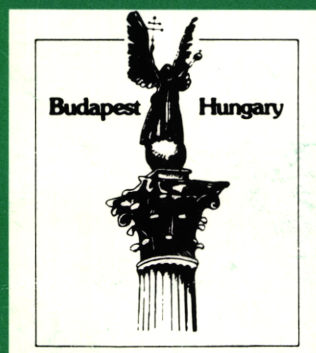

EurOpen



Autumn 1991 Conference Proceedings

Distributed Open Systems in Perspective

16th – 20th September
Budapest, Hungary





UNIX
Goes East

Proceedings of the
Autumn 1991 EurOpen Conference

September 16–20, 1991
Budapest, Hungary

This volume is published as a collective work.
Copyright of the material in this document remains
with the individual authors or the authors' employer.

ISBN 1 873611 01 3

Further copies of the proceedings may be obtained from:

EurOpen Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

These proceedings were typeset in Times Roman and Courier on a PostScript printer driven by a white swan. PostScript was generated using **refer**, **tt**, **pic**, **psfig**, **tbl**, **sed**, **eqn**, **troff**, **pm** and **psdit**.

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

UNIX is a registered trademark of UNIX System Laboratories in the USA and other countries.

AIX, RT PC, RISC System/6000, VM/CMS are trademarks of IBM Corporation.

Athena, Project Athena, Athena MUSE, Discuss, Hesiod, Kerberos, Moira, Zephyr are trademarks of the Massachusetts Institute of Technology (MIT). No commercial use of these trademarks may be made without prior written permission of MIT.

CHORUS is a registered trademark of Chorus systèmes.

CONVEX is a registered trademark of CONVEX Computer Corporation.

DEC, Vax, VMS are trademarks of Digital Equipment Corporation.

Intel 386, Intel 486 are trademarks of Intel Corp.

MC68000, MC88000 are trademarks of Motorola Computer Systems.

MIPS is a trademark of MIPS, Inc.

Motif is a trademark of the Open Software Foundation.

MS-DOS is a registered trademark of Microsoft Corporation.

OPEN LOOK, SVID, System V are registered trademