01 RMS                          Terminals                        11/07/89_

Nodename  __________  (_____________________________________________________ )

    Name  ____________

Parameter für INITTAB
Runlevel   Status          Terminal        Gettydefs
_____      ____________    ____________    ____________
           Gettydefs-Typ   + Login
           ____________    ________________

Parameter für OFIS
Terminaltyp  ____________   Deskset J/N  N
OFIS Text    ____________________________________________________________

Kommentar    ____________________________________________________________

Aktion / am  _ / _________        durchgeführt am  _________  Status  __
angelegt am  _____ _____          letzte Anderung am  _________  von  ____________

--------------------------------------------------------------------------------
         Einfügen:  NEIN     Bild 1                      Anzahl *0
================================================================================
```

User entry/update screen

```
================================================================================
ttyp01 RMS                          Benutzer                         11/07/89_

Nodename  __________  (_____________________________________________________ )

Username  ____________   Nummer  _____

Usergrp.  ____ (_____ _____)
Text      ____________________________________
Home      ____________________________________
Shell     ____________________________________

Profil    _____
Abteilung ____________________________________
Telefon   ____________ ___________

Kommentar ____________________________________________________________

Aktion / am  _ / _________        durchgeführt am  _________  Status  __
angelegt am  _____ _____          letzte Anderung am  _________  von  ____________

--------------------------------------------------------------------------------
         Einfügen:  NEIN     Bild 1                      Anzahl *0
================================================================================
```

Printer entry/update screen

```
================================================================================
ttyp01 RMS                          Drucker                          11/07/89_

Nodename  __________  (_____________________________________________________ )
    Name  ___________          Model  ___________
 Feature  ___________
TTY-Port  ___________           Type  _____
    Stty  ____________________________________________________________________
OFIS: Nr  __   Model  _____________   Text  ____________________________________
Kommentar ____________________________________________________

Aktion / am  _ / _________      durchgeführt am  _________  Status  __
angelegt am  _________       letzte Änderung am  _________  von  ___________

Systeme für Eintragung des Default Druckers: NXTBLK, ENTQRY, EXEQRY
Node        Type  Location                                        Def.Printer
__________  ____  ____________________________________________    ___________
__________  ____  ____________________________________________    ___________
__________  ____  ____________________________________________    ___________
__________  ____  ____________________________________________    ___________
__________  ____  ____________________________________________    ___________
__________  ____  ____________________________________________    ___________
__________  ____  ____________________________________________    ___________
--------------------------------------------------------------------------------
        Einfügen:  NEIN     Bild 1                    Anzahl *0
================================================================================
```

Report menu

```
================================================================================
ttyp01 RMS              Ausdruck der Konfiguration              11/07/89_

                1. Systemliste

                2. Benutzerliste

                3. Druckerliste

                4. Terminalliste

                5. Produktliste

                6. Produkt/Systemliste

                7. Produkt/System/Benutzerliste

                8. Alle Listen 1 - 7

                Ihre Auswahl:  _


--------------------------------------------------------------------------------
        Einfügen:  NEIN     Bild 1                    Anzahl *0
================================================================================
```

Porting Applications to the XVIEW Toolkit and the OPEN LOOK Graphical User Interface

Nannette Simpson

Sun Microsystems
2550 Garcia Avenue
Mountain View
CA 94043
nannette@sun.com

ABSTRACT

The OPEN LOOK Graphical User Interface Functional Specification has evolved over the past two years with the final draft, Revision 18, becoming available for review May 1989. Sun has implemented several prototype toolkits which captured the progress of the user interface design through it's infancy and adolescence. With the impending maturity of the design, Sun offers product toolkits and a suite of applications built on those toolkits. This paper details the issues of transforming existing tools that run on SunView (Sun's kernel-based window system), into integrated applications running on XView, Sun's first OPEN LOOK, X toolkit.

1. Introduction

The XView toolkit is a server-based, object-oriented user interface toolkit designed for Version 11 of the X Window System and X11/NeWS. Based upon the feature set of Sun's kernel-based, SunView toolkit, XView has been redesigned and extended to take advantage of the capabilities of a networked window system, while maintaining much of the Application Programmer's Interface of its parent. After years of being seen as a powerful operating system for developers, but demanding for users, UNIX has a new user interface, the OPEN LOOK Graphical User Interface (GUI), developed jointly by Sun and AT&T. The OPEN LOOK GUI provides a consistent, easy-to-use front-end for UNIX applications. Developers with existing UNIX software can retain the core of their packages while adding a new layer of the OPEN LOOK user interface. Developers with existing SunView applications can port with relative ease to XView. To make the system truly usable, Sun provides a suite of personal productivity tools (many of which have been ported from SunView, some of which are new).

2. XView Architecture

XView can be viewed from two perspectives: the system architecture and the application architecture [Jac89a]. In general, the application writer need not be cognizant of the system architecture when developing tools. However, it is worthwhile to have a working knowledge of the system architecture when making fundamental design decisions to avoid shipping data across the server connection repeatedly.

2.1. XView System Architecture

The system architecture of XView differs significantly from that of SunView. SunView is implemented for SunWindows, Sun's original window manager. SunWindows, like most kernel-based window systems, is hardware and operating system specific. XView is implemented for the X11 server which addresses the problem of utilizing networks populated by heterogeneous machines, operating systems, display resolutions and colour capabilities. Figure 1 shows the structure of an XView application running on a network.

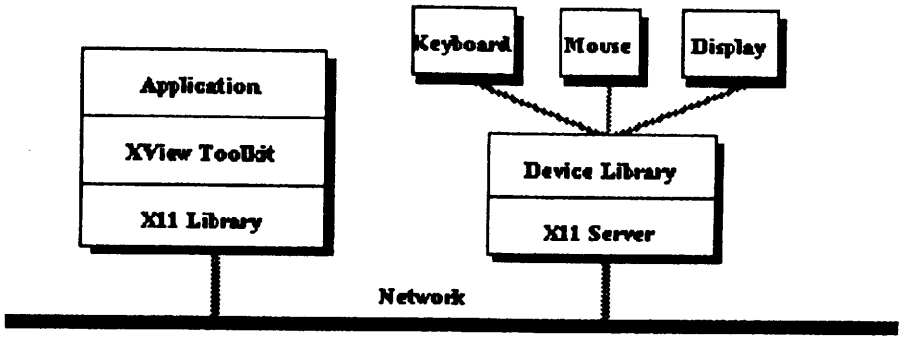


Figure 1: *XView System Structure*

2.2. XView Application Architecture

The application architecture of XView is quite similar to that of SunView. The programmer specifies objects (such as windows, buttons, menus) to be created using variable-length attribute-value lists and call-back procedures, which the toolkit calls to notify the application of events or user actions. The Application Programming Interface (API) diverges somewhat from that of SunView to reflect XView's client/server model and enhancements made to implement the OPEN LOOK GUI. Many of the features from SunWindows are now available in the toolkit. The details of changes to the API are documented in the *XView Reference Manual: Converting SunView to XView* The following discussion provides a glimpse into four toolkit changes. This list is not exhaustive by any means, but is representative of areas which may entail significant effort for the porting programmer.

- generic creation of objects
- canvases
- fonts
- icons, cursors, glyphs

2.3. Generic Creation of Objects

The use of clean, attribute-value lists has been extended to *all* packages. Calls to create objects take the form:

```
XView_object object;
object = (XView_object) xv_create(xview_owner, xview_object_class,
      xview_attribute, xview_value, ...  0);
```

SunView

```
/* Vanilla creation of objects in SunView */
Frame frame;  Panel panel;
    Panel_item button;
frame = (Frame) window_create(NULL, FRAME,
    0);
panel = (Panel) window_create(frame, PANEL,
    WIN_X, 0,
    WIN_Y, 0,
    WIN_HEIGHT, 400,
    WIN_WIDTH, 400,
    0);
button = (Panel_item) panel_create_item(panel,
    PANEL_BUTTON,
    PANEL_ITEM_X, ATTR_ROW(1),
    PANEL_ITEM_Y, ATTR_COL(1),
    0);
window_main_loop(frame);
```

XView

```
/* Vanilla creation of objects in XView */
Frame frame;  Panel panel;
    Panel_item button;
frame = (Frame) xv_create(XV_NULL, FRAME,
    0);
panel = (Panel) xv_create(frame, PANEL,
    XV_X, 0,
    XV_Y, 0,
    XV_HEIGHT, 400,
    XV_WIDTH, 400,
    0);
button = (Panel_item) xv_create(panel,
    PANEL_BUTTON,
    XV_X, xv_row(panel, 1),
    XV_Y, xv_col(panel, 1),
    0);
xv_main_loop(frame);
```

2.4. Canvases

The model of the generic drawing surface, the canvas, is modified in XView. The drawing surface is actually a separate window, the paint window, which is clipped to another window, the view window, so that only the part of the paint window "on top" of the view window shows through. This allows applications to draw on areas larger than the visible window. Figure 2 shows the XView canvas model.

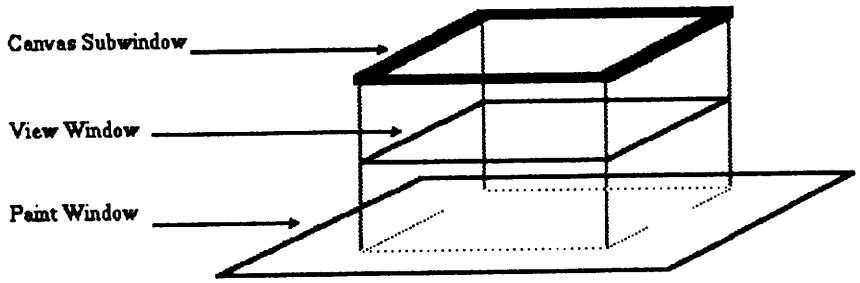


Figure 2: *XView Canvas Model*

The Pixwin library has been rewritten in XView with X graphics primitives. For compatibility, a window's pixwin can be requested and used as before, but the pixwin returned is actually the window handle, strictly an opaque object whose fields cannot be accessed. Interposing is now done on the canvas paint window rather than on the canvas itself.

According to the X11 Specification, an application must be prepared to repaint itself at any time. The attribute, CANVAS_RETAINED which asks the server to maintain a copy of the bits on the screen from which it can automatically repaint, is only a *hint* to the server. The server returns a backing store when asked, but the canvas will *not* be retained when the server runs low on memory.

For practical reasons, XView exposes some of the Xlib drawing routines to the programmer. An efficient method of drawing on a canvas is to set the new attribute, CANVAS_X_PAINT_WINDOW, and provide an X repaint callback procedure. A complete programming example using Xlib directly is found in Appendix A.

3. Fonts

A handful of fixed-width fonts of limited point size is provided in SunView. Programmers who desire a more sophisticated look or a large point size must create a font of their own with a font editing tool. The SunView Pixfont routines, which are part of the Pixrect library are replaced entirely with the XView Font package. Sun's Folio font technology enables XView to offer a wealth of new variable-width fonts which are created like all other XView objects:

SunView

```
Pixfont *font;
font = (Pixfont *) pf_open("/../../screen.r.7");
```

XView

```
Xv_font font;
font = (Xv_font) xv_create(XV_NULL, FONT,
      FONT_FAMILY, FONT_FAMILY_SCREEN,
      FONT_SIZE, 7,
      0);
```

It is possible to query the font about its dimensions in order to calculate text placement:

SunView

```
int fontx, fonty;
struct pr_size  fontsize;
fontsize = pf_textwidth(1, font, "m");

fontx    = fontsize.x;

fonty    = fontsize.y;
```

XView

```
int fontx, fonty, fontsize;
fontsize = (int) xv_get(font,
      FONT_SIZE);
fontx    = (int) xv_get(font,
      FONT_DEFAULT_CHAR_WIDTH);
fonty    = (int) xv_get(font,
      FONT_DEFAULT_CHAR_HEIGHT);
```

The FONT_DEFAULT_CHAR_HEIGHT attribute actually yields "taller" than expected dimensions

because this attribute includes the added height for international character sets. If exact measurements are desired, the font must be queried on a per character basis using the FONT_CHAR_HEIGHT attribute.

It is no longer possible to obtain a font's pixrect, change it, and expect an altered character to display on the screen. Fonts are owned by the server and are not available for dynamic manipulation.

3.1. Icons, Cursors, Glyphs

Imaging is now done by the X11 window server rather than through the Pixrect library. The server does the drawing for XView and any client programs. In order to take advantage of the data format conversion provided in the X11 protocol, XView provides server image objects. A server image is an X11 Pixmap which is represented in the client as a pixrect to maintain SunView compatibility. Icons, cursors, and glyphs are created as server images. The font package represents fonts as server images, also. As with fonts, it is no longer possible to manipulate the internal data structures of these objects to obtain special effects. Instead, calls are made to the particular package with the appropriate object handle and attribute list. The following code initializes an odd-sized† icon for a calendar program:

SunView

```
struct icon sched_icon;
sched_icon.ic_width = 128;
sched_icon.ic_height = 64;
sched_icon.ic_gfxrect.r_width = 128;
sched_icon.ic_gfxrect.r_height = 64;
sched_icon_mpr = mem_create(sched_icon.ic_width,
      schedicon.ic_height, 1);
```

XView

```
Server_image sched_image;
Icon sched_icon;
sched_image = (Server_image) xv_create(
      XV_NULL, SERVER_IMAGE,
      XV_HEIGHT, 64,
      XV_WIDTH, 128,
      SERVER_IMAGE_DEPTH, 1,
      0);
sched_icon = (Icon) xv_create(XV_NULL, ICON,
      XV_HEIGHT, 64,
      XV_WIDTH, 128,
      ICON_IMAGE, sched_image,
      0);
```

4. OPEN LOOK Productivity Tools

4.1. File Manager

The *File Manager* is a graphical front-end to the Unix file system. Similar to Apple's *Finder*, Xerox's *ViewPoint*, and NeXT's *Browser*, the *File Manager* provides direct and intuitive ways to manage files as alternatives to *grep*, *cp*, *mv*, and *ls*.

The original SunView desktop is process-oriented where icons represent running *programs*. Alternatively, Apple, Xerox, and recently Sun offer "direct-manipulation" systems based on having the user interact with pictures on the screen as if they actually are the objects they represent. The focus is on the *data*, typically a file, being manipulated rather than on the engine used to drive the data. The OPEN LOOK GUI defines a set of features and capabilities designed to support a wide range of applications. However it is not a complete desktop metaphor in the MacIntosh sense. The UNIX shell is very much alive and well in the XView Workspace.

4.1.1. Application Developer's View of the Workspace

The "drag and drop" method is an alternative to using the Copy and Paste function keys. Objects are selected and dragged to their destinations (which may be the Workspace or other tools). The application which owns the selection sends an event, ACTION_DRAG_LOAD, to the window under the cursor. The application which receives this event takes the primary selection and executes client-specific actions. For example, a text editor may load a file, a print tool may send a file to be printed, or a calendar tool may schedule an appointment.

† Icons are normally 64×64 pixels.

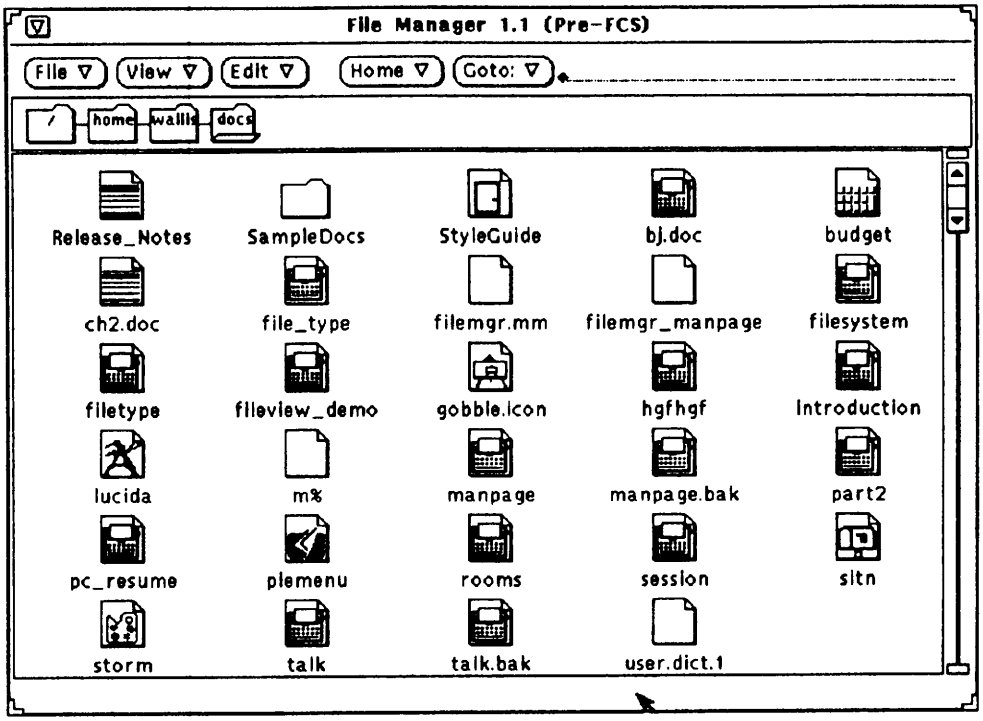


Figure 3: *The File Manager*

4.1.2. Application Responsibility

As a minimum, the application should handle the ACTION_DRAG_LOAD event by interposing on the window with a callback procedure. The following example from *textedit* queries the notifier for ACTION_DRAG_LOAD, requests the primary selection, and loads the window with the file.

```
static Notify_value
textedit_load_event_proc(textsw, event, arg, type)
      Textsw                  textsw;
      Event                   *event;
      Notify_arg              arg;
      Notify_event_type       type;
{
        char    document_name[80];

        if (event_action(event) == ACTION_DRAG_LOAD) {
                if (textedit_get_primary_sel(document_name))
                        return(NOTIFY_DONE);
                xv_set(textsw,
                        TEXTSW_FILE, document_name,
                        0);
                return(NOTIFY_DONE);
        }
        return notify_next_event_func(textsw, event, arg, type);
}
```

Additionally, the application should watch for the ACTION_DRAG_MOVE event. This event is raised when the user tries to move the application's icon on the Workspace. If the move request is to another tool's window, the application should send the destination tool an ACTION_DRAG_LOAD event.

4.2. Calendar Tool

Calendar Tool is a network-based personal time management system. Developed on one of the early OPEN LOOK prototype toolkits, it can manage several calendars simultaneously. *Calendar Tool* provides the standard day, month, and week views and allows users to specify recurring appointments through an appointment editor. Built-in browsers allow remote calendars to be viewed and updated as long as permissions are set accordingly.

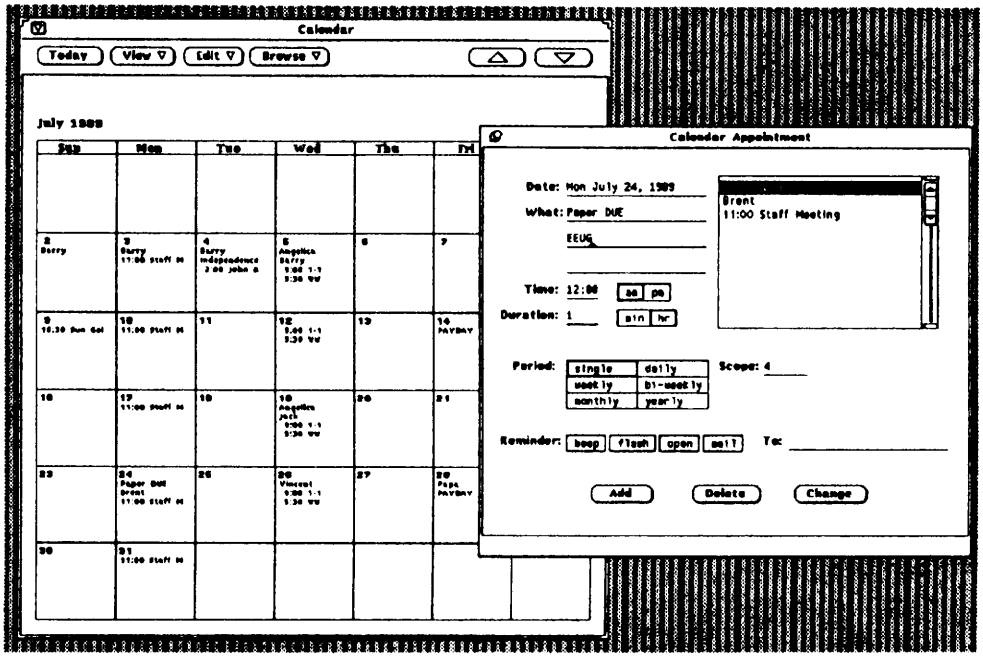


Figure 4: *The Calendar Tool and Appointment Editor*

5. Help for Porting

In general, SunView attributes fall into five categories with regard to porting to XView [Sun89b]:

Compatibility Attributes: attributes that are not part of XView proper but can still be used in XView programs. They are retained in XView for minimal conversion compatibility only and may become unsupported in future releases of XView. These attributes have no XView counterpart and generally deliver non-OPEN LOOK GUI features.

New XView Attributes: attributes that provide new features and functions or provide old ones under new names. All new attributes comply with the OPEN LOOK GUI Specification.

Redefined Attributes: attributes that are aliased (#define) to new XView attributes. All redefined attributes have true functions in XView but have retained their names to support minimal conversion efforts.

Defunct Attributes: attributes that are not carried forward into XView in any form.

Unchanged Attributes: attributes that are carried forward into XView and have the same name and functionality.

The XView Reference Manual: Converting SunView to XView presents an object-by-object comparison of the attributes in SunView and XView showing the fate of each attribute. Conversion to XView falls into two categories: minimal and full. Minimal conversion is probably adequate for low-end investment applications, not requiring OPEN LOOK compliance such as those limited to in-house use. High investment, end-products require full conversion.

5.1. Minimal Conversion

Available with XView is a shell script *convert_to_xview* which uses *sed* to help automate the conversion process. This script removes all backward-compatible attributes and makes a complete a translation to the XView API as possible. It also flags all issues that require programmer attention with references to the appropriate section of the conversion manual. However, it does nothing to help with issues of user interface style and it always requires some programmer "post-processing".

5.2. Full Conversion

Full conversion requires the removal of all SunView compatiblility features *and* a redesign for OPEN LOOK compliance. The *convert_to_xview script* is a starting point for code conversion, followed by pure programming manual labor. Fortunately, there's help for prototyping the user interface design.

5.2.1. OpenWindows

GUIDE (Graphical User Interface Design Editor) is a development tool designed to give programmers the freedom to create and test user interfaces without having to write a line of code [Sun89c].

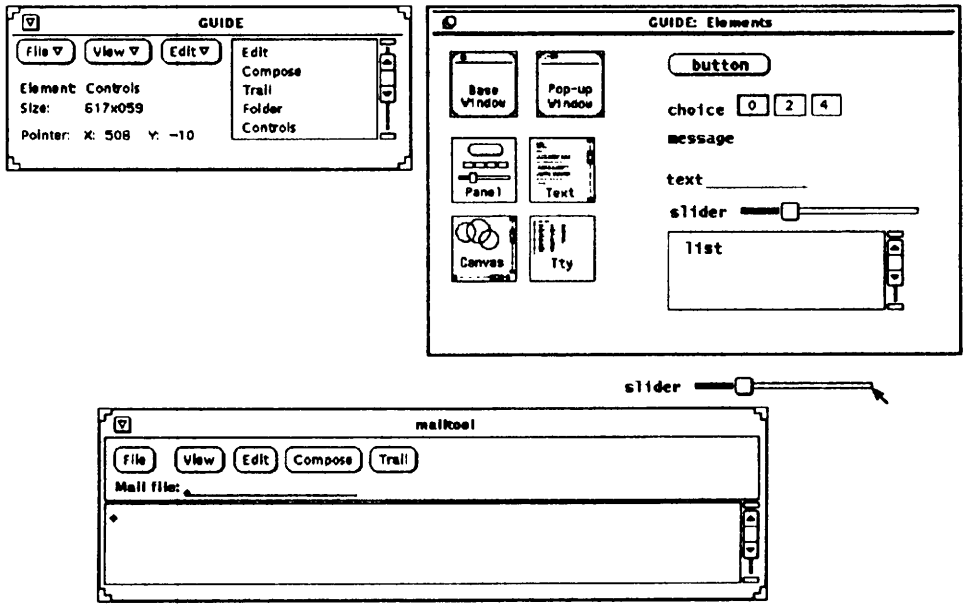


Figure 5: *OpenWindows Guide*

OPEN LOOK user interface components are generated by dragging visual representations of the components onto the Workspace. The components can be interactively named and rearranged as the programmer adjusts his design. When the user interface is assembled, *GUIDE* writes an interface file which can be recalled later for modification or generation of XView toolkit code. *GUIDE's* companion program, *gxv* reads the interface file and generates the C source code necessary to create that user interface in XView. *Gxv* also generates makefiles, header files, and stub files to help the programmer link the interface to the main body of the application.

Although *GUIDE* is primarily a programmer's tool, it is also useful for non- programmers such as software designers and project managers. It allows projects to be naturally split along user interface lines to increase parallelism and team effort.

6. Conclusion

How hard is it to port to XView? The answer is, "It depends on the goals of the application." A minimal conversion can be achieved using the *convert_to_xview* script with some post-processing programming effort. A *full* conversion to XView requires a working understanding of the XView architecture, a commitment to the OPEN LOOK user interface style guidelines and a moderate programming effort.

References

[Jac89a] Thomas W. R. Jacobs, "The XView Toolkit: An Architectural Overview," *The 3rd Annual X Window System Technical Conference Proceedings*, January 1989.

[Sun89a] Sun Microsystems, Inc., *XView Reference Manual, Beta2 Release*, May 1989.

[Sun89b] Sun Microsystems, Inc., *XView Reference Manual, Beta2 Release*, May 1989. (Part No: 800-2483-06)

[Sun89c] Sun Microsystems, Inc., *OpenWindows GUIDE User's Manual, Alpha Release*, July 1989.

Appendix A

In this appendix, "Hello World" has been implemented on the XView Toolkit using Xlib drawing routines. When compiled with XView and Xll include files and libraries, the resulting application will appear as it does in Figure A.

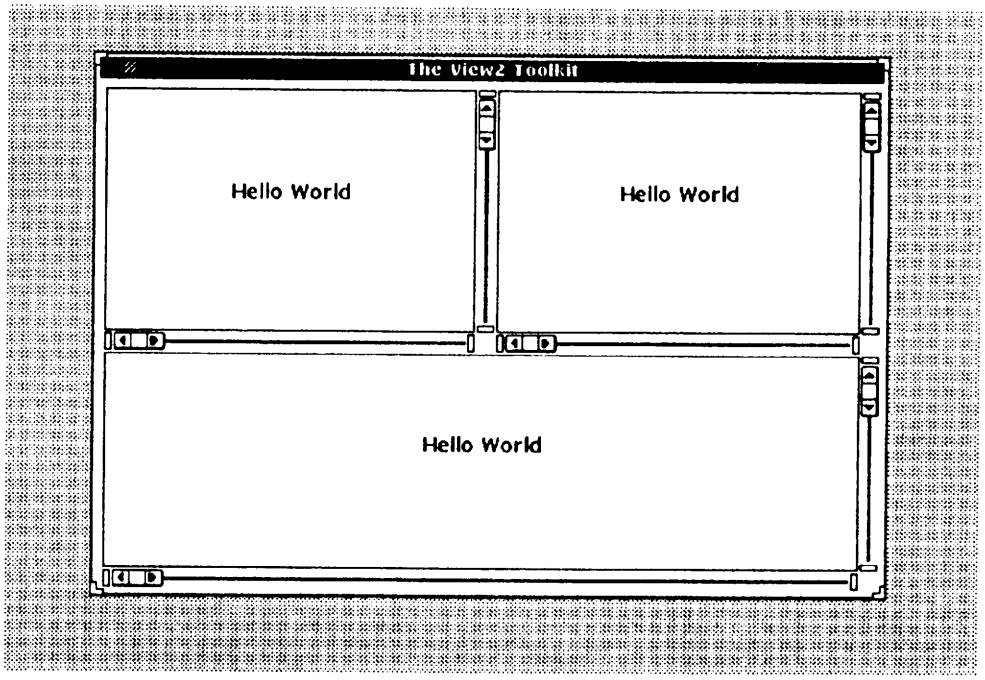


Figure A: *Hello World in XView*

Listing of "Hello World"

```
/*   Hello World in XView */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/font.h>
#include <xview/scrollbar.h>
#include <xview/xv_xrect.h>

int             str_id;
Font_string_dims     str_dims;
GC              gc;
Xv_font              new_font;

main(argc, argv)
```

```
 int argc; char **argv;
{
     Xv_Window  root_win;
     Rect       *root_rect;
     Xv_font         win_font;
     Display         *display;
     XGCValues  gcval;
     Drawable   xid;
     Frame      frame;
     Canvas          canvas;
     static int my_repaint_proc();

/* Initialize XView and process command-line arguments. */

     (void) xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, 0);

/* Create a frame and give it a title. */

     frame = (Frame) xv_create(XV_NULL, FRAME
          FRAME_LABEL,    "Hello World",
          0);

/* Get the root window (parent of the frame, then
   get the rectangular dimensions of it.
*/

     root_win = (Xv_Window) sv_get(frame, XV_ROOT);
     root_rect = (Rect *) xv_get(root_win, XV_RECT);

/* Create a canvas as a child of the frame.  The canvas should:
     - be reasonably sized (1/4 size of the display)
     - have splittable scrollbars, horizontal and vertical
     - call my_repaint_proc() when repainting/resizing
*/
     canvas = (Canvas) xv_create(frame, CANVAS,
          CANVAS_WIDTH,   root_rect->r_width/4,
          CANVAS_HEIGHT,  root_rect->r_height/4,
          CANVAS_X_PAINT_WINDOW,     TRUE,
          CANVAS_REPAINT_PROC, my_repaint_proc,
          0);
     (void) xv_create(canvas, SCROLLBAR,
          SCROLLBAR_DIRECTION, SCROLLBAR_HORIZONTAL,
          SCROLLBAR_SPLITTABLE,TRUE,
          0);
     (void) xv_create(canvas, SCROLLBAR,
               SCROLLBAR_DIRECTION,     SCROLLBAR_VERTICAL,
               SCROLLBAR_SPLITTABLE,    TRUE,
               0);
     (void) window_fit(frame);

/* Get the default font of the frame.  Rescale the font
   to the largest sized font in that family-style font group.
*/
     win_font = (Xv_font) xv_get(frame, XV_FONT);
     new_font = xv_find(win_font, FONT,
          FONT_RESCALE_OF, win_font, FRAME_SCALE_EXTRALARGE,
          0);

/* Get the dimensions of the "Hello World" string */

     (void) xv_get(new_font, FONT_STRINGS_DIMS,
                    "Hello World", &str_dims);

/* Get the XID for the XView font object */

     str_id = (int) xv_get(newfont, XV_XID);

/* Display the tool */

     xv_main_loop(frame);
     exit(0);
```

```
}

/* my_repaint_proc is called by the canvas whenever the canvas
   sustains damage, is split, is joined or is scrolled.
   it clears the area of the window in its view and then centers the
   string with its view.  The first time it is called, a
   GC (graphics context) is created and cached.
*/

static int
my_repaint_proc(canvas, paint_window, display, xid, repaint_area);
     Canvas canvas;
     Xv_Window paint_window;
     Display *display;
     XID xid;
     Xv_xrectlist *repaint_area;     /* ignored in this example */
{
     Rect *rect;
     int new_x, new_y;

     if (first_time) {
          gcval.foreground = BlackPixel(display, DefaultScreen(display));
          gcval.background = WhitePixel(display, DefaultScreen(display));
          gcval.font = str_id;
          gc = XCreateGC(display, xid,
               GCForeground | GCBackground | GCFont, &gcval);
          first_time = 0;
     }

     rect = (Rect *) xv_get(canvas, CANVAS_VIEWABLE_RECT, paint_window);
     new_x = rect->r_left + (rect->r_width/2) - (str_dims.width/2);
     new_y = rect->r_top + (rect->r_height/2) - (str_dims.height/2);

     XClearArea(display, xid, 0, 0, rect->r_width, rect->rect_height, False);
     XDrawString(display, xid, gc, new_x, new_y,
          "Hello World", strlen("Hello World"));
}
```

X Display Servers: Comparing Their Functionality and Architectural Differences to Diskless Workstations

Timothy L. Ehrhart

Ericsson Telecommunicatie BV
Haansebergseweg 1
P.O. Box 8
5120 AA Rijen
The Netherlands
+31 1612 29308
Tim.Ehrhart@ericsson.se

ABSTRACT

X Display Servers are a new generation of computing devices arriving on the scene. They consist of the now familiar high resolution bitmap screen, keyboard and mouse, replacing the single font tubes and keyboard as "stdout" and "stdin". These devices contain a powerful microprocessor, local memory, network interfaces, built-in fonts, and run only the X11 server code (i.e. not running UNIX). They don't utilise any of the local compute power in application support. The applications, or clients, in X terminology, all run remotely on another host-based computer. The output of the remote client is displayed on the local display, managed by the X11 server. These devices appear to offer the same functionality as a Diskless Workstation. Can the two be compared in an apples-to-apples fashion?

Introduction

Up till now the the lowest common denominator into the bit-mapped, window-oriented, mouse-driven user interface device world had been the Diskless Workstation. With the arrival of X Display Servers the ante has been lowered. We have been struggling to replace the dumb ASCII terminals as the absolute lowest common denominator user interface device able to communicate with a computing device. This has been a difficult task just given the economic implications. The cost of a single Diskless Workstation compared to the cost of a dumb ASCII terminal has been difficult to justify. We have used arguments such as increased productivity, ease of use, etcetera, to justify our Diskless Workstation purchases. Even though the cost of Diskless Workstations has been dropping rapidly, they still are at least an order of magnitude more expensive than an ASCII terminal. It is not difficult to demonstrate the virtues of window based displays, it's just hard to swallow the admission price.

The installation cost of additional user interface devices has also been a significant factor. The ease of just hanging another ASCII terminal off an existing time sharing system with RS-232 cable is much simpler and lower in cost than laying coax cable for the connection of a Diskless Workstation.

Slowly, but surely, we are over coming the hurdles. We are retiring our antiquated time sharing systems. There's a good chance we already have put the coax cable in the ceiling, or have accounted for it's installation in next year's fiscal budget. Our plans for total conversion to bit-mapped display devices are halfway realised. All the software developers already have their Diskless Workstations, but what about the rest of the staff? Well maybe next year...

ASCII Terminals – Just Say No

Help is on the way. X Display Servers offer a much lower per seat cost into the wonderful world of *windows*. Let's re-evaluate this year's budget... For every Diskless Workstation we planned to buy, we can maybe purchase 2.5 X Display Servers. Perhaps we can breath so life back into our not yet retired time sharing systems by turning them into compute server engines. The new file servers we planned to purchase along with our existing file servers can easily support the X Display Servers we can now purchase. Now if we could just find some convenient way to write off the retirement of the ASCII Terminals. Wait, I know, let everyone take one home with a modem. Many of us would be foolish enough to work at home in the evenings and weekends, so the company would easily recoup any costs incurred by giving us a terminal and a modem at home. How about this for an idea: Why not just give us an X Display Server at home instead? Maybe we can find some charitable institution to donate the ASCII terminal to and get a nice tax write off too.

Remote Windows – New Hope

In the past, working remotely in many case meant ASCII terminals. It was impossible to take our *windows* with us. Anyhow what good is a Diskless Workstation without being able to communicate with it's file server? Take an X Display Server running SLIP over the serial interface and combine it with a fast (9600 baud) modem and what do you get: Remote windows. This is a great breakthrough, no longer will we have to read NetNews at home on dumb ASCII terminals with 2400 baud modems. Maybe we can even convince the company to let us work at home a few days out the week. Ah, the future is looking brighter all the time.

What about Terminal Servers?

Don't they provide the same kind of functionality as X Display Servers? They allow low cost user interface devices access to the IP networks of the world. They have the major drawback of only providing the capability supporting ASCII terminals. We must look for solutions that put *etch-a-sketch* user interface devices into all user's hands.

Evolution or Revolution?

Are we entering a new era of computing, or is this a digression back to the concept of centralised computing? We have watched the evolution of computing go from batch processing, where several users shared a single computer in a serial fashion; to time-sharing, with many users having simultaneous access to one system; to desktop computing, where each user has a dedicated computer, such as a PC or a Workstation on their desktop. Are we seeing a new revolution in computing emerging or is this just the next logical evolutionary step? Time sharing with network based window systems accessing centralised shared resources might be a description of this new trend. Perhaps a more appropriate name would be **network computing.**

These products have come about due to the emergence of DoD, industry wide and de facto standards in the operating system and window system/platform arenas. The adoption of the UNIX operating system, Ethernet based LANs, and the TCP/IP network protocol suite along with operating system and hardware independent server based window systems have brought us to where we are today. With the demise of proprietary kernel based window systems, and the emergence of X Window System and NeWS server based window systems, client application no longer need run on the CPU providing the window display functionality. One is also relieved from the previous requirement of having UNIX running on every user's desk because the window system was tied to the kernel.

Mohammed Goes to the Mountain

Network computing can best be described as going to the best suited computer to accomplish a given task. One runs the application on the computer that controls the desired resource, peripherals or file systems, as opposed running all applications on the local CPU irregardless of the network resources utilised to bring the task and required resources to the local CPU. There is no need to do paging over the network or bring executable images over the network. With X Display Servers, executable images run on the remote host computer. Any resource accessed is also managed on the remote host computer, only the output of the remote client travels over the network to be displayed on the local X Display Server.

Comparing Apples to Apples

So that we are comparing apples to apples, let's specify what the minimum configurations for each type of machine. The Diskless Workstation is a 3 MIPS class machine, has a Monochrome, 1 Mega Pixel monitor, Ethernet interface, keyboard, mouse, and 8MB RAM. The X Display Server has a Monochrome 1 mega Pixel monitor, Ethernet interface, keyboard, mouse, and 2.5 MB RAM. Both are running the full TCP/IP protocol suite, including support for TELNET, ARP/RARP, BOOTP, TFTP, NFS, ICMP/PING, SLIP, and provide Domain Name Server support.

Minimum Memory Requirements

What is the minimum amount of RAM necessary in each configuration to make them acceptable user interface devices? Response time is a very important factor in computing. Users will not easily tolerate slow devices, irregardless of what the reason for the slowness. They can't see if the network is overloaded, or if the device is paging and swapping itself silly, or if the file server disk I/O bandwidth is all used up. We tend to try and solve the problem in many cases by simply giving the user more physical memory, and in some cases faster CPUs. With today's heavily layered software architectures much more CPU bandwidth and physical memory is required. Diskless Workstations require, if not demand, at least 8 MB RAM to be sufficiently fast. The days of Diskless Workstations with 4 MB RAM have come and gone. That's all right, the advances we have made have been in the right direction, they have all been made in the name of portability, modularity, and flexibility.

In the case of X Display Servers most come with a minimum memory configuration of 1-1.5 MB RAM, expandable up to a maximum of 4-4.5 MB RAM. This is much less than the Diskless Workstation, but it's requirements are different. It runs only the X11 server code and the TCP/IP protocol suite. Some of the X Display Servers run their code out of PROM so they don't have to give up RAM memory to hold the server and networking code. Power hungry Diskless Workstation users transitioning over to an X Display Server will require more than the minimum 1-1.5 MB RAM provided because of the rich set of tools they are used to running concurrently. Fully loading up a X Display Server with 4.5 MB RAM gives one an almost inexhaustible resource. I say almost, because the RAM in the X Display Servers is physical memory only, not virtual memory.

What happens when an X Display Server exhausts it's supply of physical memory? One of two things, depending on the severity of the memory exhaustion: It could die a death that is not to be envied, perhaps crashing the server, causing it to re-initialise itself, along with severing all existing connections it has with currently running remote clients; or denies the access of the remote client to the local server. In which case the remote client quietly dies. The local X11 server does attempts to return an error message back to the remote client that it cannot satisfy it's request of resources from the server. As of this moment, most clients don't know what to do with this error message. The X Consortium is investigating this important issue.

Only one of the current X Display Server product offerings has a trick to use the virtual memory of a host based computer. Extension are made to the X11 server code on the host based computer to allow the X11 server running on the X Display Server to use the virtual memory capabilities of the host based computer.

Per Seat Cost

Depending on which vendor purchased from, the quantity one price for the Diskless Workstation would be about $7-10K, the price for the X Display Server would be about $2-4K. When comparing only the per seat cost the X Display Sever is cheaper by a factor of at least two.

System Wide Manpower Costs

When integrating Diskless Workstation and X Display Servers into a heterogeneous computing environment what are some of the costs that need to be looked at on a system wide level? Naturally one of the most expensive items in a budget is in the staff required to maintain the resources of the network. Assuming an always present central core of server class machines that must be maintained, be they file servers or compute servers, much less manpower is required to maintain each additional X Display Server as compared to additional Diskless Workstations.

This is because X Display Servers have no local file systems, and have a one-time-only initial configuration process. In this initialisation you specify such things as it's IP address, broadcast address, netmask, boot server, name server, default gateway, and network font support host. Their IP entity specific information can be retained in NVRAM, and/or they have the capability to get their configuration information via the network, allowing it to be updated and maintained in centralised locations.

When looking at Diskless Workstation, each one has multiple file systems that must be maintained on an individual Workstation basis. Individual Diskless Workstation system backups, host and architecture specific file maintenance requires more manpower and very sophisticated/automated tools and procedures.

Software Updates

Software updates are also a major issue. With Diskless Workstations, every Workstation specific file system must be updated. X Display server software updates, when they are required, can be quite simple. Most have the capability to load their executable image over the network using BOOTP or TFTP, once again allowing it to be updated and maintained in a centralised locations. Software updates for X Display Servers should not really be much of an issue, the X11 server code implementations seems to be very stable, and the networking code is up to BSD 4.3 standards.

It is the clients, toolkits, and User Interfaces toolkits that are always changing and evolving. All that work will take place on the host based computers, with the result that the server code in the X Display Servers can remain unchanged and will be able to communicate with and display the output of the new client applications.

File Servers or Compute Servers?

What kind of computing resources are required to support these two types of devices. Diskless Workstations typically require file servers to provide boot, root, swap, and file access. All processes run locally on the CPU of the Diskless Workstation. This means, they must page and swap over the network to make physical memory available to hold the executable image they want to run, and any resource accessed by that process must also be brought over the network to the Diskless Workstation. This can amount to large number of packets being generated. In comparison, the remote client running on the desired host simply opens a connection to the local X11 server. The output of the application is the only network traffic generated.

Let's assume a typical server class machine has 4-5 MIPS, with approximately 1-2 GB disk, 16 MB RAM. We know this class machine can easily support about 10 Diskless Workstation without much trouble. Due to the lighter network and disk activity not required to support root, boot, swap, and file access for Diskless Workstations, I would draw the conclusion, that the server would be able to support as many, if not more X Display Servers.

Network Impact

Before attempting to say anything about the network traffic that is generated by these type of devices, one must always preface it with: YOUR MILEAGE MAY VARY. The factors having an effect on any conclusions you want to draw are almost to numerous to mention. The biggest factor is the type and mix of application you run. In general Diskless Workstations generate more network traffic than X Display Servers because they rely on the network for all file access, whereas X Display Servers transmit only the graphics portion of an application over the network. The difference is quite clear: Diskless Workstations implement the disk over the network utilising the NFS protocol, X Display Servers implement the graphics over the network using the session layer *Xwire* protocol.

The place where X Display Servers are very weak and network traffic intensive is when they are used in a remote virtual terminal emulation. Every character typed on the keyboard generates two network packets. One to send the packet to the remote emulator, and the other is the response packet back used for character echoing. The Diskless Workstation has a very clear advantage where running normal terminal based applications. The local CPU takes care of servicing the keyboard interrupts without generating an network traffic. That is not the fault of the X Display Server, we must start to develop applications that are less reliant on the dumb ASCII terminal emulation paradigm.

With the emergence of graphical user interfaces such as Open Look and Motif, maybe we will begin to see re-writes or ports of our most popular terminal based application into the windowing world.

Network Worthiness

How robust are the networking protocol suites implemented in X Display Servers? Can they hold their own against a UNIX based Diskless Workstation? For the most part, yes; they are full players on the network. They have Domain name server support and network booting capabilities. They speak both ARP and RARP fluently. Many even have monitoring or management capabilities that keep track of all vital network statistics. They collect statistics on the transport layer on a per protocol basis, network layer, and data link layer packet counts, collisions, and re-transmissions.

Font Support for the Bitmap Displays

In order to support multiple fonts on the bitmap displays, fonts are required. How do the two types of devices get their font support? In the case of a Diskless Workstation, all font support comes via the network by NFS file access. X Display Server have several ways of getting fonts. In the first case many of the X Display Servers have built-in PROM resident fonts, further fonts can be gotten from the network via NFS file access or TFTP transfer. Most X Display Servers have the capability to hold the retrieved fonts in RAM till the server is reset.

Conclusions

I have observed Diskless Workstations and several X Display Servers that were running in an existing large heterogeneous environment (subnetted Class B network, 3+ physical cables, 100+ Workstations). Initial data gathering has shown that in most cases less total network traffic is generated to accomplish a simple set of tasks which are done on a normal daily basis. These findings are not conclusive nor valid for everyone. I also did not have a sufficient number of X Display Servers to get a real feel for what their overall impact on the network would be.

I feel that one has the same functionality at the end user level at a much lower cost when compared to Diskless Workstations. X Display Servers have many advantages, but are they right class of machine for every user? Definitely not, there is still a lot to be said for UNIX operating system based host computers as user interface devices. You have much more control over them and their network interaction. Certain types of software development demand having control of their own resources for testing purposes. Certain compute intensive or graphics intensive applications, such as desktop publishing and CAD/CAE, are much better off being run on powerful single user Diskless Workstations. In these cases it is worth the extra investment to have access to larger displays and support for colour.

X Display Servers have raised the lowest cost denominator for user interface devices from a line and character oriented interface up to a window oriented graphical user interface. Sales, Marketing, and Administrative departments, as well as Technical Editors and Managerial staff don't often require the punch offered by today's Diskless Workstations. The average Diskless Workstation user is for a great majority of the time under-utilising the available power of his Diskless Workstation. They do still require a bit-mapped, window-oriented, mouse-driven user interface device, and access to network-based high performance windowing applications. This class of product is a well suited for these types of users.

Some of the drawbacks or shortcomings of this first phase of X Display Server products are potential server crashes at resource exhaustion, and limited screen resolutions or dimensions. I am sure future offerings will come with increased screen resolutions and if the market demands it, support for colour. There is also one extra added bonus: When all else fails, they can still function used as dumb ASCII terminals.

A System for the Redirection of Graphical User-Interaction

Robin Faichney

Computing Laboratory
University of Kent at Canterbury
Canterbury
Kent
CT2 7NF
UK
rjf@ukc.ac.uk

ABSTRACT

The redirection of user-interaction is complicated in a graphical environment due to the nature and variety of graphical actions, compared with character stream input/output. The solution is to redirect at a high level within the application. Accordingly, this system is designed to allow the redirection of all traffic between the user-interface and the underlying functionality of an application, to and/or from another program. This facility is quite flexible and has a variety of uses. These include facilitation of the automatic testing of the underlying functionality of graphics-interfaced applications, and allowing extensibility of user-interfaces, via the recording and replay of command macros. A prototype and some applications which use it have been implemented, demonstrating the main features of the system.

Introduction

This paper describes one of the fruits of an attempt to tackle a problem which was identified during work on developing software tools for the UK Science and Engineering Research Council's EASE (Engineering Application Support Environments) programme, for which Kent is the Software Tools Centre.

The problem was concerned with the interconnection of highly interactive software modules, and in particular the possibility of carrying over the benefits of UNIX command line interconnection (input/output (I/O) redirection) into the WIMPS (Windows Icons Mouse Pointer (or Pull-down menus) Systems) environment. An analysis was made of interconnection in the graphical environment and this is described in previous papers. (The earlier paper [Fai89a] contains an abstract analysis, and the later paper [Fai89b] a more concrete one.) It was concluded that interconnection in this context falls naturally into two categories, and that two separate systems were required to facilitate such interconnection, one for each category. This paper describes one of these systems. The other is described in the previous papers. (The earlier paper contains a more detailed description.)

The first section below is concerned with justifying the development of a high level command redirection system. Following this some of the concepts which are common to any such system are elucidated. Then this system is described. The next section discusses some applications built using the system, and that is followed by a summary of the paper and some acknowledgements.

1. Justification

Previous papers are concerned with the problems of interconnection in a graphical environment and the need for two systems to facilitate this, one each for data communication and for command (user I/O) redirection. [Fai89a, Fai89b]

This section will consist of a brief reiteration of the portions of that argument which are relevant here.

The general aim of this work is to reintroduce the benefits of UNIX command line interconnection to the graphical workstation environment. The traditional UNIX tool is not interactive. It takes a stream of data, performs some transformation upon it and outputs the result, the transformation being constant throughout the run. Command line interconnection depends upon the simplicity, and especially the uni-directional nature, of this I/O scheme. If interactive tools are to be interconnected, data I/O has to be separated from user I/O. In this way they could be used in a pipeline – where data is communicated, but user I/O is as normal – or user I/O alone could be redirected to allow one program to control another. This paper is concerned with user I/O (or command) redirection (which is the harder of the two).

Traditional I/O redirection is dependent upon character stream I/O. This can be handled almost equally easily by people and by programs, so simple switching between terminal, file and program is sufficient. The reason for the development and popularity of graphics-interfaced programs is that their I/O is specially designed for ease of use by people. Consequently, it has become very much more difficult for other programs to utilise. Not only has the range of interaction types become much greater – colour changes, mouse movements, etc., as well as text – but the range of interpretations placed upon any one type by different programs is much greater too. The obvious solution to such low level diversity is to use a higher level instead. The interface between user-interface and the underlying application functionality is not only the highest level at which user I/O can be captured, but probably the only level at which it is practical to try to define the user I/O of any interactive application. Such definition both requires and facilitates the separation of user-interface from application functionality which is very widely advocated within the human-computer interface community [Coc88a]. It was therefore decided that any system intended to facilitate command redirection in a graphics environment should deal with the traffic between user-interface and underlying application functionality.

2. Command Redirection

This section deals with certain general issues regarding high level command redirection. The concepts developed here will be used in the description of this particular system, which follows.

2.1. Inter-Program Relationship

The nature of command redirection is such that the relationship between the programs concerned is inevitably asymmetrical. One has its user I/O redirected, and for the other this is simply data I/O, with no direct relevance to its own user-interface, if it has one. Throughout this document the latter program is referred to as the "controller", and the former as the "controllee"; generally the program which is having its user I/O redirected, can be viewed as being controlled by the other, though it is not always so.

2.2. Inter-Component Relationship

The controller, and the user-interface and functionality of the controllee, are the "components" of the redirection arrangement.

Though there are three components, each with an input and an output, the ways in which they might usefully be interconnected are actually quite limited. The "normal" arrangement – user-interface and functionality connected – is one obvious requirement. For all redirection arrangements, the controller is involved, so such arrangements may be considered from its point of view. As the overall aim is the redirection of the user I/O of an interactive application, both input and output must be considered in each case. As far as the redirection system is concerned, there is no need to distinguish between user-interface and functionality; the arrangement is symmetrical in this respect. So there are only two main alternatives:

(a) both input and output of the controller connected to one of the controllee components ("one-sided"); or

(b) the controller input connected to one controllee component, and its output connected to the other ("two-sided").

Within these two main alternatives, there are sub-divisions. Generally, wherever the input or output of a controllee component is connected to the controller,

(i) it may also remain connected to the other controllee component ("teed"); or

(ii) it may be cut ("unteed").

As in traditional redirection, the controllee components should not be aware of the source/destination of their communications. From the point of view of the individual controllee component, the controller simulates the other component. "Simulation" may also take place at a higher level, however: where the controller is connected to just one controllee component (one-sided redirection), the controller is, from the application programmers point of view, simulating the other component. "Simulation" implies replacement, so the normal interface-functionality connections should be cut (unteed, see Figure 1).

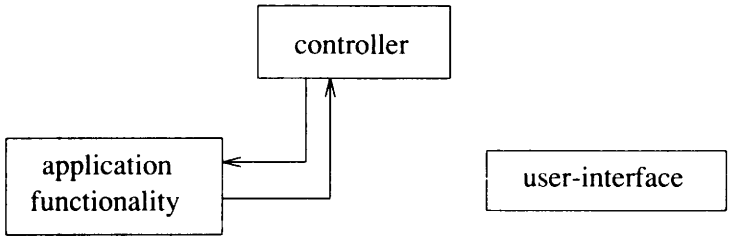


Figure 1: *Simulation of user-interface*

On the other hand, as an alternative we might allow a special type of simulation which allows in effect "duplication" of a controllee component, by leaving the normal connections intact (teed, see Figure 2).

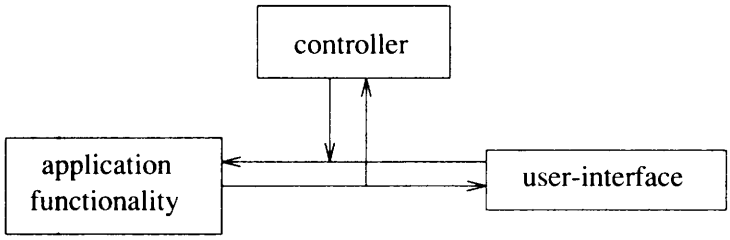


Figure 2: *Duplication of user-interface*

An analogy can be made with the **tee** [ATTa] utility in traditional redirection. So the sub-divisions of one-sided redirection are simulation and duplication of either of the user-interface and the functionality.

To return to the point of view of either of the controllee components (the lower level type of simulation): under two-sided redirection, the controller simulates different components at different times. For instance, if controller input is connected to the user-interface, and the output to the functionality, for the recording and replay of command macros ("taping"), on user-input the controller simulates the functionality, while on controller output it simulates the interface. For each of the controllee components, the controller simulates the other, but only for either input or output, not both. At the time of writing the only purpose foreseen for two-sided redirection is taping. With teed connections, in this context, the functionality remains connected so that user-input intended for recording is also passed directly to it for immediate execution, the result of which is passed back to the interface, while on replay the user can interrupt with a "live action" at any time, and again functionality output is allowed through to the interface. It would seem not only that the choice between teed and unteed on recording is independent of that on replay, but that even within these, teed might be chosen for interface-controller and unteed for controller-functionality, or vice versa (Figure 3).

It perhaps should be reiterated that the difference between simulation at the higher and at the lower levels is that at the lower level, the controller always "looks like" the user-interface to the functionality and vice versa, while at the higher level one of the functions of redirection is to use the controller to replace either interface or functionality. From this point on, the word "simulation" should be taken to mean the higher level, component replacement, function of redirection.

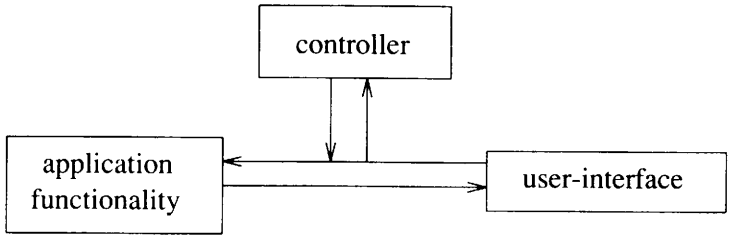


Figure 3: *Taping of user-interface, connections teed*

One other factor regarding the topology of these components has to be considered. Where simulation of a component is taking place, in principle that component need not exist. In other words, the application (controllee) may be constructed without either a user-interface or a functionality component, using another program (controller), through the redirection system, instead.

It should be noted that there is a significant difference between taping on one hand and simulation/duplication on the other, which may or may not occur also with other two-sided uses of redirection. Simulation and duplication are relatively monolithic, or single state, arrangements, perhaps in many cases persisting throughout the life of an instance of the application. Taping, however, consists of a number of different states: recording and replaying, and a state in which neither of these is occurring ("normal" operation); there is also a number of ancillary functions.

Automated testing of graphics-interfaced applications has been mentioned as a possible use for command redirection. The fact that redirection is to occur at a high level, however, restricts the sort of testing which may be done this way. A testing module would take the form of a controller, and would be connected to the component to be tested, by simulating or duplicating the other controllee component. So only one of the components could be tested at a time. Where the user-interface was to be tested, as interactive programs are normally user-input-driven, a user would still be required, and in general, little if any benefit over present testing methods would occur. On the other hand, automated testing of the functionality would seem to be quite practical, and it might be made easier to do certain sorts of user-interface testing, by using simulation of the functionality to enable an application to be more rapidly prototyped.

The various permutations described here – simulation with or without the presence of the simulated component, duplication, and taping (or other two-sided uses of redirection) with the teed versus unteed option on each connection, of each of the user-interface and the underlying application functionality – are conceived as different "types" of redirection. In order to avoid undue complexity, it was decided that only one type of redirection should be possible between any two programs at one time. Simulation in the absence of the simulated component was not considered worth supporting, for similar reasons, especially the fact that its possible uses would seem to be more easily achievable in other ways. Further restrictions on types of redirection are built into the present implementation of the system, but may not exist in future versions; see next section.

3. The Command Redirection System

This section consists of a general description of the Command Redirection System (**crs**) followed by more detailed descriptions of various aspects of it.

3.1. General Description

Crs consists of a library of routines through which all traffic between the user-interface and the underlying functionality of an application is routed.

A certain crs routine is called wherever such traffic is required. Other crs routines are for initialisation, switching redirection on/off, handling crs input and for a number of functions concerning macros.

When designing a program which may utilise redirection, all controllee routines which deal with the highest level of user I/O have to be identified. This means those routines belonging to the functionality which are called by routines belonging to the interface, and vice versa. The application must be designed so that, at this level, every routine belongs wholly to one component or the other. See "Defining User Input/Output". We term the highest level routines which are called from outwith their own component,

"target" routines.

When a target routine is to be called, the application calls a certain crs routine instead. In some circumstances, crs will then call a routine corresponding to the target in the other program, as well as or instead of the target itself.

For simulation and duplication, the arrangements at this level for controller and controllee are approximately symmetrical. A target call at the controllee will result in the corresponding controller routine being called, and vice versa. Differences are that targets are defined as such relative to controllee user I/O (that of the controller being irrelevant), and that the controller does not distinguish between simulation and duplication.

For taping, the situation is somewhat different: if recording has been switched on, a target call at the controllee results in the appropriate information being stored at the controller. On replay, the functionality target is called as if from the user-interface (or perhaps vice versa, for some other use of two-sided redirection – see "Inter-Component Relationship").

Given the restriction that only one type of redirection is allowed between any two programs at one time, to further avoid undue complexity controllers are specific to just one redirection type, in this implementation at least. (For reasons which will become clear below, they are also specific to just one controllee.)

3.2. Target Routines

As the targets have to be called by crs, there are inevitably restrictions on their format. In fact, they must be of type `void` and take just one argument, of type `caddr_t`. The actual type required is simply cast to/from `caddr_t`, unless it is too large to fit within it, in which case a pointer is used. In this way any type can be passed, if necessary using a structure where more than one argument is required.

As target arguments have to be passed between different processes, they have to be serialised and deserialised. This is done using "conversion" routines.

3.3. Conversion Routines

Sun's **xdr** routines [Sun86a] (now available also on many other architectures) are used for de/serialisation of the target arguments. (There is a standard method for handling user-defined types.) The programmer provides a conversion routine which is just a wrapper for the appropriate xdr routine. Each xdr routine does two way conversion, therefore so does each conversion routine. The conversion routines are of return type `int`, and take two arguments: an xdr stream pointer and a `caddr_t` pointer. Xdr takes care of I/O as well as de/serialisation, and crs takes care of xdr initialisation.

3.4. Initialisation

Crs needs to know about the target and conversion routines. At initialisation, therefore, a certain crs routine is called for each target, and given pointers to the target routine and the corresponding conversion routine. It returns a target identifier, for use with the crs target calling routine.

Target initialisation is always required, whether redirection of any type is actually to occur or not. Further crs initialisation is mainly concerned with making contact with the controller, and is done automatically at the switching on of simulation or duplication. Initialisation for taping is distinct from the switching on of recording or replay, for obvious reasons.

The foregoing describes initialisation at the controllee; there are minor differences at the controller which need not be gone into here.

3.5. Source Code Considerations

Certain aspects of the source code are very important for crs: some parts should be shared between controller and controllee, and this requires a specific modularisation arrangement.

3.5.1. Controller versus Controllee

In order to guarantee correct de/serialisation of target arguments, conversion routines should be shared by controller and controllee. In the case of simulation/duplication, the conversion routines for both user-interface targets and functionality targets are required at the controller, though each set will be used for one-way conversion only. For taping, the controller requires only the routines for the other component – the one whose output is not being recorded/replayed.

So that crs can match the corresponding controller and controllee targets, target initialisation code should also be shared between the two programs. The redirection type-specific requirements for targets are as those for conversion routines (previous paragraph). However, for simulation/duplication the controller targets for the component not being simulated will never actually be called, being required only for initialisation purposes, and so may be null function pointers. Similarly, in the case of taping, the controller targets may also be null function pointers.

Where code is shared, recompilation is only necessary where function pointer declarations are substituted for function declarations, in order to provide null function pointers for target initialisation at the controller.

3.5.2. Modularisation

In order to allow the appropriate controller/controllee sharing to take place, source code modularisation must follow a certain scheme: separate modules should be used for the targets, conversion routines, and target initialisation code for each of the user-interface and functionality, and xdr routines for user-defined types. Separate header files are also, of course, required.

For applications of moderate size, the user-interface and functionality will usually occupy one module each, but where this is not the case, the targets should be confined to one module per component with lower level routines relegated to other modules. There will usually be some sharing of data structures, and perhaps even routines (though not at the level of the targets), between user-interface and functionality, but this must be kept to a minimum. See "Defining User Input/Output".

It will be seen that most if not all of the restrictions imposed by crs conform to what is generally recognised as good practice.

3.6. Defining User Input/Output

Probably the main difficulty in designing an application to use crs is the definition of user I/O. In practice, this means the design of the targets. The restriction on target routine format – that they be of type `void` and take just one argument – is not found to be a major problem in practice, but the questions as to precisely what each target should do, and what information the argument should carry, require very careful consideration.

One of the more important factors in this area is that of user feedback. When user-interface and functionality are separated, each element of feedback must be explicitly allocated to one or the other. As this is being considered, it must be borne in mind that feedback which is allocated to the user-interface will not occur when it is being duplicated or replayed and the controller calls a functionality target (i.e. the functionality "thinks" a user-action has taken place).

As an example of another typical problem, a certain application may send some text, which has been selected†, to another application. The user-actions may be recorded as a macro and replayed. So the question arises as to whether, on replay, the text which was selected when the macro was recorded should be sent, or that which is selected at the time of replay. The answer depends on the particular requirements of the application concerned. If the text selected at the time of replay is to be used, communication between controllee components at a lower level than that of complete user actions, will be required. So when the functionality target which corresponds to "send text" is called, it calls an interface routine to discover what text is currently selected. The questions of whether such lower level routines should be targets, and in general terms whether non-target communications between controllee components should be allowed, require further work. Of course the use of such routines requires that the component concerned – in this case the interface – must always be in a valid state, which has implications for the choice between teed and unteed connections ("Inter-Component Relationship").

It currently seems that little or no advice, beyond the sort of guidelines contained in this section, can be given on the definition of user I/O. It can only be suggested that the potential user I/O of the application should be very carefully analysed (which should actually happen in any case), both in normal use and under any type of redirection which may occur, and the targets designed accordingly. This issue is probably more urgently in need of further work than any other in this area. However, the benefits of redirection will probably never be attainable without some significant effort.

---

† Most interactive graphics systems allow the "selection" of text, and sometimes other graphical objects, using the mouse. Selected objects are usually either displayed in reverse video, or highlighted in some other way. There can usually be only one selection at a time.

3.7. Types of Redirection

The types of redirection provided within the current implementation of crs are simulation and duplication of both user-interface and application functionality, and taping (with both connections teed for both recording and replay; see "Inter-Component Relationship" and Figure 3). Only one type may be in operation at any time.

The current implementation is a prototype only, and providing further redirection types is a high priority. The prevention of user-actions intended for recording from reaching the functionality is obviously a desirable option. It is unlikely that all possible uses of redirection have been foreseen, and so crs has been designed to make the addition of further types relatively easy. It would have been possible to avoid this problem by giving the application programmer access to crs at a lower level, thus allowing her to program her own redirection types, but it was decided that this could well create more problems than it would solve.

4. Applications

This section contains descriptions of two applications which have been implemented using crs. In both cases, an existing application was re-written to use crs. The final subsection discusses experience obtained through working with the applications.

4.1. Pconf

Pconf is described in a previous paper [Fai89b] and only a brief reiteration of that description is given here.

To appreciate pconf, a prior understanding of **vconf** [Fai89a] is required. Vconf is a visual conferencing utility. If a number of users at graphical workstations on a local network are running it, and one of them types a message into it, that message will be appended to a display of previous messages by all the instances of vconf. Also displayed at each instance is a list of all the correspondents to which that instance is connected. The user may remove any correspondent from the list, thus preventing any further communications with it. This correspondent is then automatically removed from the others list.

Pconf itself is a "pseudo-user" of vconf. Vconf was rewritten to use crs, and to switch on user-interface duplication if given a certain command-line flag. A controller was then implemented which not only duplicates the user-interface, but also the user, as it has no user-interface of its own. Pconf is the controller. As pconf is "wired in parallel with" the actual user-interface, it receives exactly the same information as the user. Like the user, it watches for anything to which it would like to react, then does so. Unlike the user, however, pconf does not have to operate through the user-interface; instead it calls the functionality targets directly. A "conversational simulator" along the lines of ELIZA [Wei66a] was implemented, and built into pconf. So pconf monitors vconf traffic for certain key words and phrases, and responds by sending appropriate messages in reply. Pconfs responses are not only textual, however: they can, in effect, substitute for mouse actions, thus demonstrating the value of high level redirection. For instance, if pconf is offended by a message it receives, it can remove the offender from its correspondent list, to do which would require the user to move the mouse pointer over the appropriate list entry and click the righthand mouse button.

4.2. Dprog

Dprog [Fai89a] was originally designed to demonstrate the system for data communication mentioned above ("Justification"), and so is usually described as consisting of a "thin-layer" graphical user-interface to that system. Here, however, the term "user-interface" is being used somewhat more loosely than for the purposes of crs, and so the line between the controllee components does not quite correspond with the programmers interface to the system.

The reason for rewriting dprog to use crs was to demonstrate taping. It is quite crude, for instance not allowing more than one macro to exist at any time. (In fact, at the time of writing crs does not provide this facility.) However, it is quite capable of demonstrating the principles. A macro can be recorded or played back by selecting the corresponding item on a menu (a "stop recording" item is also required). The use of this facility quite neatly shows both the possibilities and the potential problems of taping. Dprog allows a connection to be made with other instances of itself (and certain other programs) across a network, and messages subsequently to be sent across that connection. A macro can be recorded which attempts to make the connection and then sends a message on it. Unfortunately, not only is it possible for the connection attempt to fail, but even where it succeeds it may take some time to do so. In practice, for one reason or the other, the message sending attempt usually fails.

This problem brings out a quite fundamental difficulty: the incompatibility between interactive systems and recordings. Normally, the user would await confirmation of the connection before any message was sent.

Another problem revealed by dprog was the one described above regarding the sending of selected text ("Defining User Input/Output").

4.3. Experience

The problems demonstrated by dprog are genuine and important ones. On the other hand, pconf showed how powerful crs can be: once user I/O had been redefined with duplication in mind, design and implementation of the strictly crs-related aspects of vconf and pconf were quite trivial. The principles of crs are at least to some extent vindicated by the fact that, when connected to pconf, vconf is perfectly happy to "serve two masters" – pconf and the normal user-interface, with none of the three components experiencing any confusion due to the unconventional arrangement. Similar conclusions may be drawn from the fact that, presumably due to the well defined high level user I/O, and good modularisation of code, a non-graphics-interfaced version of vconf rewritten from the crs version, took only 2 – 3 hours to implement. Further work on crs is certainly required, but there seem very good reasons to believe that it will be well rewarded.

5. Summary

Previous work [Fai89a, Fai89b] showed that, in order to facilitate the interconnection of graphics-interfaced applications, two separate systems were required. One of these would deal with data communication, and the other with the redirection of user input/output (I/O). This paper concerns the latter.

In order to overcome the problem of the variety of graphical I/O, redirection must take place at a high level. A system has been designed, and a prototype implemented, which allows the redirection of traffic between the user-interface and the underlying functionality of an application.

The Command Redirection System (crs) consists of a library of routines and associated data structures. At initialisation it is given pointers to any routine belonging to the user-interface which might be called from the functionality and vice versa. It also requires routines which will convert the arguments of these routines so that they may be passed between different processes. Subsequently, where a routine requires to be called across the user-interface/functionality divide, a crs routine is called instead. Normally, this will result in the routine actually required being called, but where redirection is switched on, communication will occur with the other program. Depending on the type of redirection occurring, either a routine which corresponds to the one required will be called at the other program, or the appropriate information will be recorded for replay at a later time, which in turn will result in the originally required routine being called.

Two applications have been rewritten to use crs, which demonstrate duplication of the user-interface (two interfaces connected in parallel to the same application), and extensibility of it via the recording and replay of command macros.

Probably the major difficulty in using crs is in defining user I/O at the level required so that it will remain valid where redirection is occurring. Work will continue on ways of making this easier for the application programmer. However, programmer effort should be well rewarded especially as this sort of definition is good practice in any case.

6. Acknowledgements

The work described here is supported by SERC Research Grant GR/D/80612. P.J. Brown must take much of the credit for the original concept, and he and other members of the Software Tools Group, Computing Laboratory, University of Kent at Canterbury, in particular David Barnes, have subsequently made substantial contributions.

References

[ATTa] AT&T, "tee (1)," in *UNIX Users Manual.*

[Coc88a] G. Cockton, Interaction Ergonomics, Control and Separation: Open, Scottish HCI Centre, Heriot-Watt University, Chambers Street, Edinburgh, EH1 1HX, February 1988.

[Fai89a] Robin Faichney, "Dp: a System for Inter-Program Communication," in *Proceedings of the EUUG Spring 1989 Conference* , pp. 207-215, Brussels, April 1989.

[Fai89b] Robin Faichney, "Interconnection in a Graphical Environment," in *Proceedings of the UKUUG Summer Technical Meeting*, Glasgow, June 1989.

[Sun86a] Sun Microsystems Inc., "External Data Representation Protocol Specification," in *Networking on the Sun Workstation*, Sun Microsystems Inc,. 2550 Garcia Avenue, Mountain View, California, 17 February 1986.

[Wei66a] J. Weizenbaum, "ELIZA — A Computer Program for the Study of Natural Language Communication Between Man and Machine," *Communications of the ACM*, vol. 9, pp. 36-45, 1966.

Performance Analysis for Shared Oracle Database in UNIX Environment

C. Boccalini

I & O – Informatica e Organizzazione S.r.L.
Genova
Italy

J. Marino, M. Paolucci

Department of Communications, Computer and Systems Sciences (DIST),
Via Opera Pia 11A
16145 Genova
Italy
paolucci@dist.unige.it

ABSTRACT

The methodology used in order to execute a performance evaluation in Unix environment, in a case of concurrent access to a common Oracle database, is presented. The reason of such an evaluation, as well as the model adopted to structure the benchmarks, are discussed. Some results obtained are presented, pointing out their meaning in relation to the different hardware configurations tested and the model itself.

1. Introduction

In this paper the peculiar aspects of a series of benchmarks executed in order to compare the behaviour of different computers with a Unix operating system in a particular case of data sharing, is presented.

The reason for these tests was a tender made by Regione Liguria (Italy) to buy two types of computer which should have been installed in two Sanitary Local Units (USL) of the region.

The requirements for the firms which would take part in the tender were to provide a computer with a software environment able to support Oracle database applications, as well as an hardware configuration which should be suitable to one specific context of use and should provide good performance.

First we outline the circumstances which led to the development of the benchmarks.

Some years ago, the Regione Liguria, under the coordination of the Department of Communications, Computer and Systems Sciences (DIST) of the University of Genoa, started a project for the automation of the booking procedures for sanitary services. The task of analysing the problem, formalising the software specifications and writing the programmes which implement them, was entrusted to a software firm, the I & O (Informatica e Organizzazione, Genoa, Italy). The software built, named P.A.S.S. (Prenotazione Automatizzata Servizi Sanitary, Automatic Sanitary Services Booking), is based on an Oracle database and has characteristics of high portability. In 1988, the phases of software refinement and experimentation, executed through the observation of its real use inside an USL booking center, finished and the tender was announced.

In each USL designed for using the P.A.S.S., a booking center exists; this is composed of several booking-points which can accept the requests for all the medical services provided by the Unit itself, can appoint a day on the agenda, for example, of a doctor or of a laboratory and can register the payment of the fees. In order to avoid long queues of people at the booking-points, the software developed is able to exploit the capabilities of the Oracle RDBMS, that is, it can manage a shared database containing all the information about different medical divisions which provide the services; in other words, the problems of the organization of the booking procedures have been studied in order to avoid deadlock situations or unacceptable transaction time. The hardware configuration needed must support a minimum load of ten terminals (corresponding to ten booking-points), must operate with a multi-tasking operating systems and, finally, must be able to process a queue of contemporary requests avoiding long user waiting time. The

objective was to find a trade-off among computational power, dimension of the hardware configuration, cost of the hardware and software. The firms involved in the tender, offered their best solution for the problem discussed above. The aim of the tests was to provide one index of the quality of the machines, under the same workload configuration, from the point of view of the end user, that is, measuring the average waiting time and estimating how far each computer was from its saturation point.

Most of the firms involved presented a machine with an operating system belonging to the Unix family. Unfortunately the results of the benchmarks can not be explicitly presented here, since they are the property of the firms. In a following paragraph, however, some results will be shown, but no reference to any firm will be made.

In the continuation of the paper the method used to design the tests and to analyse the results will be presented. Furthermore, in the final part some relative measures, which are particularly meaningful because of the different configuration involved, will be discussed.

2. The methodology of the tests

2.1. The mathematical model

The general approach used for the benchmarks was to observe the behaviour of the different machines from the point of view of an end user (an operator of the booking center or a person who wants to book a service). For this reason all the computers with their different hardware configuration and Unix operating system, can be thought as one processing module which supports an external workload of tasks which comes from the ten terminals.

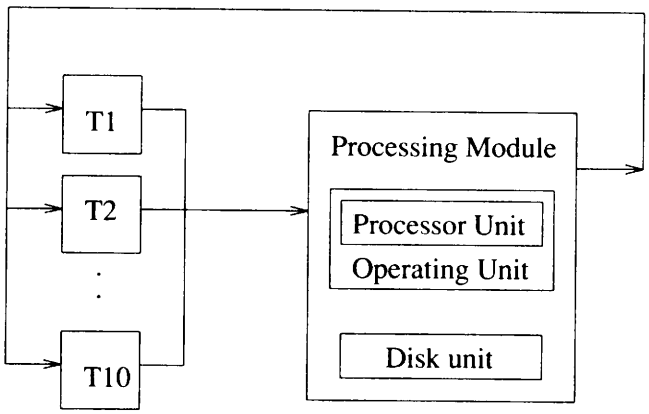


Figure 1: *A model for the systems*

The internal structure of this module is not strictly relevant for the purpose of the tests, but, in order to analyse some meaningful results, three other parts can be specified: a processor unit, an operational unit and a disk unit. The comprehensive structure of this simple model is shown in Figure 1. Some theoretical considerations which came from Queueing Theory [Kle76a] and Operating Systems Theory [Bri73a] , have to be recalled to describe the behaviour of the model.

The system analysed is an interactive system in which a finite number of users, in particular ten users, can formulate their requests through one of ten terminals. Each user request enters the processing module and proceeds to receive service according to the scheduling algorithm used by the operating unit for the time-shared processor unit.

During this time, the user remains in a waiting state. When the request is complete, the response is fed back to the terminal and the user can utilise it before making his next request. The time spent by one user in generating the new request is called "thinking-time". Periods of thinking and processing alternate for each terminal in the system. It is generally assumed that the thinking-time is exponentially distributed with a mean of $1/\lambda$ sec., where λ is the rate of request per second generated by one user; furthermore, another important measure is the mean service-time, denoted by $1/\mu$ where μ represents the rate of request served per second by the processor.

Set p_0 the probability of the steady state in which 0 terminal are waiting for a response, the probability that the processor is busy is $1-p_0$. From this fact, the average output rate of response results to be $\mu(1-p_0)$; on the other hand, the input rate of request to the system is

$$M\lambda\frac{1/\lambda}{1/\lambda+R}$$

where M represents the number of terminals and R the mean response time, that is the interval of time during which a request remains in the processing module.

Then, invoking the assumption of a steady state where the average input and output rates are equal, the mean response time R can be derived as

$$R = M\frac{1/\mu}{1-p_0} - 1/\lambda \qquad (1)$$

This model shows how the response time varies as a function of the number of terminals, M. It presents a transaction zone in which the function passes from a very slower rising to a linear slope. In the first case, this is due to the small number of users that do not keep busy the system because of their rather long thinking-time, and in the second case, the increased number of users causes the p_0 to be close to zero, that is the load of activities does not leave the processing module empty.

For a better interpretation of the results of the tests it is interesting to study the case in which the system reaches a saturation state. From equation (1) it can be observed that the saturation point the system does not becomes instable since its mean response time does not grow to infinity. Anyway, a significant measure of the system saturation was proposed by Kleinrock [Kle76a] as the cycle time, that is the sum of the mean thinking-time plus the mean required service-time, divided by the mean service-time. This is shown in the following equation, where M^* represents the maximum number of terminals that could be processed without any mutual interference.

$$M^* = \frac{1/\lambda+1/\mu}{1/\mu} = \frac{\mu+\lambda}{\lambda} = 1+\frac{\mu}{\lambda} \qquad (2)$$

For a number of terminals beyond M^*, each user request could delay all the other requests by a time equal to its entire processing time (i.e. $1/\mu$ sec.). When this happens, the probability that the system is busy approaches 1 and the mean response time R given by (1) assumes a linear asymptotic slope.

In the last paragraph we will show how the saturation number of terminals provides a very useful index; indeed, this can point out how far the limit for the workload is from the condition in which the tests have been performed.

2.2. The model of the workload

During the design of a test, a particular attention must be given to the conditions in which the different computers have to operate. What is really important in the case examined is to verify how the machines respond to a workload of requests as those that could be made during peak hours at the booking-points. As the measurements that have been collected should constitute a relevant index of the behaviour of the machines, it was adopted the solution of reproducing as well as possible the real workload that the systems should have to support. So as to obtain such a situation, the part of the software P.A.S.S. which consists of the Oracle applications dedicated to the booking-point job, was installed on the computers before the execution of the tests, together with a portion of a database used in the USL where the software was experimented by a real use. Then, it was asked ten booking-point operators, not particularly skilled in the use of the programmes, to take part in the tests. It was taught them how to make a booking for medical exams (these are cases more demanding than others) and, during the three sections of test, it was asked them to repeatedly execute that bookings.

This strategy should provide a good simulation of the real input and output flow of the system. Instead of generating a workload by means of a batch procedure, it was preferred to keep busy the different systems with a statistically equal workload and that would provide more meaningful indication of the capability of the systems. Indeed, the cost of the response time which the operator has to wait, could be directly related with the quality of the service that the booking center can offer to the people. Hence, it was not the case of stressing the tests in order to measure CPU time or response time with an high precision, since the context of use is real-time, but with the same order of the human being reaction time, that is, of seconds. Furthermore, the performance measurements represent only one index of the decisional problem, so the respect of a threshold of acceptability and the distance from a situation of saturation were the two principal information collected.

Another important consideration involves the way in which the Oracle database was tuned upon the the different machines. There are three levels of tuning operation:

- the nature of the physical installation of the database files on the disks of the systems;
- the system kernel parameters, as the maximum size of a single shared memory segment or the maximum memory per user process;
- the Oracle initialising parameters, for example, those that determine the dimension of the system global area (SGA);

The approach adopted was to ask firms to study their better solution both for the physical installation and for the Unix tuning, while, with the limits imposed by the different hardware configurations, Oracle was tuned with the same values for the parameters for all the machines.

3. The structure of the tests

Defined the conditions of "background" workload that all the machines had to support, the tests performed were structured during three sections. In each of these, a series of measures was collected in order to compute average reliable values.

The characteristics of three sections of the benchmarks are described in the following points:

a) in the first section, at least eight measures of the time of cycle were collected; in this case, cycle means a complete execution of a booking made by one booking-point operator. At the beginning of the benchmark, an operator whose computer skillness seemed to be in the average, was chosen. While the other nine operators were loading the system, the time used by the reference operator was measured. In particular, the time needed to prepare the data for one booking and the time needed by the system to complete a transaction (to commit). Therefore, the cycle observed consisted of the entire execution of the Oracle application used to book. The time of a cycle simulated, in such a way should represent the real time needed by a typical operator to book exactly in the same conditions of a real booking-center. In this way the values measured for all the machines could be directly compared with an absolute threshold of acceptability. For this first test, all the operators involved, both that reference one and the others which keep busy the system, had to book using disjoined resources. Oracle RDBMS resolves the cases of concurrent access to a shared resource, reserving the use of such data only to one of the requiring tasks at a time. This fact guarantees to avoid situations of internal deadlock, but penalises terminal users. Particular expedients can be used to reduce the number of such a cases to the bare necessary, for example, bounding the critical area to the single record and not to an entire table. For the characteristics of the application used during the benchmark, the critical area which could be cause of queueing of user requests, is identified by the records of the daily agenda of the laboratory which exams were booked. Hence, it was imposed that all the operators, included that one of reference, did not collide on the same record, trying, for example, to book for the same day. There are two reasons for such a care. The first is that, as it was observed during the software experimentation, the cases of collision are in the reality rather unfrequent; during the benchmarks, on the contrary, because of the simplified and repeated booking operations, there was an high probability of such collisions. The second reason is to avoid that the Oracle applications made the user wait, with a warning message, that another user unlock a record; in such cases the first user does have the possibility to proceed, but he must give up to update the locked record. As the working conditions should be real as well as possible, and the workload should be uniform during all the benchmarks, a disjoined resource approach seemed justified.

b) in the second section, at least six measures of time of commit were collected. This is the case in which all the operators forced the system to complete a transaction exactly at the same time. Unlikely the section a), only one type of measure was taken here; indeed, it was asked the ten operators to input the data of one booking and to wait for a conventional signal that would be given when all the operators were ready to commit the transaction. In this way, the conditions in which the system had to operate were forced to be the worst, that is, all the terminals requested the system to process their data and to write them in the physical memory. This tests, as it will been shown in the next paragraph, are interesting especially in the case of different hardware configurations, that is, with different number of disks and with a different location of the database files on the disks. During this tests, two instants were timed: the first and the last completion among the ten transactions;

c) the third and last section of the benchmarks was formed by two executions of a batch SQL routine. This routine contained a series of six sequences of select from the same tables used by the operators, two requests for update of a table, one cycle composed by a creation, insertion and deletion of data, and, finally, dropping of a new table. The operations were all studied so as to take up the systems; for example, a type of select request needed all the records of a large table to be counted. During this section no particular control was used to avoid collision and the consequent waiting state of some operators caused by the batch routine. The processing time of the single operations was measured.

4. The analysis of the results

How it was quoted in the preceding paragraph, it is not possible to show in this paper the results of the tests making a comparison among the different computers. The results shown here are some of the real values computed, but they are linked not to the real machines or firms but to a set of fictitious machines M_1, M_2 and so on. In spite of this, some interesting considerations about the meaning of these values connected to the model used through the tests are discussed.

The Tables 1 and 2 summarise some average values computed after the tests for four machines. The first table refers to a first offer for an USL of average dimension, while the second table, computed with the same conditions of test, presents values lower in the average since the supplying firms tried to satisfy the needs of an USL of larger dimension.

Machine	Cycle			Commit			Batch	
	before	**total**	**diff**	**last**	**first**	1st select	**update**	**insert**
M_1	43.77	61.88	18.11	3.84	5.2	6.37	39.64	5.06
M_2	19.21	30.78	11.57	3.15	4.77	14.41	50.10	19.83
M_3	20.12	30.5	10.37	3.28	4.36	11.87	19.3	7.28
M_4	25	42.54	17.54	3.76	4	7.76	27.82	6.95

Table 1: *Mean values for the first offer*

Machine	Cycle			Commit			Batch	
	before	**total**	**diff**	**last**	**first**	1st select	**update**	**insert**
M_1	29.33	33.44	4.11	2.69	3.92	2.92	51.62	3.35
M_3	20.12	30.05	10.37	3.28	4.36	11.87	19.3	7.28
M_4	26.25	34.25	8	2.16	2.11	13.74	27.64	10.40
M_5	34	39	5	2.46	3	7.52	30.61	5.65

Table 2: *Mean values for the second offer*

Each table is composed by three sets of columns, labeled as "Cycle", "Commit" and "Batch", that are linked to the three sections of the benchmarks performed. The set "Cycle" shows the data collected from the first type of test; the column "before" is computed as a mean of the time employed to input the data of a booking (in the reality, asking people for it or input it from a badge card and reading it from a prescription, during the benchmark section reading it from a text prepared before), measured during at least eight executions of the booking Oracle application, made by the operator taken as reference. The column "total" reports the total time of completion of a booking, that is the column "before" plus the time needed to commit the transaction, while the column "diff" shows exactly the mean time of commit for the first type of test. The set "Commit" presents two columns which report respectively the mean last commit time and the mean first commit time measured during the second section of test. It is to be noted that the column "last" is computed as mean of the time of the tests divided by the number of terminals involved. Finally, the last set of columns, labeled with "Batch", refers to the time of three batch SQL commands used during the third section. These commands were found more meaningful than the others in order to discriminate among the machines. They are a query time, "1st select", and two time of "update" and "insert" in which a physical writing on the mass memory device is required. All the values presented in the tables are in seconds.

The first consideration that can be pointed out, derives from the computation of the number of saturation terminals for the four machines. Using the equation (2) for this compute, the first assumption made was the identification of a parametric value for the thinking-time; the values measured which better approach that time are those reported in the column "before". These values do not represent exactly what is generally indicated as thinking-time in the sense of the theory presented in the second paragraph; indeed, each "before" time is made by a set of thinking-time and service-time, that is, typing data, for example the registration number of a person, and retrieving other information from the database, as the address of such a person. Since the operation that more charges the system is a commitment of a transaction, the "before" column represents the average time needed to be ready for committing, that is, the reverse of the average rate of commitment requests. However, the values given for the thinking-time are not taken from this column, but a fixed length of time (precisely, 40 seconds) is used, since it is a better approximation of the behaviour of an operator in the real case of bookings, as it was verified during the period of experimentation. The values for the service-time required to compute the number of saturation terminals is taken from the column "last" of the set "Commit". Indeed, this measure represents the time in which all the requests of commit, each made from one terminal at the same instant, are served, divided by the number of terminals (requests). This situation is the worst case possible, because all the operators require to the system to execute a cycle of writing on the physical memory; for this reason, the values in the column "last" are considered a good approximation of the service-time.

The results obtained with this computations are reported in the Table 3, where the two columns are relative to the two configuration offered by the supplying firms. It can be noted from this table that the conditions in which the benchmarks were made (ten terminals) are beyond the saturation threshold computed.

Saturation Terminals		
Machine	1^{st} offer	2^{nd} offer
M_1	11.4	15.8
M_2	13.6	=
M_3	13.1	13.1
M_4	11.6	19.5
M_5	=	17.8

Table 3: *The number of saturation terminals*

For sake of precision, to make repeatedly the same booking, as in the case of the benchmarks, does not represent exactly a typical workload for a system; Oracle RDBMS, in fact, is able to manage requests about the same data (or area of data) better than a set of random-distributed requests. This fact is another reason to use a parameter value for thinking time which is a probably more real one, but this problem did not worry us since the same conditions of tests were forced on all the systems.

The second consideration that could be made is relative to the different configurations tested. The behaviour of an Oracle application or command derives from the amount of processor time and physical memory accesses required. Oracle uses a two-steps strategy to commit transactions in order to maintain consistence of data; each insertion, deletion or update is written on a "before image" database located on the physical memory, and then, if and only if the total transaction succeeds, the new data are copied on the real database. This characteristic behaviour must be considered in order to explain the different mean values computed for those machines, i.e. M_1 and M_4, which in the two offers presented two different configurations.

So far, the systems involved in the benchmarks were considered from an external point of views, that is, the point of view of the end user. In order to understand in which way the transactions are processed in the particular case of different configurations, it is necessary to consider a lower level, that is, that one of the units contained inside the processing module of Figure 1. The machine M_1 in the first offer has a single-processor unit, while in the second one has a two-processors unit. It is clear how the performances of this machines could be changed. An higher processing power in the second case increased the capability of serving tasks and, therefore, the mean waiting time in case of concurrent commit decreased of about 30 percent. The performances obtained form the machine M_4, on the other hand, have a more complex cause. The difference for this machine in the two offers is the number of disks as well as the structure of the Oracle files on such disks; these files, which correspond to the above-mentioned database (DB) and before image (BI), are composed by contiguous physical blocks. In both the cases the firms were adviced to use row-devices to install the DB and the BI on the disk(s) of their computers. This care should improve the performance of Oracle RDBMS of 2 percent. Moreover, the machine M_4 presented in the two cases the

following configurations:

1) disk A with the system and user directories, disk B with DB and BI;

2) disk A with the system and user directories, disk B with the DB, disk C with the BI.

All the disks had the same characteristics, i.e. the same access speed.

Each time that an operator commits, Oracle forwards to the host system a request of writing the changed data from the BI into the DB. The time needed to serve the request is due to the physical retrieval of the data in the BI and to the following writing into the DB blocks. In the first case the system have to move the heads of the disk B forward, from the BI physical blocks to the DB ones, and backward in the opposite way, every time it have to copy a physical record. In the second case, the same operation is easier; indeed, the heads of the disk B, as well as those of the disk C, have to move through physical sectors of disk that are closer than the first case, since each disk contains only one database file composed by contiguous physical blocks. The performance improvement observed for the second case is about 40 percent and it is quite higher than the improvement due to the two-processors configuration of M_1; the values of the column "last" of the Tables 1 and 2, in fact, refer to a case in which the main workload were composed by a lot of commit requests.

5. Conclusions

In this paper the characteristics of the benchmarks made to select two computer configurations were presented. The tender was made by the Regione Liguria (Italy) and the environment of use of the machines was a booking-centers of two USL. The tests discussed here were designed to measure the performances of different machines in a situation as closer as possible to the real context of use. For this reason, the workload of the systems was simulated asking ten booking-point operators to execute their usual job using a subset of the software P.A.S.S. (Automatic Sanitary Service Booking). In order to estimate the saturation point of the machines, a mathematical model proposed by Kleinrock [1] was used; with this model the expected upper bound for the number of terminals that each machines could support maintain the same performances was computed.

Some considerations were made about the different results observed for two machines which had a different configuration for the two offers. It was pointed out the way in which Oracle RDBMS requests the host system to process a transaction and how the performance of this operation cold be improved not only through an obvious increase of processing power, but especially with an suitable layout of the Oracle files on the host disks.

References

[Bri73a] H. Brinch, *Operating System Principles*, Prentice Hall, New Jersey, 1973.

[Kle76a] L. Kleinrock, *Queueing Systems, vol.2: Computer Applications*, Wiley – Interscience, New York, 1976.

A Transaction Monitor for SINIX

Heike von Lützau-Hohlbein

GBI
Gesellschaft Beratender
Informatiker mbH
Schieggstr. 9 D
D-8000 München 71

ABSTRACT

The use of transaction monitors is widespread in the mainframe arena. Information retrieval-, booking-, warehouse control-, stock control- or personnel administration systems only work efficiently due to their use. Taking into account that the commercial EDP market is becoming more and more penetrated by UNIX systems, there is a rising demand for DB/DC applications. It is in this context that transaction oriented program to program communication gains importance in both homogeneous and heterogeneous computer networks including those containing UNIX systems.

The problems of the implementation of transaction monitors in UNIX systems in general and known implementations in particular are discussed.

As an example UTM (SINIX), Universal Transaction Monitor for SINIX (SINIX is the SIEMENS derivative of UNIX), is presented in more detail. In particular its interfaces to the user, interfaces to the system and how it is embedded in the SINIX system.

Connectivity in a heterogeneous environment is explained together with what types of partner can be addressed and how. It will be shown how UTM(SINIX) can help to integrate SINIX computers and their users into distributed DB/DC applications, how mainframes can be accessed and their power used.

1. Introduction

With UNIX gaining a larger share of the market in commercial areas the demand for system support of transaction oriented type of application processing is growing. The situation is more or less the same as we had in the mainframe area 20 years ago.

With a growing number of end users using a fixed set of programs processing predetermined tasks and using a specific database, the system support then available was not adequate. The application programrs needed more than half their time to implement solutions to problems which should have been available in such environments, taking into account the so called ACID attributes of a transaction where A stands for atomicity, C for consistency, I for isolation and D for durability.

Atomicity means that all changes in relevant data that the transaction causes will either be made completely or not at all.

Consistency means that the global database used by the application, which can be changed by several users simultaneously, is always in a logically consistent state.

Isolation means that changes to the global database made by a not yet terminated transaction only effect this transaction but not others currently running. Only after successful completion are the changes visible to other transactions.

Durability means that changes to the global database made by a transaction are not destroyable by hardware or software failures. When such failures occur, recovery must be reliable and fast.

If we look at UNIX systems in commercial areas which are really used in production environments, for example information retrieval-, booking-, warehouse control-, stock control- or personnel administration systems and if we are old enough to remember or well informed in computer history, we recognize the same situation.

The demand for transaction monitors is there, but seems not to be satisfied as yet.

2. Charateristics of Transaction Monitors

Most operating systems support, what we call "Teilnehmerbetrieb", but support for the so called "Teilhaberbetrieb" is much less prevalent.

To characterize "Teilnehmerbetrieb": Many users use the system in a such way that they have their own programs and own database, they run their own applications, the application resources belong to the user (Timesharing).

To characterize "Teilhaberbetrieb": Many users use the same set of programs, gathered in applications, which use one database; application resources like files and memory areas belong to the application, the users share the resources (Resource sharing).

In an environment like this, a transaction monitor is a must.

Other characteristics of transaction monitor applications are:

- Typically the user at the terminal is not an EDP expert but an expert in his own field, he is interested in doing his predetermined jobs with his well known formats, he is typically not interested in the configuration of his application.
- A unique transaction is built out of changes to the monitor internal data and the data of the database controlled by a DB system with ACID attributes.
- A task consists of one or more dialog steps which build one or more transactions, which have to be terminated within seconds. If there is a failure in between, hardware or software, the user must be able to resume at a predetermined point in the application.
- The set of programs which form the application are independent of the configuration and have an interface to the monitor that is easy to learn and handle by the application programr or end user.
- The application must be available around the clock; in case of failure recovery times must be very short.

3. Transaction Monitors in UNIX Systems

In the mainframe area it is remarkable that many of the more than 30000 instances of transaction monitor applications are running under general purpose operating systems. There are only a few dedicated operating systems, which achieve remarkable response times and throughput, whereas for the majority the services of a general purpose operating system are sufficient.

What is missing in UNIX systems based on System V Release 2 and onwards?

Weak points of UNIX which are normally stated:

- a filing system which is optimized for a number of small files. This is to be considered together with the DB system used.
- the use of a file system cache which does not give the user a guarantee of the ACID characteristics. In the X/OPEN portability guide synchronous writing of blocks to disc is defined and is indeed implemented in some systems, so that this point is no longer relevant.
- signals.

Something which is clearly missing for the implementation of a transaction monitor is a sort of event managing facility, where from the application point of view all relevant events for the application can be queued.

Another major issue is the lack of standards for a user interface. The situation is getting better with the activities of standardisation groups like OSI/TP and especially X/OPEN, the XTP group together with the activities in the DB area. At least there is clarification about the main components in this environment which are a transaction manager and resource managers and their interaction with each other.

With regard to user interfaces these are still in discussion.

Taking the UNIX system as a base there are a variety of possible solutions. One is the transaction monitor built at Carnegie Mellon University called CAMELOT (the Carnegie Mellon Low Overhead Transaction Facility). CAMELOT is based on MACH, which is functionally a UNIX kernel and is compatible to UNIX 4.3 BSD. CAMELOT is built on the client-server model, with both clients and servers implemented as MACH tasks. In particular, long-lived data objects are contained within data server tasks, which execute operations in response to remote procedure calls (RPCs). RPCs are issued by applications or other data servers located at either local or remote nodes. A transaction may include calls to any number of servers. Transactions may also be nested. The different tasks of the transaction monitor are separated. MACH divides UNIX processes into two categories – namely tasks and threads. A task provides the environment required for processing, which may contain several threads, which are independent and may run simultaneously. Interprocess communication within the MACH operating system can be spread over distributed systems and network.

4. UTM (SINIX): Universal Transaction Monitor for SINIX

4.1. Development

Another example is what SIEMENS did with its development of UTM (SINIX). Three years ago when development started the situation was even worse because of the absence of standards. The only standard application interface to transaction monitors was and is still the German DIN standard, DIN 66265 (KDCS), which is functionally comparable to IBM's IMS/DC interface. In comparison to the most widely used transaction monitor, IBM's CICS, it has advantages with regard to recovery and restart. In the SIEMENS mainframe area with its operating system BS2000 SIEMENS offers UTM (Universal Transaction Monitor) which has been installed more than 3000 times. This monitor incorporates the KDCS interface as a subset and is embedded in BS2000 with very limited interfaces to the operating system.

Architectually UTM consists of a set of homogeneous processes which contain the UTM system code and the application programs. UTM takes care of messages coming from end users and addressing specific application programs, starting these and then routing the answer back to the end user. UTM takes care of logging, resetting in error cases, all what is necessary for transaction processing. UTM is generated and configured offline and can be modified by means of administration commands dynamically.

This monitor was ported into the other main SIEMENS operating system SINIX. The advantages are obvious:

- existing know-how, with both system developers and customers
- common user interfaces
- short development time
- future development on a common source base.

The only disadvantage is probably that the concept is not "brand new", UTM is nearly ten years old, but as stated before, no standards are available which could serve as guidelines for a new development.

UTM is written in SPL, a SIEMENS internal system programming language for BS2000, a subset of PL1 with extensions relevant to a systems programming environment. A tool was written to convert SPL modules into C modules. After mapping the structure of BS2000 processes into the SINIX system and simulation of the relevant BS2000 system interfaces UTM (SINIX) came into being.

4.2. Processes

The monitor and its architecture has been ported together with its homogeneous processes (= utmwork processes), but to complete the environment other processes are needed to do dedicated work.

The application is started by an overall control process which creates the utmwork processes. These processes have a common wait point with a common message queue. When one of them is awakened when a message is to be processed, this message is received from the terminal via a so called dialog terminal process which serves only one terminal. This process waits for the response at its specific wait point.

Printing service is done by a printer process serving on the one side UTM and on the other the SINIX spool system.

The remote partners are connected via a network process, serving on the one side UTM and on the other the transport system.

Timer service, a known weak point of UNIX with only one outstanding timer request per process possible, is implemented in a dedicated timer process, controlling all timers which UTM sets.

4.3. System Interfaces

For interprocess communication UTM uses semaphores for signalling and shared memory segments for exchanging data between UTM work processes and external processes, shared memory segments are also used between utmwork processes for applicationwide control.

In general UTM (SINIX) uses only recommended X/OPEN system calls.

UTM uses for connection of remote partners the ICMX transport interface, which is the ISO transport interface in SINIX. When XTI is available this interface will also be supported.

For applications' convenience there is a coupled format handling system. UTM has a standard interface to format handling systems inherited from the original BS2000 environment.

The interface to database systems is also standard so that a connecting DB system has to support this interface syntactically and semantically.

5. Transaction Monitors and Distributed Databases

Considering nowadays database systems in the UNIX environment the need for a system like UTM lies in the area where in an environment with different DB systems a controlled distributed transaction processing is required. Distributed DB systems are only capable of controlling transaction processing in a homogeneous world.

In the terminology of the model of the X/OPEN XTP group UTM is a transaction manager usable for controlling heterogeneous database systems.

6. Distributed Transaction Processing

In interconnecting heterogeneous DB/DC systems the possible partners of UTM (SINIX) are first of all UTM (BS2000) applications with SIEMENS databases like UDS or SESAM and in the IBM world CICS and IMS applications.

The protocol used for distributed transaction processing is SNA LU 6.1, which was selected because of its existence and support some years ago and because of the need to connect to IBM's transaction systems and to avoid gateways. The protocol support is unchanged in UTM (SINIX).

In the BS2000/IBM environment Distributed UTM is now installed more than 200 times and it is hoped that with the possibilities presented by distributed transaction processing the advantages discussed will prove even more advantageous.

An Approach to Reliability in Distributed Programming

Giandomenico Spezzano and Domenico Talia

CRAI
Località S. Stefano
87036 Rende (CS)
Italy
dot@crai.uucp

ABSTRACT

This paper proposes the use of a distributed concurrent language for the implementation of fault-tolerant distributed systems. In our approach, distributed software systems are composed of a set of cooperating processes, which communicate using the message passing model, and are placed on the various hosts of the distributed architecture. The interesting aspects of our approach are presented in terms of modularity, concurrency, portability and reliability. This approach is bound to the use of a high-level concurrent language in contrast to the traditional approaches for the implementation of the reliable distributed systems.

1. Introduction

The transaction concept was born in the area of database management systems. Distributed computing systems are an area in which the transaction concept was later utilized. In it a transaction is seen as basic operation to implement fault-tolerant applications, namely as a programming construct to support operations on persistent abstract data types. A system that supports transactions (defined as a sequence of operations on shared data types) must manage concurrent accesses on them by guarantying the following properties:

- *atomicity*: either all or none of transaction's operations are executed;
- *serializability*: if several transactions execute concurrently, they affect the shared data as if they were performed serially;
- *permanence*: if a transaction completes successfully, its results will never be lost.

A distributed computing system is a collection of nodes or sites connected by means of a communication network without shared memory. In this environment, transactions are facilities for writing programs which are to be executed on the network's nodes, by guaranteeing:

1. *Location transparency*: the program operates on data as if they were all in the same node;
2. *Concurrency transparency*: the system executes many transactions concurrently, but each one of them performs as if it was alone in execution;
3. *Replication transparency*: although the data are replicated in several network nodes the program uses them as if there were a single copy;
4. *Fault transparency*: either all of the transaction's operations are executed or none, if a failure has occurred; after the transaction execution its results will survive hardware or software faults.

Presently several implementations of transaction systems exist in different areas: programming languages, database management systems, and distributed operating systems. They differ in the required functionality, the models used, the objects manipulated, the granularity of parallelism, and the degree of transparency for the programmer. The most known of the distributed systems area are: *Argus* [Bai84a], *TABS* [Spe85a], *ISIS* [Bir86a], and *Locus* [Mue83a]. All of these systems support fault-tolerant distributed transactions on shared data types.

To implement a transaction system in a distributed environment it is necessary to install at each node a transaction support (*transaction kernel*). The *transaction kernel* is the basic component that provides primitives for supporting the transactions and the shared data types (*objects*) on which they operate [Spe83a]. The collection of transaction kernels located on each node of the distributed system that cooperate for transaction support is named the *distributed transaction kernel.*

In this paper we neither propose a new transaction system, nor a new programming language which provides transaction mechanisms. We propose the use of a concurrent language for the development of a *distributed transaction kernel*, and in particular the language that we have implemented. Actually, it is our opinion that the fundamental features of a transaction kernel like *concurrency, reliability* and *efficiency* can be exploited in the best way if a high-level concurrent language is used for its implementation instead of low-level languages with extensive use of system calls. This is the way in which the transaction systems mentioned above are implemented.

In our approach the *transaction kernel* is composed of a set of processes in which each process implements a single function of the system. They perform in a concurrent way, exploiting their inherent parallelism.

In the remainder of the paper, the principal characteristics of the concurrent language and the advantages that its use offers with respect to the traditional approaches (section 2) are described. Section 3 describes one example of cooperating processes that are written by our language, and constitute an elementary *transaction kernel.*

2. The language based approach

This section is divided into two subsections, in the first the concurrent language utilized to discuss the proposal will be described. The second subsection discusses some of the advantages that a concurrent language can offer with respect to the traditional approaches. Although we utilize our language, it is important to underline that in our opinion any concurrent language which explicitly expresses communications, control of nondeterminism, and process structuring is suitable for a robust implementation of a transaction kernel.

2.1. Language outline

The language used to discuss our proposal is the NERECO system language [Spe87a], it is a message-passing language based on the CSP model [Hoa78a] and in particular on the ECSP language [Bai84a]. The NERECO system has been implemented by us on a network of Sun workstations with UNIX 4.2BSD [Lef83a]. It is a prototype of an environment for the development of distributed concurrent programs. The principal features of the language are described below.

In the NERECO system, a distributed program is a set of processes cooperating through message passing that are located on one or more computers. The proposed environment provides a methodology for modular and robust structuring of distributed programs by:

i) using unidirectional typed channels,

ii) expressing communication forms either by point to point (*rendez-vous*) or by diffusion (*broadcast* and *multicast*),

iii) controlling nondeterminism in communications,

iv) handling, in a simple and flexible way, error conditions by detection, confinement and recovering.

The language of NERECO has been implemented by adding a set of concurrent mechanisms, with well-formed syntax and semantics, to a sequential language, extending its static development tools to deal with the concurrent part. At the moment NERECO is based on Pascal, CHILL and C language.

Inside each process there are the declarations of the process itself and its partners, as follows:

self <*process_id*> : <*process_type_id*> ;

partners <*process_id*>,...,<*process_id*> : <*process_type_id*>;

Like the ECSP language, the processes cooperate through communication channels using input/output commands. Channels can be symmetric or asymmetric, but generally they are asynchronous. The syntax of the declaration of an asymmetric synchronous static channel inside a receiver process is:

chan from(<*process_id*>,.....,<*process_id*>) **type** <*constr_id*> (<*msg_type*>);

The *<constr_id>* is the type constructor with *<msg_type>* constituting the type of message transmitted on the channel. Channels can be dynamic since in this case the name of the partner process is a variable of *processname* type.

To allow dynamic channel management, two constructs are defined: **connect**(*<proc_var>, <process_id>)* (to assign a value and the communication rights) and **detach**(*<proc_var>)* (to assign the undefined value and to revoke whatever communication right).

Communications are implemented by the i/o commands of **send** and **receive**. The **send** construct can have a symmetric or asymmetric form:

send (*<process_id>, <constr_id>(<msg_var>)*);

send (**all of** (*<process_id_list>*), *<constr_id>* (*<msg_var>*));

in the first case, only a partner exists, in the second there is a set of partners, defined by a list (*send multicast*) or by a process type identifier (*send broadcast*). The syntax of the **receive** statement is:

receive(*<process_id>,<constr_id>(<msg_var>)*);

receive (*<proc_var>*: **any of** (*<process_id_list>*),*<constr_id>(<msg_var>)*);

in the first form, there is only one sender, in the second there is a set of senders, but only one of them delivers the message.

The nondeterministic constructs are similar to those provided by CSP, namely the **repetitive** and **alternative** commands with input guards and priority.

Additionally, the **terminate** construct allows a process to terminate itself at any time and informs all of the process' partners of the termination.

The language offers fault treatment policies to handle the communication failures caused by a *partner termination, channel disconnection*, or *physical communication media fault*. Failures can be handled by the **onfail, onterm, onprot** clauses. They make it possible to execute some recovery actions (*forward recovery*) when a failure occurs, as shown in Figure 1. If the first **send** fails, because of partner termination, a different process is connected (Sec_server), and the execution can continue.

These facilities are especially useful in a distributed environment where there are many processing element cooperating with each other. In this environment, when a process or a processor fails, the user can handle this failure on-line and can avoid propagation to other processes or processors.

```
.....
connect (Proc_name, First_server);
send (Proc_name,exec(param))
    onterm
    connect (Proc_name, Sec_server);
     send (Proc_name,exec(param));
    endrec;
receive (Proc_name,result);
.....
```

Figure 1: *Example of* onterm *use*

2.2. Features and advantages

Low-level languages with a large use of system calls are generally used for the implementation of transaction systems. This traditional approach as been viewed as efficient, but it presents many negative aspects in terms of lack of modularity, poor reliability, low parallelism and so on. These problems have greater effect in a distributed environment where these factors have more importance than in a centralized one.

The approach that we propose consists of utilizing a concurrent distributed language like that presented in the previous section for the implementation of reliable distributed systems like the *distributed transaction kernel*. A similar proposal in a different field (i.e., data base management systems) has been done by Moss [Mos86a].

According to our approach the *transaction kernel* is implemented as a virtual machine by a set of concurrent processes each of which carries out one of the expected functionalities. For instance, one process deals with the protection of data, another deals with the control of concurrency and another executes on data the operations specified by the transactions. The data themselves can be kept in the manager processes and their manipulation can perform concurrently. The virtual machine provides:

1. *Location transparency* is provided by the concurrent language, because it offers uniform communication mechanisms both between local and between remote processes.
2. *Concurrency transparency* is achieved through the process realizing the functionality of concurrency control, which interacts with the transaction processes that are executed in parallel. The process controlling the concurrency on data can be also divided in a set of processes working in parallel.
3. *Replication transparency* can be simply realized through the contemporary execution of more copies of the same process that can be addressed through a type.
4. *Fault transparency* can be achieved using the mechanisms offered by the language to handle explicitly the faults that can occur.

Another important characteristic offered by the language is the explicit management of the nondeterminism. It in fact can be utilized for increasing the parallelism in the execution of the transactions.

In addition to the advantages listed till now, it is necessary to add some concepts typical of the software engineering that are present in our approach and show advantages already known. They are:

- *data abstraction*: implemented by keeping the data in the manager processes;
- *modularity*: implemented by isolating particular functionalities inside the single processes;
- *portability*: supported by the virtual machine which is defined by the run-time support of the language and allows the re-hosting of the system without rewriting the whole *transaction kernel*;
- *reliability*: provided by the language's strong checks at compile-time and the static checks of consistency among the processes.

A factor carrying out an important role is the efficiency. Clearly the efficiency of the *transaction kernel* implemented by a concurrent language depends on the language efficiency. Overall system efficiency could obtain benefits deriving from the possibility offered by the language, which allows the exploitation of a high parallelism in the execution. At last, the benefits which the programmer can achieve in the development and testing of the system using the tools offered by the development environment of the language must not be undervalued with respect to the deficiencies of the traditional approach.

3. An example

To illustrate some characteristics of the language in support of the implementation of a transaction kernel, the following presents a processing transaction system implemented by a collection of cooperating processes (Figure 2).

The proposed solution does not pretend to solve the distributed transaction problems, or to be an exhaustive example of a transaction kernel. The aim is to describe how the concurrent language can be utilized to develop this kind of distributed systems. The system makes it possible to service transaction requests that are statically defined, entered by users from terminals and operating on shared objects.

Each *Trans_proc* encapsulates one of the transactions. The *User_interface* processes communicate with the *Trans_proc* processes to ask for the execution of a transaction. In the *transaction kernel* case the transactions will be defined by the user and the processes *User_interface* will be substituted by the process that encapsulate the user's transaction. Objects are encapsulated into processes called *Object_M*, which serve the requests generated by *Trans_proc* by means of more *Executor* processes. For brevity, Figure 3 presentes only the code of process *Trans_proc*.

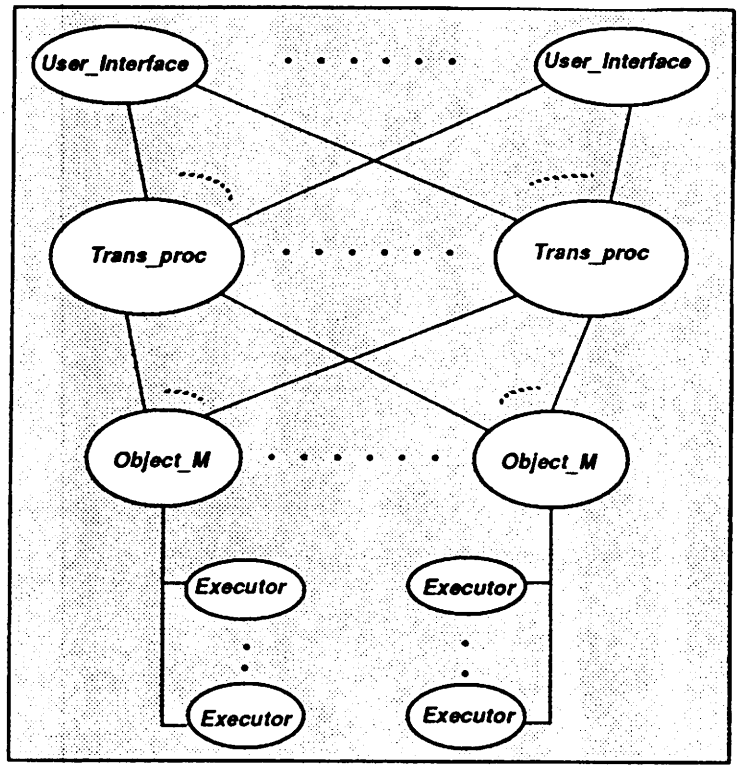


Figure 2: *Process network.*

The fundamental goal of this system is to exploit the parallelism among the running transactions when they operate on different objects or when they operate on the same object at the same time. The parallelism on the same object is obtained utilizing more *Executor* processes which execute operations concurrently on behalf of different transactions on object copies.

The generic *Trans_proc TRi* is activated by the receipt of a request from a *User_interface UIj* and tries to execute the transaction, then returns the result (*abort*/*commit*) to *UIj*, which if the result is an *abort* can redo the request. In the most general case the requests can arrive both from local processes and from remote *Trans_proc*. To implement this we only have to declare the channels and to add the remote process identifiers into the **receive** statement. Once a request is received from a *UIj* the *TRi* sends a message to all of the managers of the objects to allocate an *Executor* process to which it will require the operations execution, if necessary forcing the transaction's abort, and with it will establish the two-phase commit protocol. Therefore each *Executor* is statically devoted to an object and dynamically allocated to a transaction.

At any arbitrary time many transactions can be executed on each object concurrently. To guarantee the *serializability*, a commit of one transaction causes the abort of the others. The concurrency on an object is provided so that a transaction which aborts does not preclude the object use in the other transactions.

4. Conclusion

The paper presented an approach for the implementation of a *distributed transaction kernel*. This approach is based on the use of a distributed concurrent language which allows the exploitation of communication management, parallelism, data abstraction and modularization.

We have briefly described the syntax of the concurrent language implemented by us on a network of Sun workstations. After having discussed the advantages of this approach as regards to the traditional approaches that have been utilized until now, we have presented an example of a processing transactions system written in the concurrent language.

```
program TR1 ();

< Declaration part >

self trans1 trans_process;
partners usr_int1, usr_int2 : user_interface;
partners obj_m1, obj_m3 : object_manager;
partners exec1, exec2, exec3, exec7, exec8, exec9 : executor;

< channel declaration part >

begin
 while true do
  receive(U:any of (usr_int1,usr_int2),request);
  st := send( all of (obj_m1,obj_m3),acquire());
  alt
   receive(E : any of (exec1,exec2,exec3),change());
   docl
    connect(EXE_O1,E);
    receive(E : any of (exec7,exec8,exec9),change());
    connect(EXE_O2,E);
   endcl;
  % receive(E : any of (exec7,exec8,exec9),change());
   docl
      connect(EXE_O2,E);
    receive(E : any of (exec1,exec2,exec3),change());
    connect(EXE_O1,E);
   endcl;
  endalt;
  code_op:=1;
  send(EXE_O1,code_op)
   onprot
   < ABORT >
   endrec;
  receive(EXE_O1,result1);
  if result1.status =0
   then < ABORT>;
  < operations for other EXECs >
        .....
  send( EXE_O2,ready_to_commit())
   onprot
   < ABORT >
   endrec;
  send( EXE_O1,ready_to_commit())
   onprot
   < ABORT >
   endrec;
  alt
   receive(EXE_O2, answer3);
   docl
     receive(EXE_O1, answer2);
   endcl;
  % receive(EXE_O1, answer2);
   docl
     receive(EXE_O2, answer3);
   endcl;
  endalt;
  if answer2 and answer3
   then
   begin
    <COMMIT >
    answer1.answer:=1;{committed}
    answer1.tid1:=request.tid2;
    send(U, answer1);
   end
  else
   begin
    <ABORT >
    answer1.answer:=0; {aborted}
    answer1.tid1:=request.tid2;
    send(U, answer1);
   end
 < end while >
end.
```

```
<ABORT > ::
begin
  send(EXE_O1,abort())
   onprot
    <SKIP>
   endrec;
  send(EXE_O2,abort()):
  onprot
     <SKIP>
  endrec;
  < go to next request >
end;
```

```
<COMMIT > ::
begin
   send(EXE_O1,commit());
   send(EXE_O2,commit());
end;
```

Figure 3: *The Trans_proc process code*

References

[Bai84a] F. Baiardi, L. Ricci, and M. Vanneschi, *Stating Checking of Interprocess Communication in ECSP*, 19, pp. 290–299, ACM Sigplan Notices, 1984.

[Bir86a] K. P. Birman, "ISIS : A System for Fault-Tolerant Distributed Computing," TR-86-744, Dep. of Comp. Sc. Cornell Univ., Ithaca, New York, April 1986.

[Hoa78a] C. A. R. Hoare, *Communicating Sequential Processes*, 21, pp. 666–677, Communications of the ACM, August 1978.

[Lef83a] S. J. Leffler, R. S. Fabry, and W. N. Joy, "A 4.2BSD Interprocess Communication Primer," in *UNIX Programmer's Manual Berkeley Software Distribution, Virtual VAX-11 Version*, vol. 2, Univ. of California, Berkeley, CA, August 1983.

[Mos86a] J. E. B. Moss, "Getting the Operating System Out of the Way," *IEEE Database Engineering*, vol. 9, no. 3, pp. 35–42, September 1986.

[Mue83a] E. T. Mueller, J. D. Moore, and G. J. Popek, "A Nested Transaction Mechanism for LOCUS," in *Proc. 9th ACM Symposium on Operating System Principles*, pp. 71–89, Bretton Woods, NH, October 1983.

[Spe83a] A. Z. Spector and P. Schwartz, "Transactions: A Construct for Reliable Distributed Computing," *Operating Systems Review*, vol. 17, no. 2, April 1983.

[Spe85a] A. Z. Spector, "The TABS Project," CMU-CS-85, Carniege Mellon University, 1985.

[Spe87a] G. Spezzano, D. Talia, and M. Vanneschi, "NERECO: An Environment for the Development of Distributed Software," in *EUUG Conference Proceedings*, pp. 153–167, Dublin, September 1987.

UNIX and Object Oriented Distributed Systems

Donal Daly
Vinny Cahill
Chris Horn

Distributed Systems Group,
Department of Computer Science,
Trinity College, Dublin 2,
Ireland.
daly@cs.tcd.ie
vjcahill@cs.tcd.ie
horn@cs.tcd.ie

ABSTRACT

UNIX is a well established system interface, as can be seen from the work of POSIX and X/Open. It has been gradually extended to support distribution and embrace concepts such as object orientation. Systems like Mach try to make the kernel smaller while providing increased support for distribution. Object oriented systems promise the potential for re-usable software, along with higher level data modelling. The Esprit COMANDOS project is supporting distribution and object orientation. It intends to provide an integrated platform for the development and online management of distributed applications. Placing a UNIX interface on top of such a distributed object orientated kernel is a possible approach to integrating UNIX and distributed object systems, which is explored in this paper. The motivation for supporting UNIX in an object oriented distributed environment is presented. We describe then, the main features of the COMANDOS kernel. Finally, an approach to supporting UNIX with an object oriented kernel is outlined. Such an approach would provide a migration path for existing UNIX users towards a fully object oriented system. It would also provide to UNIX users not interested in object orientation access to the increased functionality available in a distributed system.

1. Introduction

The aim of the Esprit project COMANDOS† (COnstruction and MANagement of Distributed Open Systems) is to provide an integrated platform supporting the development and on-line management of distributed applications. The COMANDOS platform will provide a range of services including support for location transparent distributed processing (but with the possibility of overriding location transparency if required); heavyweight and lightweight concurrency and synchronisation; an extensible, distributed data management system; support for transaction management and replication, and finally a range of tools to monitor and administer the distributed environment [Com89a]. COMANDOS will also provide a programming language to facilitate the programming of distributed data management applications in the COMANDOS environment [Com89a]. The COMANDOS platform is intended to be vendor independent and to run on a range of systems from different manufacturers.

The platform is made up of a kernel and a set of system services which run as applications above the kernel. The COMANDOS kernel provides the minimum functionality required by the applications running in the distributed environment, and low level support for the system services such as data management and management tools. The COMANDOS kernel can potentially be provided as a native kernel running on bare hardware or as a guest layer on top of an existing operating system.

† This work is partially funded by the Esprit programme under contract 2071

COMANDOS has chosen to follow the object oriented paradigm as a unifying technology to integrate programming language, data management system and kernel. As such the platform provides its user with an interface defined in terms of objects whose behaviour is specified by the application developer. The COMANDOS kernel provides the necessary low level support for the use of objects.

Three prototype implementations of the COMANDOS kernel have already been undertaken. In Lisbon Inesc have built a single user version of the kernel as a native kernel on top of PC/AT compatible machines. In Dublin TCD is implementing the kernel as a native kernel (known as *OISIN*‡) on Digital micro VAX II workstations and NS32000 based machines, and also as a guest layer on top of Ultrix. Finally in Grenoble, Bull and LGI, in conjunction with the French nationally sponsored Guide project, have implemented a kernel interface on top of SPIX (Bull's UNIX V2.2 kernel with BSD extensions) and SunOS. A demonstration of some of these results was given at the annual Esprit week in Brussels in November 1988.

1.1. COMANDOS in the real world!

An important aspect of the COMANDOS platform is that use of its facilities is not limited to programmers using the COMANDOS language. Indeed a range of languages are expected to be used to program COMANDOS applications: both object oriented – C++ [Str86a] or Eiffel [Mey88a] for example – or non object oriented such as C or Modula-2.

Such languages will be supported by the provision of appropriate libraries giving them access to the functionality of the COMANDOS environment including the ability to make use of application components (possibly encapsulated as COMANDOS objects) written in other languages. Clearly, it is important that existing applications be available in the COMANDOS environment.

Existing operating system interfaces should be provided in the COMANDOS environment. In particular, it is seen as important that the UNIX system interface be available to COMANDOS application developers. The importance of UNIX stems from its widespread acceptance as a de facto vendor independent system interface. Current standardisation [Bol89a] efforts will serve to increase the importance of UNIX as a standard systems interface.

1.2. COMANDOS and UNIX

Our aim is to provide application developers with a single interface consisting of both the well established UNIX system interface and the interface provided by the COMANDOS platform.

From the COMANDOS viewpoint, this approach provides COMANDOS application developers with a wealth of existing applications and tools. On the other hand, UNIX users will be provided with an integrated set of services, allowing them to access distributed and persistent data in a uniform way as well as providing support for fault tolerance. Moreover the COMANDOS platform will provide excellent runtime support for object oriented programming which is rapidly gaining widespread acceptance.

1.3. Implementation approach

Two approaches to the integration of COMANDOS and UNIX are possible. Either the COMANDOS kernel is provided as a layer on top of the UNIX kernel (e.g. in UNIX user mode) or the UNIX interface is provided on top of the COMANDOS kernel. We believe that both approaches are feasible and are actively exploring both possibilities.

Making a decision as to which approach is most appropriate depends largely on what performance is attainable in each case. Is UNIX a good basis for building a distributed object oriented system? Can *OISIN* support a performant implementation of the standard UNIX system primitives?

Previous experience with the implementation of distributed and object oriented systems on top of UNIX would suggest that the performance of a COMANDOS system implemented above UNIX cannot be as good as that achieved by a native implementation of the COMANDOS kernel tuned to the needs of the object oriented distributed environment [Dec88a, Alm85a]. However such an implementation will be more portable and consequently likely to be more widely used if, as we believe, performance is acceptable.

‡ OISIN was a warrior in Irish folklore who was given the gift of eternal youth.

Previous experience with implementing UNIX on top of another kernel suggests that this is a difficult task [Ras81a]. However we believe that we can provide a performant implementation of the UNIX interface on top of COMANDOS while providing at the same time good performance for object oriented and distributed working.

COMANDOS has drawn inspiration from a number of research projects elsewhere, and is attempting to integrate their respective approaches into a uniform platform. The Apollo Domain system [Lea83a] is similar to the COMANDOS platform in that it provides a distributed store, and storage objects which can be accessed independently of their location. Researchers at Purdue [Dew89a] have implemented a pre-processor and a series of associated libraries to support objects using UNIX. Chorus provides a low level kernel which supports both heavyweight and lightweight processes [Roz88a]. It also has a basic IPC mechanism which is transparently extensible over a network. On top of this minimal distributed kernel the UNIX V interface is provided. The Mach [Acc86a] project has developed a UNIX compatible system with low level support for distribution. Mach also asserts to supporting a style of object based programming.

In the remainder of this paper we try to motivate this latter approach by showing how UNIX concepts map on to the facilities provided by the COMANDOS kernel. Section two describes the interface provided by the COMANDOS kernel – in particular the *OISIN* implementation. Section three shows how a UNIX emulation layer (*ROISIN* †) can be provided on top of *OISIN*. Section four describes the current state of our work and outlines future implementation plans. Finally, section five contains a summary of conclusions from our work to date.

2. The OISIN kernel

At Trinity College, the Distributed Systems Group has undertaken a native implementation of the COMANDOS kernel on both NS32000 based workstations and Digital Microvax II machines. The main aim of this implementation of the COMANDOS kernel, locally known as *OISIN*, was to experiment with low level techniques to provide good performance for an object oriented system on conventional demand-paged hardware. In the following paragraphs we briefly review the main features of the COMANDOS kernel interface and its implementation in *OISIN*. Further details can be found in [Mar88a, Dec89a].

2.1. Object Model

In COMANDOS all *objects* are described by *types* which define any public fields of the object and the operations available to manipulate the object's state. The public fields of an object may be read and written by implicitly defined read and write operations, and provide the basis for associative retrieval of objects (not discussed further in this paper). All objects are instances of *implementations*. An implementation describes the private fields of its instances and the operations defined on them. Note that the relationship between types and implementations is one to many in both directions: an implementation may implement many types and a type may be implemented by many implementations.

Each object has a low level location independent identifier (LLI) which serves to identify and locate the object within the distributed and persistent system. Such "LLIs" are not intended to be used directly by application programmers or end users, but may be stored within the instance data of an object to refer to a constituent or related object. An object reference consists of the objects LLI and a location hint, discussed later in section 2.3.

Most objects are passive (active objects – jobs and activities – will be discussed later in section 2.2) and execute only as a result of an operation invocation on the object. Moreover, objects may be of any size from a few bytes (objects also contain a fixed size header) to several megabytes. The size of an object may be changed dynamically during the lifetime of the object.

Once created an object persists until no more references to it exist in the system or the object is removed by aging [Com87aa]. Thus all objects are potentially long lived and their lifetimes are not necessarily bound by, for example, the duration of the creating job.

† ROISIN was a beautiful princess in Irish folklore

2.2. Computational Model

The only truly active entities in a COMANDOS system are *jobs* and *activities*. A job may be considered as a distributed heavyweight process. A job consists of a set of *contexts* (address spaces) and a set of activities. At each node visited by the job, the job has at least one, but possibly several contexts into which the objects being used by the job at that node are mapped. The contexts of a job are private to that job and not shared with any other job. Activities are distributed lightweight threads of control i.e. analogous to lightweight processes but with the possibility of executing in several contexts at the same node or in several different nodes at different times. Activities of the same job share the contexts of the job at each node that they visit during their lifetime. In order for an activity to access any object, that object must be mapped into some context belonging to the job.

Normally activities execute by invoking operations on passive objects synchronously. If the target object of such an operation invocation is mapped in the same address space as the invoking object then the operation is carried out immediately, although possibly returning an *exception* in the event of a failure. If the target object is not mapped in the current context, then an *object fault* is said to have occurred which the kernel must resolve. The kernel must first locate the target object using the reference for the object passed to it. The object could be located in a context of another job at the current node, mapped at a remote node or stored (locally or remotely) in the distributed storage system. If mapped locally the object will normally be shared between the contexts of the two jobs seeking to use it. Otherwise, the object fault may be resolved either by fetching the object from a remote node and mapping it into a context of the invoker, or by carrying out a remote invocation on the object. When an activity invokes an operation on an object at a node which it has not previously visited it is said to have *diffused* to that node. The decision as to whether to fetch the target object to the invoking node or to carry out a remote invocation is seen as a policy decision, which could be based on load balancing criteria, security or heterogeneity considerations.

Invocation is, by default, location transparent (i.e. the invoker does not need to be aware of the location of the target object) although, it is also possible to override location transparency either by specifying where an invocation is to take place, or by explicitly positioning an object.

Although, operation invocations on objects are synchronous, an asynchronous invocation is possible if a new activity is started to carry it out. Furthermore *OISIN* supports the idea of a *channel* – analogous to a UNIX pipe – into which the results of one operation may be deposited to be picked up as the parameters to another operation. Note that the entities passed along the channel are typed invocation frames as opposed to raw bytes in the case of a UNIX pipe.

A set of operations are available to control the operation of jobs and activities, including the ability to suspend or resume a job/activity; kill a job or activity and finally to retrieve status information for a job or activity. The latter includes retrieving the results of the invocation which the job/activity was created to carry out if it has terminated.

Exceptions have already been mentioned and can potentially be raised by any invocation. Such exceptions are propagated up the activity's invocation stack (possibly across node boundaries) until an appropriate handler is located. It is also possible for one activity to asynchronously raise an exception in another activity. Arrival of such an exception causes the current invocation in the receiving activity to be suspended, while an appropriate handler is located and executed. Such asynchronous exceptions are similar to UNIX signals but are arbitrarily typed.

2.3. Clustering

In the *OISIN* implementation of the COMANDOS kernel the object space is split into a set of *clusters*. In *OISIN* every object is entirely contained within a cluster. A cluster is implemented as a set of contiguous virtual memory pages somewhat analogous to a segment in some other systems. Each cluster contains a group of objects which may vary dynamically either by the creation of new objects in the cluster or by garbage collection of objects in the cluster, or migration of existing objects between clusters.

In *OISIN* clusters are the units mapping into and sharing between contexts. Each context contains a set of clusters which may change dynamically as clusters are mapped and ummapped. When an object is required by a job the entire cluster containing the object is mapped in the appropriate context although not necessarily loaded i.e. pages of the cluster may be faulted from the storage system (or from another context on the node if the cluster is already mapped locally) on demand. Note that the mapping of a cluster appears transparent to application code i.e. there is no explicit primitive to fetch a cluster.

Clusters allow the *OISIN* kernel to exploit the locality of reference between related objects by allowing them to be grouped in the same cluster. Thus the number of object faults may be reduced. Moreover invocations of objects within the same cluster are much cheaper than those of objects in a different cluster. Note however that it is not a function of the kernel to decide which objects should be grouped together – rather this is seen as a higher level management function.

A cluster is normally shared between all the contexts using it and written back to the storage system when no longer required. However, it is possible to specify that a particular cluster is to be *copy-on-write* so that every job using it gets its own private image of the cluster. It is also possible to specify that a particular cluster is to be *immutable* meaning that it is not written back to disk even if changed. A cluster can be marked as *position dependent* so that it is always mapped at a designated virtual address or as meaning that once mapped it cannot be moved within the context.

In *OISIN* a reference for an object includes a hint for the cluster in which it is currently contained.

2.4. Security

In *OISIN*, security is based on the notion of a *domain*, which is a set of clusters that have a common ownership. Whenever an attempt is made to execute an operation on an object from a different domain, the kernel is invoked and the user's right to execute the specified operation checked by consulting an access list associated with the object. Thus every object accessible from outside of its domain has an associated access list. If the user has the right to execute the specified operation on the target object then its cluster will be mapped on behalf of the invoking job. However the cluster will not be mapped in the original (calling) context but in a context created for the job for objects of the target domain. Thus each job has one context per node per domain accessed i.e. objects of different domains are never mapped in the same context, and thus all inter-domain calls result in object faults.

This implementation of domains prevents interference between objects from different owners being used by a particular job. For example if a job executing on behalf of user A is using objects owned both by user B and user C to which A has access – in particular implementation (code) objects – it is guaranteed that C's objects cannot read or maliciously alter B's objects or vice versa. Likewise neither B nor C can damage A objects. However, user A must still trust B and C to provide the specified interface correctly.

2.5. Kernel Components

The *OISIN* kernel is itself structured as a set of co-operating objects. The Virtual Object Memory (VOM) component is essentially responsible for the handling of object faults including locating the target object; making the decision as to how and where to resolve the object fault; checking access permissions etc. The VOM also implements contexts at each node and co-operates closely with the storage subsystem to handle mapping of clusters and page faults.

The Storage Subsystem (SS) provides for long-term storage of clusters. The SS also supports replication of clusters for increased availability. Finally, the SS includes low level support for data management services.

The Activity Manager (AM) implements job and activities and the operations on each. It also provides a remote invocation service in co-operation with the Communications System and low level synchronisation facilities.

The Communications System (CS) provides the underlying network communications support. The main service provided by the CS is the inter-kernel message (IKM) service. The IKM currently runs above raw ethernet and it is intended will eventually run above ISO class 4 connectionless transport and UDP/IP. The CS will also provide direct access to the underlying ISO and IP protocol stacks.

3. Supporting UNIX with an Object Oriented Kernel

3.1. Introduction

The UNIX emulation *(ROISIN)* is based on the Berkeley 4.3 distribution [Lef89a]. We term it an emulation because we do not intend to implement another kernel above ours. What we do instead is use the facilities provided by *OISIN* to provide a standard UNIX interface. It is our intent that it will be system call compatible, albeit with a few exceptions. We chose Berkeley UNIX because we are most familiar with this version of UNIX and have access to its source code. Current development work being carried out at Berkeley is of interest to us also. This includes providing a POSIX compatible UNIX and supporting OSI [McK85a]. We felt that if our design proved feasible, we could incorporate these new features quite easily.

ROISIN is structured much like a normal UNIX system. Part of *ROISIN* will actually reside in the same address space as the UNIX process. When the user executes a system call, control is transferred to this code. This may then result in object invocations on other *OISIN* objects. These would include *ROISIN* specific objects such as the Process Manager or File Descriptor Manager or a Name Service. *ROISIN* is very much integrated in an *OISIN* environment. It provides an application programmer with an integrated environment of a UNIX system alongside a distributed object oriented system.

3.2. UNIX Processes

A Process Manager is used to maintain our "proc" structure. The Process Manager is implemented as a regular *OISIN* object. Naturally, the process manager must support atomic updates to the entries in the proc structure. Using *OISIN* low level synchronisation primitives (semaphores) will allow us to provide this atomic support. The Process Manager will implement the following system calls:-

getdtablesize, getgid, getegid, getgroups, getpgrp, getpid, getppid, getpriority, setpriority, setgroups, setpgrp, setregid, setreuid, fstat.

3.2.1. Fork, Exec, Wait and Exit

UNIX processes are naturally laid out in memory in the same manner as under a native UNIX system. Each UNIX process is represented as a COMANDOS job with only one activity. A UNIX a.out file is represented as an *OISIN* object. This object is placed in a cluster on its own. On invoking the exec system call, the cluster which contains the a.out is simply mapped into VOM at a fixed address. The exec call will then setup the data and stack areas. The arguments and environment are then copied to the top of the stack. The process is now ready to run. A COMANDOS job is created whose initial activity will call main().

The a.out cluster has a number of important attributes. Firstly it is immutable (as previously described), it is also position dependent so that the a.out (actually the text area) is mapped in VOM at the correct address. The cluster is also of fixed size. If we were to increase the size of the cluster (so as to increase the size of the heap or stack), this could result in the cluster being remapped to another area of VOM. The effect of this would be that C pointer references, would be invalidated and the process would inevitably crash. In *OISIN*, regular objects utilise location independent references so that they are unaffected by a cluster being moved.

The implementation of the sbrk, and also how the UNIX process may grow its runtime stack, is thus also limited. Using a fixed sized cluster to hold our UNIX process is not as bad as it would seem. Firstly, we will create the cluster large enough to allow reasonable growth. This will only involve allocation of virtual memory, as the whole cluster need not be loaded in. The cluster is faulted as required. We also allow the user to supply us with hints, using a mechanism similar to the csh(1) limit command so that we can determine the size of the cluster.

In supporting the fork system call we also make the cluster copy-on-write. Using copy-on-write provides us with an efficient means of implementing fork, and should significantly improve performance over traditional UNIX implementations.

With our proposed implementation of fork and exec, we have a number of interesting means to exploit distribution of UNIX processes. We could implement a scheme so that this distribution would be transparent to the user. We could decide to execute the process on the least loaded node, or use some more sophisticated load balancing criterion to determine which node to execute the process. Another approach would be to extend the existing system calls so that the programmer has the ability to choose on which node his process should execute. For us to implement these extensions should be quite trivial, and they serve as a clear example of the powerful underlying distributed system which is available to *ROISIN* and the application programmer.

Another area currently of interest is migrating of running processes [Man88a, Alo88a, Hun88a]. Its exploitation is being considered in the areas of load balancing and reliability. To achieve this you must be able to take a snapshot of the process stack and data areas. Remember that in our case these are contained, along with our text image, in one cluster. We hope in the future to be able to provide this facility.

The wait and wait3 system calls are implemented using the facilities provided by the *OISIN* Activity Manager. These system calls will use a semaphore maintained by the Process Manager for each process. When the wait system call is executed the process will block on a semaphore, if there are no zombie children of the parent process. The exit system call will be invoked on the Process Manager, who will signal this semaphore and tidy up the "proc" structure. It will also send a SIGCHLD signal to the parent process. This exit system call will invoke on the File Descriptor Manager so that any open file descriptors

belonging to that process are cleared.

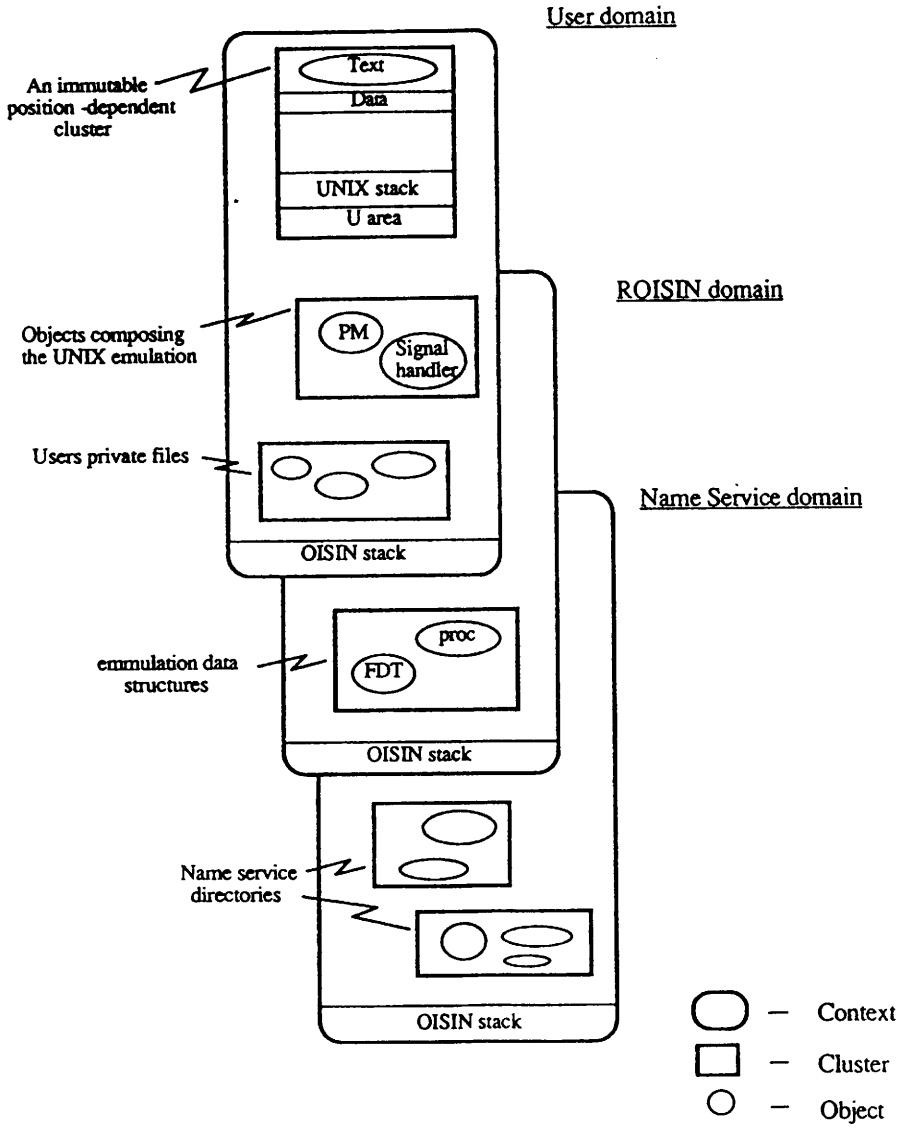


Figure 1: *A possible layout for a UNIX process in ROISIN*

3.2.2. Use of Domains

Figure 1 show a possible layout of a UNIX process in *ROISIN*. We have three domains, for reasons of security. In the user domain we will keep user objects and those emulation objects which can operate within that domain. These would include the *ROISIN* signal handler and Process Manager which would implement such calls like getpid or getppid. In the *ROISIN* domain we will keeps objects which are shared by all *ROISIN* processes. In this way we can protect them. The "proc" structure object in this domain would implement system calls like setpgrp or setpriority. When a user executes one of these system calls it will involve a cross context call. Our third domain is the Name Service domain, This domain will hold directory objects of the Name Service. If we allowed these to be mapped in the user domain, it may be possible for the user to alter inode information contained within these directory objects. We keep this third domain so that the *OISIN* Name service can access our directory objects without crossing into the *ROISIN* domain. It will also protect the Name Service from *ROISIN*. When a user reads a file which he owns it will be mapped into the user domain, however if he has only group read rights to the file it would be mapped, for example into his domain as immutable.

3.2.3. Signals

Signal handling facilities in 4.3BSD differ radically from the facilities found in other version of UNIX [Lef89a]. Some of the main differences include:-

1 Signal handlers are permanently installed with a single sigvec.

2 The signal currently being received is automatically masked from delivery while the application signal handler is invoked.

3 An alternative stack can be used for delivering signals (sigstack).

4 System calls which are interrupted by the receipt of a signal can be automatically restarted whenever possible and reasonable.

This provided an interesting problem to solve. One solution is to use a parallel activity to the main UNIX one. This activity would block on a semaphore until a UNIX signal was sent to its peer. It could then suspend the UNIX activity, and execute the stipulated handler, or kill both itself and the UNIX activity if the signal was not caught. We have chosen instead to use the facilities provided by the *OISIN* exception model. We would receive asynchronous exceptions which would include an optional parameter. These exceptions are handled using a special *ROISIN* signal handler. Its function is to process the exception so as to maintain the semantics of signal handling in 4.3BSD. It would in turn invoke a user supplied signal handler if one was specified. Before calling the user supplied signal handler it will create a stack frame on the users stack, so that executing the signal handler will appear as if the process had called a user level function, and when it returns from the signal handler normal execution will be resumed. We are confident using the above mechanism that signals can reliably be handled. The *ROISIN* signal handler will also implement the default actions on the receipt of signals.

The kill system call would first ask the *ROISIN* Process Manager for the activity reference for the process it wished to send a signal. It would then invoke an operation on the AM and request to which activity an exception is to be raised, passing in the exception the specified signal number. In the implementation of the exec system call an exception record is setup, so that the *ROISIN* signal handler will be invoked whenever a UNIX activity receives an exception.

3.3. UNIX Filesystem

In the COMANDOS architecture a system service, called the Name Service, is defined [Com87ba]. This Name Service is implemented by a distributed set of directory objects which provide associations between object references and symbolic names. The directory objects themselves have names, which implement a graph structure, somewhat like the UNIX filesystem hierarchy. The Name Service provides an equivalent to the UNIX internal "namei" routine. Presently we have this Name Service implemented under *OISIN*.

ROISIN exploits the underlying facilities provided by an *OISIN* environment, to obtain a transparently distributed directory hierarchy. It also needs to change the structure of the hierarchy to make it conform to the UNIX model, that is there is only one root and all other directories are below this. In doing this it will choose by default the root directory object of the local node of the user. In performing directory lookups, valid names will include *grainne:/usr2/comandos/kernel* which will inform the Lookup operation to use the directory which is located on Grainne. We could however just simply invoke a Link operation on the Name Service to link this directory to the local file system hierarchy, and use this instead.

ROISIN when possible, will attempt to place UNIX files (objects) into the optimum cluster. To *ROISIN* and to an application programmer, dealing with local or remote files is identical. It is of course possible to find out where files are stored and also to decide where to store your files.

Due to the means by which objects are mapped in Virtual Memory, *ROISIN* is able to provide support for memory mapped files [Tev87a]. In fact, this is the means by which files are accessed. Using this mechanism we intend in the future to implement shared libraries under *ROISIN*.

We plan to extend the *ROISIN* Name Service so that the directory objects will contain the inode information for the objects about which the directory knows. Much like the UNIX *namei* routine we will maintain a cache of recent lookups, so as to maximise performance. From profiling studies it has been found that nearly one-quarter of the time spent in the kernel is spent in namei [McK85a]. It is critical therefore that we have an efficient lookup operation. We expect also to derive increased performance from clustering. All Name Service directory objects will be contained in one or more clusters. Because of the way clusters are mapped into virtual memory we should have efficient access to directory objects.

3.3.1. File System Calls

Our extended Name Service will support the following system calls through object invocations on directory objects

access, chdir, chmod, chown, chroot, link, mkdir, mknod, rmdir, stat, lstat , unlink , utimes

We do not consider the mount and umount system calls as having any functionality under our extended Name Service. A possible implementation could extend/limit the Name Service view of the Name Space.

Every UNIX File is defined as a special type called UnixFile. This type will implement a number of operations which will support the following system calls:

creat, close, open, lseek, read, readv, truncate, write, writev

There will be a direct mapping between these system calls and the operations on type UnixFile. For example, the implementation of the open system call would first invoke a lookup operation on a suitable directory object so as the LLI for the UNIX file can be obtained. It would then invoke on the File Descriptor manager so that a file descriptor can be setup. Further system calls like read or write will cause the file to be mapped into virtual memory. UNIX files (objects) are not loaded at a fixed address like a.out files. The read and write system calls will therefore interact with the *OISIN* runtime so as to find the current address of the UNIX file. This will then be added to the current file offset so that they will position themselves at the correct position within the UNIX file. The close system call will invoke on the File Descriptor manager so as to release the file descriptor for the file. When the process exits, and the *OISIN* job terminates, the file will be written back to secondary storage. The chroot system call will be implemented as an operation on the Name Service.

Currently the *ROISIN* file system would only manage files of type UnixFile. This could be extended in the future to support any *OISIN* object via a registration mechanism. The *OISIN* Name Service can deal with any object, where as our extended version will only deal with objects where a directory object contains inode information for them. Our initial implementation we will thus have two name services, but we see no reason why these cannot be merged in the long term.

3.4. UNIX I/O System

The UNIX I/O system will be supported as a layer above the *OISIN* I/O system. We are considering implementing this layer using channels. The *OISIN* I/O system would pass or receive a byte stream to or from a channel. We would have one channel for each serial device available to *ROISIN*. This channel would implement simple I/O processing, and support the following system calls:

ioctl, read, readv, write, writev

More sophisticated I/O can be accomplished by pushing another channel onto the I/O stream.

We will implement a File Descriptor Manager to maintain information about active inodes within *ROISIN*. It will also support the semantics of the "open file table" as under native UNIX implementations. The File Descriptor is implemented as an OISIN object and will support atomic updates to file descriptors which it manages. As such it will support the following system calls

fcntl, flock, dup, dup2, ioctl, select

The File Descriptor Manager will also maintain the current file offset into each file so as to support the lseek system call. It will interact with the Process Manager so that complete and consistent information can be kept for each UNIX process. It will also maintain the information relating to file descriptors contained in the u.area of a process address space. We plan to use semaphores to implement the flock system call. The AM provides us with a mechanism to allow us to see if we would block if we tried to do a wait on a semaphore. Using this feature we are able to implement advisory locking as supported by flock. We could also quite easily implement enforced locking using the same mechanism.

OISIN channels support typed data, but it is still possible for us to use channels to pass low level byte streams. With this in mind, we for see few problems in supporting UNIX pipes, with channels.

We will support block oriented devices (like disks) by providing a channel interface much as we support character oriented devices (serial lines or tape units). This channel would implement a direct interface to the Device Manager of the *OISIN* I/O subsystem.

The *OISIN* kernel provides a rich networking environment. The Communications Subsystem will support both the OSI and IP protocol stacks. To the application programmer we supply a socket interface, so like the rest of *ROISIN* the programmer needs not have any knowledge of the inner workings. Presently we see channels as the appropriate *OISIN* facility to provide this socket interface.

3.5. UNIX Protection Model

The COMANDOS Security Architecture shares features with the UNIX Protection Model, along with some extensions of its own [Hor89a]. The UID and GID of *OISIN* can act as the UID and GID of *ROISIN* without change. Effective UID's GID's can be implemented in a similar way as well.

The three categories of users in UNIX can be supported as follows:- the user and group categories will map onto the user and group types in *OISIN* with the exception that groups may not be nested (In *OISIN* it is possible to have nested groups). The implementation of the world category is also supported by *OISIN*.

The superuser privilege is a difficult concept to support in an "distributed object world". It will be supported initially by ensuring that root has the control access right over every object under the control of *ROISIN*. A more long term solution needs to be found, and probably lies in splitting the superusers powers. Work is already being done in this field [Hec87a]. Superuser privilege will not extend across machines and a check will be made to ensure this.

The main problem in implementing the UNIX Protection Model is to give adequate protection to files, while at the same time ensuring efficient access to them. As has been stated the finest granularity of protection in *OISIN* is clusters. This is due to the way objects are mapped in virtual memory. So in the worst case under *OISIN* each UNIX object would have to be placed in its own cluster. What is required is a method or rule to decide if more than one or two objects can be stored in the same cluster so that the UNIX protection model is not broken. A possible rule is as follows:

> One or more UNIX objects may be placed in the same cluster if and only if their protection modes are the same otherwise, they must be placed in a cluster where their protection are the same.

Thus an extra check needs to be placed in the UNIX system calls which deals with protection. These calls would include:

> *open, creat, mkdir, chmod, chown.*

ROISIN maintains a table of the protection attributes associated with each cluster under her control. This table will be implemented as an *OISIN* object on which the system calls that effect security will invoke. These system calls will then be returned a suitable cluster to place the object, if it needs to be migrated as the result of a protection mode change. Using this mechanism when multiple ownership, group or protection changes occur as a result of chmod(1), chown(1), chgrp(1) commands, objects can still be clustered together, albeit in a different cluster than before.

For the most part it should be able to cluster UNIX objects. Under a users home directory users files can be stored together in the one or more clusters. Similarly files under group directories can also be kept together if they have the same protection. It should be noted that for each individual protection mode a separate cluster is needed. This is not seen as a problem, as in most UNIX systems a small subset of protection modes are used.

In an attempt to organise UNIX files, so that we can utilise clustering fully, we introduce a more effective mechanism for dealing with groups. The association between a group-id and a file is made optional rather than requiring that each file have an associated group [Car88a]. Using this mechanism the grouping of objects into clusters is simplified.

4. Current Status and Unresolved Issues

To prove the feasibility of our design we are currently implementing *ROISIN* in C++. Any utilities making use of */dev/kmem* or */vmunix* will have to be re-written. Apart from these we expect that we will only need to compile utility programs and link in the new system calls in order to make them available to *ROISIN*.

There are some system calls which we have not considered for various reasons. We do not implement ptrace or profil, as *OISIN* does not currently provide any support for this type of operation. Because of the way in which we have implemented processes, it would be possible to access these active entities as files. This has already been accomplished in Version 8 UNIX [Kil84a]. Implementing this mechanism should be straightforward. With this in place, we could implement a debugger for distributed applications using these

files to open or close processes and to read and write their code segments. This has been implemented also in Version 8 UNIX [Car86a].

We also do not implement sync and fsync directly as there is no equivalent to them under *OISIN*. We would implement them in the future, by extending the un-map cluster call in *OISIN* to maintain the integrity of disk data structures. We feel however given the facilities of the SS, which includes replication, that *OISIN* provides a consistent, reliable environment already to *ROISIN*. Other system calls which we currently do not support are those that deal with file system quotas and resource limits. We do make use of resources limits, so that we can decide on cluster sizes for UNIX processes, but they are not strictly enforced (i.e. we do not limit the cpu time of a process).

5. Conclusions

The main aim of the COMANDOS project is to provide runtime and linguistic support for the development and on-line management of distributed applications. To date the project has defined the interface to be provided by the platform and has demonstrated several prototype implementations running on a range of machines from different vendors.

We believe that integration of the COMANDOS platform with existing system interfaces is necessary both to provide a means for existing applications to be used alongside emerging distributed and/or persistent applications and to provide a migration path for existing systems users towards COMANDOS. Because of its widespread acceptance, support for the UNIX system interface is considered crucial.

COMANDOS will bring to UNIX users an integrated set of facilities for programming distributed, persistent applications using an object oriented framework.

We have identified two alternative approaches to the intgeration of UNIX and COMANDOS: either by building the COMANDOS platform on top of an existing UNIX kernel; or by providing the UNIX system interface above the COMANDOS platform. Both approaches have their merits and demerits – a basic tradeoff is between portability and performance - however we believe that both approaches are feasible and are actively investigating both.

In this paper we have tried to show the suitability of the COMANDOS platform to host the UNIX interface. Previously reported work has discussed the construction of a COMANDOS like system on top of UNIX [Dec88a].

6. Acknowledgements

The authors wish to acknowledge the contribution to this work made by all the members of the TCD COMANDOS kernel team: Edward Finn, Andre Kramer; Annrai O'Toole, Gradimir Starovic, and John Slattery. We also wish to acknowlegde the input from all the members of the TCD COMANDOS team: Alexis Donnelly, Sean Baker, Neville Harris, Damien Lynch, Faris Naji, Ahmed El-Habbash, Mark Sheppard, Brendan Tangney, Bridget Walsh and Iseult White.

References

[Acc86a] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for UNIX Development," in *USENIX Summer Conference*, pp. 93–112, June 9-13, 1986.

[Dec88a] D. Decouchant et al., "Implementation of an Object-Oriented Distributed System Architecture on Unix," *EUUG Autumn Conference*, pp. 181–193, October 3-7, 1988.

[Alm85a] G. Almes et al., *The Eden System: A Technical Review*, IEEE Software Engineering, Vol SE-11, No 1, January 1985.

[Mar88a] J. A. Marques et al., "Implementing the COMANDOS Architecture," *Proc. ESPRIT Conference 1988 - Putting the technology to use*, North Holland, November 1988.

[Alo88a] Rafael Alonso and Kriton Kyrimis, "A Process Migration Implementation for a Unix System," *USENIX Winter Conference*, pp. 365–372, February 9-12, 1988.

[Bol89a] Cornelia Boldyreff, "UNIX Standardisation: An Overview," *EUUG Spring Conference*, pp. 151–156, April 3-7, 1989.

[Car86a] T. A. Cargill, "Pi – A Distributed Debugger," *EUUG Conference*, pp. 137-141, September 22-24, 1986.

[Car88a] Scott D. Carson, "Using Groups Effectively In Berkeley Unix," *USENIX Winter Conference*, pp. 171-173, February 9-12, 1988.

[Dec89a] D. Decouchant, M. Riveill, C. Horn, and E. Finn, "Experience With Implementing and Using an Object Oriented, Distributed System," *To be presented at the Workshop on Experiences with Distributed and Multiprocessor Systems*, Sponsored by USENIX, SERC, ACM and IEEE-CS, October, 1989.

[Dew89a] Prasun Dewan and Eric Vasilik, "Supporting Objects in a Conventional Operating System," *USENIX Winter Conference*, pp. 273–285, January 30 – February 3, 1989.

[Hec87a] M. S. Hecht, M. E. Carson, C. S. Chandersekaran, R. S. Chapman, L. J. Dotterrer, V. D. Gligor, W. D. Jiang, A. Johri, G.L. Luckenbaugh, and N. Vasudevan, "UNIX without the Superuser," in *USENIX Summer Conference*, pp. 243–256, Phoenix, AZ, June 8-12, 1987.

[Hor89a] Chris Horn, Edward Finn, and Andre Kramer, "Security Facilities in the OISIN Kernel," *COMANDOS Project Working Paper*, May 16, 1989.

[Hun88a] Chad Hunter, "Process Cloning: A system for duplicating UNIX Processes," *USENIX Winter Conference*, pp. 373–379, February 9-12, 1988.

[Kil84a] Tom J. Killian, "Processes as Files," *USENIX Summer Conference*, pp. 203-207, June 12-15, 1984.

[Lea83a] P. Leach, P. Levine, B. Douros, J. Hamilton, D. Nelson, and B. Stumpf, *The Architecture of an Integrated Local Network*, pp. 843-857, IEEE Journal on Selected Areas in Communications, Vol SAC-1, No 5, November 1983.

[Lef89a] Samuel J. Leffler, Marshall Kirk Mc Kusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison Wesley, 1989.

[Man88a] K. I. Mandelberg and V. S. Sunderam, "Process Migration in UNIX Networks," *USENIX Winter Conference*, pp. 357–364, February 9-12, 1988.

[McK85a] Marshall Kirk McKusick, Samuel J. Leffler, Michael J. Karels, and Luis Felipe Cabrera, *CSRG Technical Report*, November 30, 1985.

[Mey88a] Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[Com87aa] ESPRIT COMANDOS Project, *Object Oriented Architecture*, September 1987.

[Com87ba] ESPRIT COMANDOS Project, *Kernel and System Service Functional Specifications*, September 1987.

[Com89a] ESPRIT COMANDOS Project, *Oscar Programming Language Reference Manual v1.1*, April 1989.

[Roz88a] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "CHORUS Distributed Operating Systems," in *Computing Systems*, vol. 1, pp. 305–370, USENIX, Fall 1988.

[Ras81a] R. Rashid and George Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *In Proc. 8th Symposium on Operating Systems Principles (ACM)*, pp. 64–75, December 1981.

[Str86a] Bjarne Stroustrup, *C++ Programming Language*, Addison Wesley, 1986.

[Tev87a] Avadis Tevanian, Jr., Richard F. Rashid, Michael W. Young, David B. Golub, Mary R. Thompson, William Bolosky, and Richard Sanzi, "A Unix Interface for Shared Memory and Memory Mapped Files Under Mach," *USENIX Summer Conference*, pp. 53–67, June 8-12, 1987.

XEiffel: An Object-oriented Graphical Library and an OPEN LOOK Based on it

Marco Menichetti

UniRel
Firenze
Italy

ABSTRACT

Object-oriented languages can be extended by user-defined classes libraries. In this paper I present XEiffel. It is a graphical extension of the object-oriented language Eiffel. It is based on the X Window system but adds to it some interesting object-oriented features. As XEiffel application I present an OPEN LOOK graphical interface toolkit. The whole work has been developed using SUN 3/60 workstations at UniRel.

1. Eiffel: an Object-oriented Language

1.1. Introduction

Eiffel is an object-oriented language, created in 1986 by Bertrand Meyer, and evolved in version 2.1 on July 1988. Most of you know some object-oriented languages or have heard about them and so I only list the particularity of Eiffel, without explaining the meaning of concepts related with object-orientedness.

1.2. Eiffel Concepts

Eiffel is an object-oriented environment for development of high-quality production software. Traditionally object-oriented languages are implemented by way of interpreters, as exemplified by small-talk; by contrast, Eiffel is a compiled language. Eiffel elements, common to other languages too, are: class, object, inheritance and polymorphism.

- A **class** is an user-defined type or, using a more formal definition, a class is the implementation of an abstract data type. In Eiffel a class can be "deferred". It means that it can only be used as ancestor, because some features in it are defined but not implemented. Such features must be written in its descendant classes. If you think to a class as a type, a deferred one can be thought as a frame on which we can build other types, i.e. other classes.
- An **object** is an instances of a class. In Eiffel the code can be written only in classes and so the creation of an object require a client-supplier relation between classes. It's usually said that an object belongs to a class for meaning a program entity is declared of an user-defined type.
- **Inheritance** is the mechanism which allow the incremental software construction. By inheritance a class can be defined as an extension or a restriction of others. Inheritance is the way by which object-oriented language increase reusability of software.

  Graphically inheritance is represented by an arrow meaning "inherit from". A class inheriting from another is said to be a "descendant" and the class from which it inherits is said "ancestor". Most of object-oriented languages support only single inheritance, by contrast in Eiffel the programmer can enjoy multiple and repeated inheritance.
- **Polymorphism** is the possibility for programs entities to refer to objects of more then one class. This means that if an entity is declared to be of type A, run-time it can also refers to an object of type B. In Eiffel polymorphism is controlled by inheritance and so the sentences above is correct only if the class B is a descendant of class A. Dynamic binding is related with polymorphism, in fact in a descendant class you can redefine same inherited feature, for example for increasing performances. Redefinition allows you that the same feature name refers to different actual feature depending on the type of object to which it is applied. The mechanism implementing this feature is called dynamic binding.

In Eiffel there are also some interesting feature, not related with the object-oriented paradigm, useful for increasing software robustness and correctness. As example I only cite assertion, automatic make and garbage collector.

2. X Window System

2.1. Overview

The X Window System is a network transparent window system that was designed at MIT and runs under many operating systems. The X Window System implements a server-client model. The server provides the basic windowing mechanism. It handles connection from clients, demultiplexes graphic requests onto screens and multiplexes input back to appropriate clients. The clients are programs which connect to the screen by some interprocess communication path. The X server and clients communicate by requests and events: the clients send requests to the X server and the X server sends events to the clients. The events may be either generated from devices or generated as side effects of clients request. Events are never sent to the client unless the client has specifically asked to be informed of that type of events. The set of event types a client requests relative to a window is described by using an "event mask". The elements handled under X are called resources. Resources can be:

- Windows
- Pixmaps
- Cursors
- Fonts
- Graphics Contexts
- Color Maps

These resources can be created and handled by the C functions of the Xlib library.

3. XEiffel

3.1. Introduction

Object-oriented language can be extended by user-written library of classes. XEiffel is an extension of Eiffel and it allow the Eiffel programmers using most of the X Window System facilities.

3.2. XEiffel Architecture

XEiffel is based on the X Window System, because it uses the Xlib functions to work out low level graphics, but in it there are also new aspects and so XEiffel is actually an object-oriented graphic environment. We can think to XEiffel as two layers software package. The lower layer is at a "low level" and I mean that classes composing it are near Xlib. The upper layer involves the events and events management. This layer, solving in an object-oriented way the events management problem, make event-driven programs much easier to built.

3.3. Low Level Classes

Classes at low level codes the X Window resources and encapsulate in them functions for creating and handling the resources. Examples of classes at low level are DISPLAY, FONT, G_CONTEST and DRAWABLE. The classes DISPLAY, G_CONTEXT and FONT are actually type but the class DRAWABLE, a deferred one, is used as ancestor for building, by inheritance, other classes.

Application programs before using the XEiffel's classes have to open a connection with the X server; this problem is solved creating an object of type DISPLAY. It's easy to guess that classes G_CONTEXT and FONT provide the programmer with the X resources graphics context and font. In the X jargon the word "drawable" means graphical support, on or off screen, i.e. a "drawable" can be a pixmap or a window.

The natural object-oriented way for modelling this subject is the building of a deferred class DRAWABLE and use it as ancestor for building the classes WINDOW and PIXMAP. The entity of type DRAWABLE can, run-time, be a pixmap or a window because of the dynamic binding.

We couldn't obtain directly the class WINDOW from DRAWABLE and so between them we defined GEN_WINDOW and WINDOW_X. The design of showed classes was not very difficult: we almost translated the X Window manual in Eiffel classes. The real sticky problem were events and events management.

3.4. Event-driven Programming and Event Management Problem

The event management problem is common to every event-driven program; we often run into interactive programs today and so the event management problem need a general solution. We can solve it in two different way: a traditional one and an object-oriented one.

Traditional Solution

The traditional solution is the great switch:

```
from
until
loop
    event:= display.next_event;
    if event.type = ButtonPress
    then
        .
        .
    elsif event.type = Expose
    then
        .
        .
    elsif event.type = ...
    then
        .
        .
    end;
end;
```

Listing 1

Referring to Listing 1, we get an event from the the event queue and, depending on its type we run the correct routine. This solution looks a good one; it seems clear and logically consistent, but it is not so: usually an application program use many windows and handles at least ten events on every window.

If we use 30 windows and mask ten events on every window, our switch will have 300 cases. Certainly our code will be complex, hence error-prone. However the worst consequence of using this simple-minded solution is that it bring us to non-reusable code and this is a great problem because the management of many events is common to most applications. We have an object-oriented language and, using it, we can find a better solution.

Object-Oriented Solution

The X server can send 33 different kinds of event. We built a class for each kind of event: some fields are common to every fields of event so we developed a class EVENT and then we specialized it by inheritance. Every object-oriented design require an abstraction phase before its implementation.

In the event-management problem we can see three sub-problems:

- window and event masking on them
- the routines to be run when an event is read, called from now on "process"
- a manager of the window and routines.

and we can translate in classes each of these issues.

The events and windows are modelled in the WINDOW_X_EVENT class. Process are put in the descendant of the deferred class EVENT_PROCESS and the manager is an object of the class EVENT_MANAGER.

I am not going to explain all implementation detail but I only show you the difference between the code of the traditional solution and the object-oriented one. Pay attention to Lisiting 2:

```
P1.creare (...)
P2.create (...)
    .
    .
Pn.create (...)
event_manager.create
event_manager_insert_event_process(P1)
event_manager_insert_event_process(P2)
    .
    .
event_manager_insert_event_process(Pn)
event_manager.mask;
event_manager.get_event;
```

Listing 2

In Listing 2 we have an initializing part; from *P1* to *Pn* are objects of classes descendent from EVENT_PROCESS and contain the code to be run in response to events. Listing 3 shows a class inheriting from EVENT_PROCESS: it knows the kind of event that must be handled, the window in which the event must be generated and how to process it.

```
class EXAMPLE_PROCESSOR export

    event, window, process

inherit

    EVENT PROCESS
    redefine
    event
feature

    event: TYPE_OF_EVENT;
    process: BOOLEAN is

    do

        code to be run when an event
        of type TYPE_OF_EVENT is
        generated on window

    end; -- process

end; -- class EXAMPLE_PROCESSOR
```

Listing 3

Referring again to the Listing 2, we have the set of instructions *event_manager.insert-process(Pi)*, using these statements we notify the event-manager the code to be run for responding to the X server events. The last two lines, *event_manager.mask* and *event_manager.get_event*, mask the correct events on the windows and begin reading the event queue.

About an object-oriented solution we can notice that it is very simple. We have only a sequence of statements, without control or conditional instructions. We have no critical instructions. Of course there are somewhere critical pads of code, but they are hidden and well framed into EVENT_PROCESS descendant classes. Closing this paragraph I list the necessary steps for building an event driven application using the XEiffel library.

1. Build, inheriting from EVENT_PROCESS, the classes containing the code for handling events.
2. Create object of type DISPLAY, WINDOW, EVENT_MANAGER and EVENT_PROCESS descendant.
3. Call the EVENT_MANAGER's routines *insert_event_process, mask and get event.*

This is neither depending on the type and number of managed events nor on the number of windows.

4. OPEN LOOK Toolkit

4.1. Introduction

Graphical user interfaces for workstations are difficult to built. To help programmers create such interfaces was born the concept of toolkit. A toolkit is set of pre-cooked modules that allow the easy building of standard graphical interfaces. OPEN_LOOK is the AT&T standard for graphical interfaces and it suggests the lay-out and the behavior of what should become the UNIXstandard interface.

From XEiffel to a Toolkit

XEiffel is a general purpose graphical library, so for developing from it our toolkit we used the inheritance mechanism for specializing the classes. As example we can cite the classes WINDOW and BASE_WINDOW: WINDOW is a general-purpose class, and it is available in XEiffel, a base window is a particularly kind of window, used in OPEN_LOOK G.U.I.. It has some banners, resize corner, a close mark etc. By inheritance we built the class BASE_WINDOW adding to WINDOW the necessary features. It worth to note that XEiffel as an intermediate layer between the toolkit and the X Window System, so we could for example replace the current version of the X Window System with a new one, or perhaps with another similar graphics system, without changing a statements in the toolkit.

On the other side at the moment it is difficult to understand which graphical user interfaces will become actually standard, and so we must be prepared to built a new toolkit (a Motif one?). We can do it at low cost specializing again the XEiffel classes.

At the end some consideration about object-orientedness and toolkit. In '70 Dijkstra noted that the more "goto" was used the more error-prone the program was, and so he killed the instruction "goto". Today we can say that not only "goto" are dangerous to every day programming, but also "loop", "if" and other control instructions so, walking on the Dijkstra road, we should avoid using them. Of course, somewhere must be written also control instructions, but as more as possible, they must be written in reusable classes.

A toolkit, in my opinion, is the tool for programming without explicit control instruction. Perhaps this could be the definition of toolkit. Of course, object oriented languages offer the best environment for building and using toolkit, as proved by our experience.

Acknowledgements

The XEiffel library and the OPEN LOOK toolkit has been developed by Dott. Marco Crescioli, Ing. Luciano Papini and myself. I would like to gratefully acknowledge Prof. Giovanni Soda of the Università degli, Studi di Firenze for his constructive criticism to this paper, MariaTeresa Bonarini and Sabrina Papucci for their good typing service.

Bibliography

B. Meyer, "Object-oriented Software Construction," *Prentice Hall*, 1988.

"OPEN LOOK Graphical User Interface Functional Specification," February 1989 (Revision 15, prerelease version).

Efficient implementation of low-level synchronization primitives in the UNIX-based GUIDE kernel.

D.Decouchant

Laboratoire de Genie Informatique

E.Paire, M.Riveill

Centre de Recherche BULL
Unite Mixte BULL-IMAG Systemes
Z.I. de Mayencin
2, Rue de Vignate
38610 Gieres, France
decoucha@imag.imag.fr
paire@imag.imag.fr
riveill@imag.imag.fr

ABSTRACT

When developing new and complex applications on top of UNIX system, implementors are usually faced with synchronization problems whose solution is not simple. This is especially true when such applications are in fact a new system level which defines a different model of synchronization. Standard mechanisms normally provided are simple, general but not efficient enough when heavily used. Synchronization implementation should be of low cost with respect to other system components, but this is not usually the case. This paper first summarizes the synchronization mechanism required by our object-oriented environment, then describes the implementation of our final solution, which was derived in several steps, and finally presents experience and performance measurements of different progressive improvements.

1. Introduction

The work described in this presentation is carried out in project GUIDE, a joint project of Laboratoire de Genie Informatique and Centre de Recherche BULL at Grenoble. This group is also part of the COMANDOS ESPRIT Project. Project GUIDE develops a specific implementation of the COMANDOS object-oriented architecture [Hor87a]. The first version of this implementation is based on UNIX and is described in [Dec88a].

A high-level synchronization mechanism has been defined to control access to shared objects [Dec88b]. This paper concentrates on the efficient implementation of this mechanism on UNIX.

2. Synchronization on Shared Objects in Guide

2.1. Object model

Let us briefly remind the main features of our object model.

- Objects are passive entities. An object is the association of a set of data (some of which are visible attributes, and the others internal variables), and a set of methods.
- Objects are typed. A type specifies attributes and signatures of methods applicable to the objects of that type.
- A class describes an implementation of a type. It specifies the data internal representation and the methods code. A type may be implemented by several different classes.

- Types and classes are organized in a hierarchy; the current version supports single inheritance.
- Objects are persistent (the lifetime of an object is independent of that of the program in which the object was created).
- Objects are internally named by references, which allow to locate an object system-wide. References may be embedded into objects, thus providing the means to build complex structures. These structures may be distributed, although an individual object is always located on a single node.
- Computation is organized into a two-level scheme: jobs and activities. A job may be viewed as a "distributed virtual machine": it may spread over several nodes and dynamically diffuse. A job defines a multi-node address space in which objects may be loaded on demand. Within a job, activities are defined as sequential threads of control. The execution of an activity is a sequence of (synchronous) method calls to objects.
- Shared objects are the only means of communication between activities, within a job or between different jobs.

Jobs and activities are special objects for which methods are provided by the operating system.

The motivation for the choice of a passive object model has been derived from the intended use of the system. We expect to use a large number of fairly small objects, and to build many composite structures. We therefore preferred to avoid the overhead of associating one or several processes to an object. However, the model still allows to implement "servers" (or "guardians") in the form of a job with multiple activities and shared objects used as interfaces. Such a server may even be distributed on several nodes.

The object model is supported by a language, which is described in [Kra89a] Within the language, support for concurrent activities within a job is provided by a `COBEGIN-COEND` construct, in which the "join" is controlled by a termination condition, which allows (for instance) to wait for the termination of all activities, of the first activity, etc.

2.2. Synchronization mechanism

Shared objects introduce the need of a synchronization mechanism. In accordance with the object model, we choose to associate this mechanism to objects, not as separate synchronization primitives within activities. Thus an object is entirely self-contained, including the specification of synchronization.

Since a class provides a model for the implementation of objects, it is also the natural place for the description of the synchronization. This is achieved by a `CONTROL` clause; if this clause is not present in a class description, the objects of the class have no synchronization constraints.

A `CONTROL` clause is essentially defined as a set of activation conditions (i.e. guards) associated to the methods of the object. If an activity calls a method of the object, the guard must be true in order for its execution to proceed. Guards are expressed in terms of the internal variables of the objects and of internal counters maintained by the system. These counters are defined as follows for each method m:

```
invoked(m)      # invocations of method m
started(m)      # non-blocked invocations of method m
completed(m)    # completed executions of method m
current(m)      # activities currently executing method m
pending(m)      # activities currently blocked on invocation of method m
```

These counters are not independent since

```
current(m) = started(m) - completed(m)
pending(m) = invoked(m) - started(m)
```

For example, the "readers-writers" synchronization scheme may be expressed as follows, if writers have priority:

```
CONTROL
    Write: (current(Write)=0) AND (current(Read)=0);
    Read : (current(Write)=0) AND (pending(Write)=0);
END
```

In this example, an activity which calls the `Write` or `Read` method must first evaluate the control clause associated with method. Each control clause returns a boolean value that indicates the possibility to execute the requested method. The `Write` method may be run only if the current number of `Write` method calls and `Read` method calls are both zero. The `Read` method may be run only if the current number of `Write` method call is zero, and the number of pending `Write` method is also zero. The `Read` control clause is not expressed is terms of `Read` counters, so several `Read` method calls may be performed in parallel.

3. Implementation of the Synchronization Mechanism

The current implementation of the GUIDE system is based on UNIX. Activities are mapped on UNIX processes and shared objects are represented in shared memory (the implementation details are described in [Dec88a]). We therefore had to rely on the UNIX primitives for an efficient implementation of the high-level synchronization scheme described in section 2.

Translation and evaluation of the high-level synchronization constraints expressed by a user are performed in each GUIDE system call by the following code:

```
        /* PROLOGUE: Synchronization expression evaluation */

P(Synchro_Eval);
WHILE ( NOT MethodSynchroFunction(object) ) DO
        /* Test the Control clause associated */
        /*   with the Effective Method Code   */
BEGIN
        /*      Activity must wait on this object       */
        /* 1) Insert its identification into the object */
        /*         waiting activity list                */
        /* 2) Release the CPU                           */

   Wait_On_This_Object(object);
   V(Synchro_Eval);
   Stop_This_Activity();
   P(Synchro_Eval);
END
V(Synchro_Eval);

        /* BODY: Effective call to the requested object method */

EffectiveMethodCall();

        /* EPILOGUE: Restart of stopped activities */

P(Synchro_Eval);
activity = First_Waiting_Activity_On_Object(object);
WHILE ( activity != NIL_ACTIVITY ) DO
        /* All waiting activities must be resumed because     */
        /* they must themselves re-evaluate their own         */
        /* synchronization expressions. Many resumed          */
        /* activities may really restart in parallel, while   */
        /* the others return to wait for a future evaluation. */
BEGIN
   Resume_Activity(activity);
   activity = Next_Waiting_Activity_On_Object(object);
END
V(Synchro_Eval);
```

4. UNIX Methods for Internal Synchronization

4.1. Standard semaphore synchronization

UNIX System V provides a natural way of synchronizing independent processes: these are semaphores (*semget*(2), *semop*(2), *semctl*(2)), whose characteristics are:

- Their identification is a key whose uniqueness must be handled by applications themselves.
- They are "generalized semaphores", which means that within one system call, many "atomic" operations on many semaphores in the same group may be performed, and the global operation is atomic.
- Their existence is permanent in the system, even after the death of their creating process.
- They provide access rights based on the classic owner-group-other UNIX protection levels.

For our usage, they present two major drawbacks:

- They are very slow because we often use them, and most of the time, the operations involved are non-blocking (take a look at the final performance comparison table in section 5 of this paper).
- We must provide a mapping between the GUIDE objects to protect and the semaphore keys.

The main advantage of this solution was its simplicity. It was used for the first implementation, and we decided to change when the performance experiments showed us that synchronization was the most expensive part of the GUIDE kernel.

4.2. Other kinds of synchronization

The natural way to upgrade performance is to run synchronization operations in user space. We will reach the performance maximum speed depending on the access time of the item used as semaphore counter (a part of shared memory in our case). The use of "subtract quick indirect" and "add quick indirect" assembly operations permits the counter management in mutual exclusion because we only use monoprocessor machines. And this actually works well in most of our tests. The method used was:

- For P(sem):

```
if (increment(i)) {
        push_me_in_the_waiting_queue(i);
        pause();
}
```

- For V(sem):

```
if (decrement(i)) {
        p = get_next_process_in_the_queue(i);
        kill(p, wakeup);
}
```

The method is attractive and simple, but provides a big trap due to the UNIX process scheduling which is appropriate for a time-sharing kernel. In this case, this means that between the "increment(i)" operation, and the "pause()" system call, the CPU may be preempted and allocated to another process. Unfortunately, the scheduled process may operate a V(sem) operation on the semaphore, which will produce the deadlock of this mechanism: The process running the P(sem) operation may receive the signal when rescheduled, go on into the "pause()" system call, and wait forever since it has already received the signal whose goal was precisely to make it leaving the "pause()".

This mechanism is too coarse and thus needs to be refined.

4.3. The complete mechanism

The goal to reach is to suppress the gap existing between the "increment(i)" operation and the "suspend" operation initiated by the "pause()" system call. The classic solution is to use the "*setjmp/longjmp*(3)" library call, whose generic explanation is: "non local goto". The complete mechanism becomes:

- For P(sem):

```
#include <setjmp.h>
jmpbuf ret;
signal(SIGUSR1, onsig);
if (setjmp(ret) == 0) {
        if (increment(i)) {
                push_me_in_the_sem_queue(i);
                pause();
        }
}

onsig(i) {
        longjmp(ret, 1);
}
```

V(sem) is not modified. We thus insure that the process executing P(sem) never falls into the "pause()", if rescheduled before executing the "pause()" system call. We want to warn people on a particular point: when you use a "longjmp()" in a signal handler, the future behaviour of the signal mechanism for the signal it is used to, is UNIX Version dependent; it depends also on the system call used to associate the handler to the signal (*signal*(), *sigset*(), *sigvec*(), ...).

The main drawback of this mechanism is the complexity (signal management, number of library and system calls, ...). The last point decreases the performance of this system, whose main advantage is its portability on any UNIX system. This is the reason why we decided to develop a different solution. In fact, we do not implement this complete solution and prefer to examine immediately the following one, which is more flexible and rich.

4.4. The driver solution

A semaphore is in fact an association between a counter and a queue of waiting processes. The main problem encountered in the precedent solution is the fact that we do not control the CPU allocation between the test of the counter value and the insertion of the process into the waiting list.

One way to solve this problem is to develop into the kernel the sequence of mutual exclusion. In fact, UNIX offers to its users a way to develop pieces of kernel code even if they have no access to the kernel sources: writing a new driver. In addition, most UNIX kernels are today not preemptible, which means that if a process runs a part of the kernel, it may be interrupted only by hardware interrupts, and in this case, it will get the CPU back after return from the interrupt. This solves in a nice manner the problem of mutual exclusion in a monoprocessor machine.

In UNIX, there are two sorts of drivers: character drivers and block drivers. The latter are used if it is necessary to pass through a cache (block cache) and are usually reserved for mass-storage drivers. The link between drivers and other parts of the kernel is achieved through a kernel table (called "cdevsw[]") whose index is the major number of device, and elements entry points to the driver functions. There are at most open, close, read, write, ioctl, ... It is then quite easy (with the help of manufacturer documentation) to add a driver as a piece of kernel while keeping the kernel independence of the GUIDE system. Moreover, this piece of kernel will do exactly what we want, and will of course be efficient.

We will use two ioctl commands to support P(sem) and V(sem) operations, while using the "*open*()" and "*close*()" normal operations to deal with initialization and shut-down. One immediate advantage is the possibility to initialize each time we want the semaphore driver at any "*open*()". We notice that the close driver routine is called only on the last "*close*()", while the open routine is called at each "*open*()". We introduce an ioctl function with two commands: GUIDE_SEM_P and GUIDE_SEM_V, and a semaphore value initialization command: GUIDE_SEM_INIT.

The P(sem) and V(sem) operations automatically becomes ioctl system calls with the right command (GUIDE_SEM_P and GUIDE_SEM_V). The simplicity and the ability to write our own process synchronization are the main characteristics of this solution. The major drawback remains in the fact that the modification of the driver needs a kernel relinking and a reboot of the system, and that we introduce GUIDE system configuration parameters at different places in a machine.

From a performance point of view, the results were better (cf. the final performance table), but still not sufficient. So we imagined a compromise between the two precedent solutions linking the simplicity of the kernel driver and the performance of the user level calls.

4.5. The final solution

Our goal is to minimize the number of context changes due to system calls to keep the performance level reached by the user level synchronization, since the P(sem) operations in our system are most of the time non blocking. In such cases, system calls are a waste of time and we imagine a system with two levels of synchronization:

- At **user level.**
- At **kernel level.**

The kernel level mechanism is heavier than the user level and is called only when the P(sem) operation is blocking or when the V(sem) operation requires inter-process communication. The semaphore counter always contains the number of processes waiting for the resource, while the increment and decrement remains the "add quick indirect" and "subtract quick indirect" assembly operations on MC68020.

The P(sem) operation becomes:

```
if (increment(i))
        ioctl(fd_driver, GUIDE_SEM_P, i);
```

The V(sem) operation becomes:

```
if (decrement(i))
        ioctl(fd_driver, GUIDE_SEM_V, i);
```

The user level is reduced to the two functions "increment(i)" and "decrement(i)" written in assembly language and which role is to modify the counter value in an atomic operation. All the needed inter-process communication and synchronization are implemented as a kernel driver whose listing is given in annex.

The counter value is a rough approximation of the real and atomic value which is computed in the kernel **IF NEEDED**. In fact, the control traps into the driver only on a real mutual exclusion or on a wrong one generated by a scheduling policy. It is the reason why we call this mechanism a "two-level lazy synchronization".

We do not need the UNIX semaphore "undo" flag, because, if a process dies between P(sem) and V(sem) operations, it is a GUIDE system error on the object locked, and an emergency procedure must be applied to this object.

4.6. GUIDE implementation details

This final solution is used at two different levels in our synchronization mechanism: first, each object in our system has an entry in a fixed size array, each entry of which represents a currently mapped object. We use the object index in this array to index the semaphore array present in the kernel. This semaphore is used as synchronization evaluation semaphore for one given object and its counter is initialized to 1. When an activity has to stop, it must use a private semaphore on which it will run a P(sem) to stop, and on which the resuming activity will perform a V(sem). Since we use an array with the same characteristics as those of the mapped objects, we will use the same indexing method for activity semaphores which will be initialized to 0.

One other advantage which appears with such a mapping is the possibility to avoid mapping between semaphore array entries and semaphores.

5. Experience and Performance

A prototype implementation of GUIDE has been developed on top of UNIX, on a local area network connecting Sun3, Bull DPX1000 and Bull DPX2000 workstations [Dec88a] This prototype has been in experimental use for 6 months, and several medium scale applications have been developed in the GUIDE language, for which a compiler has been written. The system supports multi-node computations and a distributed persistent storage. The synchronization mechanism has been used in several applications.

The performances of the intermediate solutions, developed for synchronization mechanism implementation, have been measured with UNIX processes that performed sequences of P(sem) and V(sem) operations. The following table shows the user elapsed time (in microseconds) to perform a P(sem) operation followed by a V(sem) operation.

# of concurrent processes	1	4	8	16
Standard UNIX Semaphores	850	3500	9400	12800
Our semaphore driver	400	1500	3100	4800
Our "two-levels" semaphore	10	40	85	140

Figure 1: *Performance Comparison Table.*

The "increment(i)" and "decrement(i)" parts of P(sem) and V(sem) operations are implementeded as subroutines. So, we lost a few microseconds to run the "jump to subroutine" and "return from subroutine" assembly instructions. But the main interest is the scale difference between the three solutions.

It is important to point out that this solution is general enough to be reused by other kinds of UNIX applications. However, for using such a method in a multi-processor environment, it is necessary to protect the "increment(i)" and "decrement(i)" parts of P(sem) and V(sem) with operations insuring atomic access between processors. This may be provided by "Compare And Swap with operand" MC68020 instructions. We may use them, but we prefer keeping the simplicity of the "add quick" and "subtract quick" instructions.

References

[Dec88a] D. Decouchant, A. Duda, A. Freyssinet, E. Paire, M. Riveill, X. Rousset de Pina, and G. Vandome, "GUIDE: An Implementation of the Comandos Object Oriented Architecture on UNIX," *Proc. EUUG Autumn Conf.*, pp. 181–193, Lisbon, October 1988.

[Dec88b] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, and C. Roisin, "A Synchronisation Mechanism for Typed Objects in a Distributed System," *Workshop on Object-Oriented Concurrent Programming*, San Diego, October 1988. In SIGPLAN Notices, 1989

[Hor87a] C.J. Horn and S. Krakowiak, "An Object-Oriented Architecture for Distributed Office Systems," *Proceedings ESPRIT Technical Conference*, Brussels, September 1987.

[Kra89a] S. Krakowiak, M. Meysembourg, M. Riveill, and C. Roisin, "Design and Implementation of an Object Oriented, Strongly Typed Language for Distributed Applications," GUIDE report number 8, June 1989. To appear in Journal of Object-Oriented Programming

Appendix: The Driver Source Code

First, we give the driver include file which will be included in user application source code to use the driver functionalities.

```
/*---------------------------------*/
/* Ioctl block parameter structure */
/*---------------------------------*/
typedef struct {
   int       semNo;       /* semaphore number */
   int       semVal;      /* semaphore init value:
                             only used with GUIDE_SEM_INIT flag */
}         T_guideSemParam;
/*-------------------------------------------------------------*/
/* Kernel ioctl() commands                                     */
/*-------------------------------------------------------------*/
#if ( defined(sm90) && (!defined(SPS_TEST)) )
/* Semaphore command introducer */
#define GUIDE_SEM    ('G' << 8)
/* Semaphore init */
#define GUIDE_SEM_INIT   (GUIDE_SEM | 1)
/* Semaphore P operation */
#define GUIDE_SEM_P  (GUIDE_SEM | 2)
/* Semaphore V operation */
#define GUIDE_SEM_V  (GUIDE_SEM | 3)
#endif

#ifdef sun
/* Semaphore init */
#define GUIDE_SEM_INIT   _IOW(G,1,T_guideSemParam)
/* Semaphore P operation */
#define GUIDE_SEM_P  _IOW(G,2,T_guideSemParam)
/* Semaphore V operation */
#define GUIDE_SEM_V  _IOW(G,3,T_guideSemParam)
#endif
```

And finally, we present the UNIX driver source code.

```
#include <sys/types.h>
#ifdef sun
#include <sys/time.h>
#include <sys/ioctl.h>
#endif
#ifdef sm90
#include <ufs/dir.h>
#endif
#include <sys/proc.h>
#include <sys/param.h>
#include <sys/user.h>
#include <sys/file.h>
#include <errno.h>
#include "driver/guideSem.h"

/*-----------------------------------------------------*/
typedef struct {
   int       nextProcIndex;  /* next waiting process structure */
   struct proc *uProcAddr;   /* waiting process structure address */
}         T_procElem;

#define P_NIL_INDEX  -1
/*-----------------------------------------------------*/
typedef struct {
   T_long    val;          /* semaphore value */
   int       queue;        /* queue head of waiting processes */
}         T_guideSem;
/*-----------------------------------------------------*/
/* Error Return function different between BSD and SV */
/*-----------------------------------------------------*/

#ifdef sm90
#define RETURN(val)      { u.u_error = val; return; }
#define RETNUL           return
#endif

#ifdef sun
#define RETURN(val)      return(val)
#define RETNUL           return(0)
#endif

/*----------------------------*/
/* SEMAPHORE DRIVER VARIABLES */
/*----------------------------*/
int       g_procFreeHead;
T_procElem g_procTab[MAX_PROC_ELEM];
T_guideSem g_sem[MAX_SEM];
/*-------------------------------------*/
static void procElemInit ()
{
int       i;

   for (i = 0; i < (MAX_PROC_ELEM - 1); i++)
      g_procTab[i].nextProcIndex = i + 1;
   g_procTab[i].nextProcIndex = P_NIL_INDEX;    /* Last process element */
   g_procFreeHead = 0;          /* Head of process element free list */

}
/*-------------------------------------*/
static int procElemAlloc ()
{
int       result;

   if (g_procFreeHead == P_NIL_INDEX)
      return (P_NIL_INDEX);
   result = g_procFreeHead;
   g_procFreeHead = g_procTab[result].nextProcIndex;
```

```
   return (result);

}
/*--------------------------------------*/
static void procElemFree (processIndex)
int       processIndex;
{

   g_procTab[processIndex].nextProcIndex = g_procFreeHead;
   g_procFreeHead = processIndex;

}
/*--------------------------------------*/
static void addToList (headAddr, processIndex)
int      *headAddr;
register int processIndex;
{

   g_procTab[processIndex].uProcAddr = u.u_procp;
   if (*headAddr == P_NIL_INDEX) {   /* This list is empty */
      *headAddr = g_procTab[processIndex].nextProcIndex = processIndex;
   } else {                /* There is at least one element */
      g_procTab[processIndex].nextProcIndex = g_procTab[*headAddr].nextProcIndex;
      g_procTab[*headAddr].nextProcIndex = processIndex;
      *headAddr = processIndex;
   }
}
/*--------------------------------------*/
static void deleteFromList (headAddr, processIndex)
int      *headAddr;
int       processIndex;
{
int       i,
          previousProc;

   i = processIndex;
   do {
      previousProc = i;
      i = g_procTab[i].nextProcIndex;
   } while (i != processIndex);
   if (previousProc == processIndex) {     /* Only one element */
      *headAddr = P_NIL_INDEX;
   } else if (previousProc == g_procTab[processIndex].nextProcIndex) {
      /* Only two elements in the list */
      *headAddr =
      g_procTab[previousProc].nextProcIndex =
      previousProc;
   } else {                /* At least three elements in the list */
      g_procTab[previousProc].nextProcIndex = g_procTab[processIndex].nextProcIndex;
      if (*headAddr == processIndex)
         /* The element to be deleted was the list head */
      *headAddr = g_procTab[processIndex].nextProcIndex;
   }
}
/*--------------------------------------*/
static int extractFromList (headAddr)
int      *headAddr;
{
int       result;

   if (*headAddr == P_NIL_INDEX)     /* the list is empty */
      return (P_NIL_INDEX);
   else {            /* the list is not empty */
      if (g_procTab[*headAddr].nextProcIndex == *headAddr) {
      /* Only one element in the list result = *headAddr;
      *headAddr = P_NIL_INDEX;
      } else {            /* More than one element */
      result = g_procTab[*headAddr].nextProcIndex;
      g_procTab[*headAddr].nextProcIndex = g_procTab[result].nextProcIndex;
      }
      return (result);
```

```
   }
}
/*--------------------------------------*/
guideSemopen (dev, flags)
dev_t     dev;            /* device to open */
int       flags;          /* open flags */
{

   if (flags & FWRITE) {
      printf ("GUIDE semaphore driver init0);
      procElemInit ();
   }
   RETNUL;
}
/*--------------------------------------*/
guideSemclose (dev, flags)
dev_t     dev;            /* device to close */
int       flags;          /* close flags */
{

   RETNUL;

}
/*--------------------------------------*/
guideSemioctl (dev, cmd, arg, flags)
dev_t     dev;            /* device to control */
int       cmd;            /* command to apply */
caddr_t   arg;            /* argument */
int       flags;          /* open flags */
{
T_guideSemParam parameters;
register T_guideSemParam *params;
register int processIndex;

   switch (cmd) {
   case GUIDE_SEM_INIT:
      {

#ifdef sun
      params = (T_guideSemParam *) arg;
#endif

#ifdef sm90
      params = &parameters;
      if (copyin (arg, (caddr_t) params, sizeof (T_guideSemParam)) == -1)
         RETURN (EFAULT);
#endif
      if ((params->semNo < 0) || (params->semNo >= MAX_SEM))
         RETURN (EINVAL);
      g_sem[params->semNo].val = params->semVal;
      g_sem[params->semNo].queue = P_NIL_INDEX;
      RETNUL;
      break;
      }
   case GUIDE_SEM_P:
      {
#ifdef sun
      params = (T_guideSemParam *) arg;
#endif

#ifdef sm90
      params = &parameters;
      if (copyin (arg, (caddr_t) params, sizeof (T_guideSemParam)) == -1)
         RETURN (EFAULT);
#endif
      if ((params->semNo < 0) || (params->semNo >= MAX_SEM))
         RETURN (EINVAL);
      if ((--g_sem[params->semNo].val) < 0) {
         if ((processIndex = procElemAlloc ()) == P_NIL_INDEX)
            RETURN (EAGAIN);    /* No more waiting process structure */
         addToList (&g_sem[params->semNo].queue, processIndex);
```

```
      if (sleep (&g_procTab[processIndex], (PZERO + 1) | PCATCH)) {
            deleteFromList (&g_sem[params->semNo].queue, processIndex);
            procElemFree (processIndex);
            g_sem[params->semNo].val++;
            RETURN (EINTR);
         }
       }
       RETNUL;
       break;
       }
   case GUIDE_SEM_V:
       {
#ifdef sun
       params = (T_guideSemParam *) arg;
#endif

#ifdef sm90
       params = &parameters;
       if (copyin (arg, (caddr_t) params, sizeof (T_guideSemParam)) == -1)
          RETURN (EFAULT);
#endif
       if ((params->semNo < 0) || (params->semNo >= MAX_SEM))
          RETURN (EINVAL);
       if ((++g_sem[params->semNo].val) <= 0) {
          if ((processIndex = extractFromList (&g_sem[params->semNo].queue))
             == P_NIL_INDEX)
             RETURN (ESRCH);     /* There isn't waiting process on this semaphore */
          wakeup (&g_procTab[processIndex]);
          procElemFree (processIndex);
       }
       RETNUL;
       break;
       }
   default:
       {
       RETURN (EINVAL);
       break;
       }
   }
}
/*---------------------------------------*/
```

Social Aspects of EUUG and USENIX

John S. Quarterman

Texas Internet Consulting
701 Brazos Suite 500
Austin, TX 78701
U.S.A.
jsq@longway.tic.com

ABSTRACT

EUUG and USENIX conferences are quite similar in many ways, but they also differ, particularly in their social aspects. This paper presents some comments about technical and social events scheduled as part of the conference proper, and about more informal activities. The continental network cultures are contrasted. The purpose is not to show that one conference is better than another; many of the more basic features are not reproducible (and probably not desirable) in a different environment. But the organisers of each conference have benefitted in the past by adopting some features of the other, and still could do so.

1. Introduction

The basic topic of this paper is UNIX communities on the two continents, as reflected in and served by the organisations, conferences, and networks. Details of the formal EUUG and USENIX organisations are avoided here.

1.1. EUUG and USENIX

EUUG and USENIX are the major technical and professional associations related to the UNIX operating system in Europe and North America, respectively. They each draw from a large community of people who use and develop that system, and they serve similar functions for their communities. The conferences and communities are similar enough that there is much interaction between them. Messages pass between the continents over computer networks. People present papers at conferences on the other continent, or just attend them. The newsletters of each organisation publish material from the other. The two sets of boards of directors communicate frequently and occasionally fund joint projects, such as the current joint representative to the ISO/IEC TC22 WG15 POSIX committee.

All this interaction has led to mutual curiousity. Many similarities, such as those just mentioned, are easy to see. Some of the differences are also obvious, for example, about 400 people attend each EUUG conference, while about 2000 attend the USENIX ones. Others are not well known, but are clear if investigated, e.g., EUUG has about 4000 members because all members of *EUnet* are also EUUG members, while USENIX has about 2500 members and has never tied itself closely to the *USENET* or *UUCP* networks. Large facts such as these lead to more subtle effects, some of which are discussed in this paper.

1.2. Networks

The computer networks used by these communities are as important to them as these conferences. Some important similarities and differences between the networks and their services on the two continents are discussed below.

1.3. Organisation of This Paper

This paper is arranged in three main parts after this introduction:

- Technical Events
- Social Events
- Networks

Within each of these sections, the presentation starts with the more formal events or aspects and moves to more informal ones. There is a question at the end, and acknowledgments.

2. Technical Events

The technical events are the reasons people use to persuade their employers to send them to these conferences, so we'll discuss them first.

2.1. Formal Technical Agendas

The formal technical agendas of the conferences are their most similar aspects. Each organisation holds two annual conferences with a formal program chair, a program committee, and three days of presentations that last about 20 minutes each, usually plus a keynote speaker. Most of these conferences have required abstracts be submitted for consideration, both publish proceedings, and both reserve the right to cancel presentations if papers are not received in time for printing in the proceedings. USENIX has recently experimented with requiring submissions of full papers, and come to the conclusion that this should be done at most every other conference.

Each conference has two days of full-day tutorials on specialised topics.

Both EUUG and USENIX hold vendor exhibits. USENIX usually only does this at their summer conferences, because their winter conferences are usually held concurrently with the UniForum conferences, and UniForum includes a large trade show. EUUG does not have this restriction.

USENIX has a press room and a paid public relations person who publishes a daily conference newssheet.

In addition to their conferences, USENIX also holds frequent small two or three day workshops on specialised topics. These topics have included graphics, supercomputers, software management, transaction processing, large systems installation administration, and distributed systems. One workshop, on the C++ language, has turned into an annual conference series, although there are still attempts to keep a small workshop associated with it. These workshops are widely liked by the community. But they are also widely seen as a major reason for the decline in number and quality of papers submitted to the main conferences in the last few years: people submit papers to a workshop specialising in their topic instead of to a conference.

EUUG has not held any workshops so far. Discussions between the EUUG Executive and the USENIX Board about a possible joint workshop on system administration to be held in Europe are planned for the Autumn 1989 EUUG conference in Vienna.

2.2. Informal Technical Events

Both conferences have panel discussions on the regular technical agenda from time to time. The credibility and popularity of these waxes and wanes.

USENIX schedules WiP (Work in Progress) sessions, to which anyone can submit a one-page abstract. Time slots of ten minutes are assigned first-come, first-serve. WiPs seem to be quite popular, and are useful for work not ready in time for submission of a regular paper. They are also used for various other purposes, such as to get a feel for interest in a topic. WiPs are normally scheduled on the regular daytime technical agenda.

BOF (Birds of a Feather) meetings are held at USENIX conferences. A BOF is an informal gathering with a moderator but no formal agenda, held at the request of conference attendees. Each is traditionally two hours long and is held in the evening. Some BOFs are scheduled months in advance, and any that are scheduled by the time the preprinted agenda goes to press are listed in it. Others are scheduled at the conference, and may be added up until the actual time slot by writing on a board displayed for that purpose.

These BOFs have become so popular at USENIX that there are as many as twelve scheduled against each other on the two evenings traditionally reserved for them, Thursday and Friday. BOFs were also scheduled on Tuesday and Wednesday at the most recent USENIX conference, but even that does not seem to be sufficient. It is likely that the next USENIX conference, Winter 1990 in Washington, D.C., will schedule BOFs during the day, possibly opposite some technical program items.

Because USENIX supplies meeting space for BOFs, any conference attendee must be admitted, and money cannot be collected. There is no restriction on topics to be discussed. Most BOFs are technical, but some are social. Some have led to changes in the networks and the conferences. One, the Women's BOF, presented proposals to the Board for most of the new conference events described here.

There has never been a BOF at an EUUG, to my knowledge. Something called that was held at the Spring 1989 meeting in Brussels, but was in fact a series of formal presentations followed by a panel discussion.

USENIX has had a terminal room at each conference starting with Summer 1988 in San Francisco. The basic service is dialup modem access. Since Winter 1989, it has also included local computers and direct access to the TCP/IP *Internet* by means of a dialup SLIP connection. The original problems were finding equipment to use, volunteers to staff the room, and conference attendees interested in using it. The current problem is deciding what equipment to accept, because more is being offered than can be used. This facility has become an accepted and expected part of the conference, and the room also serves as a social gathering place.

Other recent USENIX innovations include an opening night party (usually Sunday evening, and intended to allow those who came for the early tutorials to meet), an orientation session for newcomers (usually Monday evening) a free short tutorial on writing and presenting papers (often Tuesday), and a lounge (with soft drinks) open throughout the conference. Other possibilities are being considered for upcoming conferences. Some of these might be useful at EUUG. Others, such as the opening night party, are clearly not needed (everybody gets together in the bar anyway).

These USENIX events that are not part of the formal technical program have become sufficiently numerous that the USENIX Board of Directors has appointed a Chair of Informal Programming for each conference since the Summer 1989 one in Baltimore. The purpose of this position is to allow the Technical Program Chair to concentrate on organising the technical program, while still ensuring that someone organises the other events. The Informal Chair is in charge of organising events, especially for finding volunteers to run them.

3. Social Events

The social events are a large part of why people like the conferences when they go. They are also invaluable opportunities to meet technical peers.

3.1. Scheduled Social Events

The major scheduled social event held by each organisation is the mid-conference reception, with food and drink in some attractive local setting. EUUG also includes lunches in the registration fee. USENIX only does this for tutorials, and supplies only boxed lunches. As mentioned above, USENIX has an opening party, an orientation session, and a lounge.

An EUUG specialty that USENIX has tried to duplicate without success is the contest. The errno contest of years ago is still remembered fondly, even if ENOJOY. The awards presented are also better at EUUG.

3.2. Unscheduled Social Events

Non-scheduled social events and aspects are seldom described in writing, although conference attendees spend much time discussing them.

3.2.1. Discussions

Technical discussions are more politely phrased in Europe. Some speakers are so reserved as to be boring, but some speakers in North America are so disorganised as to be incomprehensible.

Papers at EUUG are often about completed projects, with all the conclusions and reasons wrapped up in a neat package, and little room for further development or question. EUUG papers also tend to be very tightly focussed on specific projects.

USENIX attendees are more likely to write about part of a project even before it is finished, or to write about general subjects that are not limited to a particular project.

EUUG question and answer periods draw fewer questions from the audience. Attendees don't usually ask questions unless a topic directly affects them. One almost never says a speaker was wrong.

Questions asked at USENIX conferences tend to be more thorough, and more oriented towards a solution to a problem. The audience practically competes to show the speaker was wrong.

USENIX is incorporated in the State of Delaware and is thus a U.S. (not-for-profit 501(c)3) corporation, but it has always been international. Canadians have been involved from the beginning, and there are many members from elsewhere. However, language is seldom a problem at USENIX.

EUUG is far more international due to the national divisions of Europe itself. Most EUUG conferences have English as their offical language, but that is not the native language of many of the attendees. Adherence to presentational caution behooves one in avoidance of obfuscation, cant, and presto prose. (If you didn't understand the last sentence, you saw the point illustrated.) This applies no matter what the speaker's native language is. A native English speaker speaking quickly and a speaker of another mother tongue speaking quickly are equally unintelligible to large (though perhaps different) parts of the audience.

The effect of nationalities extends not only to language but also to social conventions and can lead to miscomprehension. But Europeans who think that the U.S. is all the same will be in for a surprise when they attend their first USENIX conference.

3.2.2. Eating and Drinking

As already mentioned, there are about 2000 attendees at each USENIX and about 400 at each EUUG. Just finding someone at a USENIX can be difficult, while most EUUG attendees can be located in the hotel bar. This difference in size partly accounts for many other things, such as the higher attendance fee at EUUG, although the inclusion of lunches in the fee also contributes to that.

Everyone at EUUG eats lunch together, and the reception is a sit-down affair. This is impossible at USENIX, because there are just too many people. There is usually no restaurant list distributed with EUUG conference materials, but there are usually many restaurants within walking distance.

Drinking is more socially acceptable at EUUG. Wine is served at lunch, and it seems that almost everyone has wine or beer at dinner and goes to a bar afterwards. EUUG has a semi-official beer for each conference. Even Americans at EUUG agree that "the beer's better here." Mixed drinks and liquors are more popular at USENIX, but not during the day, and only for some attendees.

There's no ice tea at EUUG, and no ice for that matter, and fewer soft drinks. There is coffee, and usually good coffee.

Europeans roll their own cigarettes and use good tobacco. Smoking is far more common in Europe.

3.2.3. Vendor Hospitality

Hospitality suites are common at USENIX. These are suites in the conference hotel that are rented by vendors so that they can offer free refreshments and access to technical people in hopes of attracting conference attendees so that they can improve the visibility of their company, find potential employees, or even sell products. These hospitality suites are a big part of the evening socialising at USENIX. Some are held at EUUG, but not as many, and people are more likely to just go out to a bar.

Vendors give away free trinkets ("freebies") at the vendor exhibits and hospitality suites of both conferences, but they tend to be better at USENIX (although they weren't in Baltimore). Good examples include the famous mt Xinu calendar with all three dates that 4.3BSD was released and humorous artwork, and the Sun Microsystems shoelaces. HCR of Toronto probably is the most consistent, with good items ranging from their "Sex, Drugs, and UNIX" buttons in Toronto 1983 to their "Condoms, Aspirin, and POSIX" buttons at Baltimore 1989, not to mention their UNIX Port and their UNIX Founding Fathers plates.

There is a job board at USENIX, where people can advertise jobs wanted and offered.

3.2.4. Composition

Conference attendance differs not only in size but also in composition. Perhaps twenty five percent of the attendees at an EUUG conference are regulars who go to most EUUG conferences. There are such people at USENIX, but not as large a proportion, and it seems to be more common to go to only one conference a year. Few EUUG attendees go to a conference without support by their employer, and EUUG tends to be somewhat more oriented towards organisations.

As someone put it, there are "fewer suits" at EUUG. Both organisations seem to worry about that less now that they have been around for a while and their favorite software seems to be taking over the world. As someone else put it "we were hypersensitive to suits in those days."

The number of women at USENIX continues to increase, and is currently about fifteen percent. The number of women at EUUG tends to be smaller, and of those seen at conferences, more are non-technical spouses; the common USENIX phenomenon of couples of people both in the business is very rare at EUUG. Nonetheless, babies are seen more often at EUUG, and attendees do not seem to expect day care. Many USENIX attendees have children but do not bring them, complaining instead about the lack of day care; there are tentative plans to provide this at the next conference.

The proportion of homosexuals at USENIX conferences is generally estimated at about twenty percent. If there are any at EUUG, they are very inconspicuous.

Unmarried women report that they are "hit on"† almost never at EUUG and almost all the time at USENIX conferences. If they ever accept at EUUG, they are very discreet about it. Europeans tend to be rather reticent about discussing sexual habits in general, so I'll change the subject.

3.2.5. Swimming

EUUG meets in places like Portugal, where you could swim in the ocean or a salt water pool. USENIX meets in places like San Diego, but only in the winter. But USENIX usually has pools (for the synchronised swim team), and often a jacuzzi (for gossip).

3.2.6. Outside Issues

Public transportation in Europe is generally better than in North America, and there is a continental train system that works, so getting to conferences and getting around once you get there is generally easier. This may help account for the higher proportion of regular EUUG attendees.

Americans are more willing to double up in hotel rooms at conferences, but this could be because European hotel rooms are smaller. Europeans stay with friends more, both at conferences and generally when travelling. They also tend to travel in groups, and are more likely to stay or travel after a conference. This may be partly because of the European tradition of six weeks vacation per year, while Americans usually make do with two.

Telephones in Europe are incomprehensible, at least to Americans, and expensive. But, then, Americans are also confused by currency exchange.

4. Networks

EUUG and USENIX draw from and influence communities that exist outside of the conferences. The computer networks used by respective communities are some of the major determinants of the communities, and differ qualitatively and quantitatively between the communities.

4.1. *EUnet*

EUUG can be seen as the political arm of *EUnet*, the European UNIX network. All *EUnet* members are EUUG members. Almost all EUUG members and conference attendees have access to *EUnet*. So there is one universal electronic mail service used by the EUUG community, and many newsgroups are also shared. The continental X.25 infrastructure allows remote login to some extent, although this does not seem to be widely used. The network is organised heirarchically according to national boundaries, fees are collected, service is rather dependable, and there are concerted development efforts.

† An American phrase meaning "propositioned sexually."

Some EUUG members may have access to other networks, such as *EARN*, *HEPnet*, or the Ean networks [Karrenberg and Goos 1988], but those other networks do not appear to have a large influence on the community. Many European countries have national research networks, such as *JANET* in the United Kingdom or *DFN* in Germany. As influential as these may be nationally, they do not usually have much effect on other nations, and they are not specifically oriented towards UNIX, unlike *EUnet* and its various national branches. The European UNIX community has essentially one common network culture, that of *EUnet*.

4.2. *USENET*, *UUCP*, and *UUNET*

The network in North America corresponding to *EUnet* is two networks: *USENET* (news) and *UUCP* (mail) [Todino and O'Reilly 1988]. These are far more loosely organised than *EUnet*. There is no imposed hierarchical structure, and regional divisions tend to be more economic than political. There is no close formal relationship with USENIX. There are no fees collected, except by the telephone companies. Development efforts usually occur when people decide they need to be done, sometimes at BOFs at USENIX conferences.

USENIX has, however, sponsored experiments in improvement of these two networks. For example, *UUNET*, originally such a USENIX experiment, has proved to be a technical and financial success, and has been spun off into a separate non-profit corporation. It performs many of the functions of cwi.nl (mcvax), the central *EUnet* host, although it is in some senses more restrictive in what it supplies and in some senses less. Unlike the average *USENET* or *UUCP* host, *UUNET* charges for access. It also provides authorised access to the *Internet*.

4.3. The *Internet*

The United States is the center of an unusual network, the *Internet*. This is a continental research and academic network supported by many different agencies and companies. Although many of the long-haul links are still subsidised by the federal government, the National Science Foundation (NSF) encourages any new regional networks it funds to become self-supporting within a few years.

The *Internet* is sometimes confused with its predecessor, the *ARPANET* (named after DARPA, the Defense Advanced Research Projects Agency). The major distinction is that the *Internet* connects a variety networks using media at various speeds. It can do this because it uses a common set of high level protocols based on the Internet Protocol (IP), and usually called TCP/IP. The *ARPANET* was one of the constituent networks of the *Internet* until it was retired this year.

This internetwork provides fast mail service and is increasingly used for carrying *USENET* news. But in addition to these features it also provides interactive services such as one-to-one conferencing, remote login, and, perhaps most importantly, file transfer. Various hosts on the network keep archives of large amounts of software and other useful information. For example, the *USENET* newsgroup comp.sources.unix is available from *UUNET* by anonymous FTP (login as anonymous, password guest).

There are many other computer networks in North America [LaQuey 1989]. Some are analogous to certain European networks. *BITNET* in the U.S. and *NetNorth* in Canada are equivalent to *EARN*, and are integrated into a common name and address space. *HEPnet* in North America and *HEPnet* in Europe are essentially one large network. Others are more specialised. *SPAN* is mostly composed of Digital machines running VMS. MFEnet caters to the Magnetic Fusion Energy community.

The networks used most by the UNIX community are *USENET*, *UUCP*, and the *Internet*. This produces a certain social distinction between those who have access to the *Internet* and those who don't. This difference is diminishing as more groups and people join the *Internet* through *CSNET* or *NSFNET*, or who gain access to many of the same archives and facilities through *UUNET*.

The services provided by the *Internet* are not widely available in Europe. A common European lament seen in *USENET* newsgroups is "don't tell us it's available by anonymous FTP, that does us no good!"

4.4. ISO-OSI

A major difference between the continents is the wide use of X.25 in Europe. This allows almost any machine to call almost any other using a packet protocol, rather than having to use modems to go through the voice telephone system. X.25 is also rather widely used in Canada, but much less so in the United States.

The International Organisation for Standardisation (ISO) Open System Interconnection (ISO-OSI) layering model and protocol suite are often cited as being the future of the networking world. At the moment, however, there are few continental ISO-OSI networks, other than the European X.25 infrastructure (X.25 is the basic ISO-OSI network layer protocol). The Ean networks and their succesor RARE Experimental MHS Networks are widespread in Europe, in the sense that they exist in most countries, but they have few users. There are national research networks based on ISO-OSI, such as *DFN* in Germany. Most other European networks have ISO-OSI migration plans. Even the *Internet* has one. But no network originally based on other protocols has as yet converted to ISO-OSI, to my knowledge.

The next release of the Berkeley Software Distribution (BSD), perhaps to be called 4.4BSD, and tentatively scheduled for 1990, will include a complete ISO-OSI protocol suite implementation. The major reason for the spread of TCP/IP was the earlier implementation of TCP/IP in 4.2BSD. It's free, and everybody at least looks at it when doing their own implementations. It runs in everything from mainframes to gateway boxes. It will be interesting to see what effect the 4.4BSD ISO-OSI implementation will have.

It is worth noting that the ISO-OSI mail protocol, X.400, is more versatile than the SMTP protocol used in the *Internet*, and the RFC822 mail format used both there and in *EUnet* and *UUCP*. It supports structured documents, multiple human languages, and multi-media mail. But there is no ISO-OSI equivalent to news and newsgroups.

4.5. The *Matrix*

Continental and other distinctions in network service are rapidly diminishing. Most networks are interconnected at least for mail service [Frey and Adams 1989] into one worldwide metanetwork, sometimes called the *Matrix* [Quarterman 1990]. Specifically, *EUnet*, *UUCP*, and the *Internet* are so interconnected, and the two continental UNIX communities make use of this fact constantly. Similarly, many newsgroups are shared between *EUnet* and *USENET*.

EUnet is installing high-speed dedicated links between Amsterdam and Paris, Stockholm, and other places. These connect various European national and international networks such as *FNET* in France or *NORDUnet* in the Nordic countries into a fast continental network. Since IP is used over them, they form the backbone of a continental IP network, sometimes called Reseau IP Européen, or *RIPE*. These links also interconnect most of the transatlantic links to the *Internet* in North America. Thus the classes of service available to the EUUG community may soon be the same as those available to the USENIX community.

5. Summary

USENIX conferences have more things to do, but EUUG conferences are often more fun. The networks connect the communities regardless of what happens at the conferences.

5.1. A Question

Is EUUG following the development pattern of USENIX? Answers tend to range from "no," because the continental backgrounds differ so much, to "maybe," because it is the same subject matter that draws the communities together, and many former differences, such as those of network technologies, are dissolving. What do you think? What will the development paths of the two organisations be? More importantly, what *should* they be?

This question is important because many of the decisions that have shaped the paths of these organisations, conferences, and networks, and thus indirectly the communities they serve, have been intentional. Such a simple thing as a job board was a major point of contention at USENIX at one time. *UUNET* was not invented by the USENIX Board, it was funded by them as the result of a proposal. The interrelations of *EUnet* and EUUG are too well known to Europeans to repeat here. There is a major current controversy in USENIX about what to do about the diminished quality of papers submitted to conferences. EUUG is considering whether to have workshops. The boards of USENIX and EUUG listen to their attendees and members, and need suggestions and proposals, not to mention volunteers.

5.2. Acknowledgments

A structure has been imposed on the paper to make it readable, but in the end it is a compendium of comments from many people who have attended EUUG or USENIX conferences, including Susanne Steuermann, Susanne Smith, Debbie Scherrer, David Rosenthal, Marc Nyssen, Sharon Murrel, Dan Klein, Daniel Karrenberg, Teus Hagen, Anke Goos, Linda Branagan, Eric Allman, Jaap Akkerhuis, and no doubt others who will remind me. Thanks to all of you. The author is solely responsible for the interpretation and presentation of the material in this paper.

References

[Frey and Adams 1989]
Donnalyn Frey and Rick Adams, *!:/@ A Guide to Electronic Mail Networks and Addressing*, O'Reilly & Associates, Inc., Newton, MA, 1989.

[Karrenberg and Goos 1988]
Daniel Karrenberg and Anke Goos, *European R&D E-mail Directory*, European UNIX systems Users' Group, Buntingford, Herts., England, December 1988.

[LaQuey 1989]
Tracy Lynn LaQuey, *Users' Directory of Computer Networks*, Digital Press, Bedford, MA, 1989.

[Quarterman 1990]
John S. Quarterman, *The Matrix: Computer Networks and Conferencing Systems Worldwide*, Digital Press, Bedford, MA, 1990.

[Todino and O'Reilly 1988]
Grace Todino and Tim O'Reilly, *Using UUCP and Usenet*, O'Reilly & Associates, Inc., Newton, MA, 1988.

For further details, contact
The Secretariat

European UNIX® systems User Group

Owles Hall, Buntingford, Herts SG9 9PL, UK
Tel: + 44 763 73039
Fax: + 44 763 73255
Network address: euug@inset.uucp

SBN 0 9513181 3 6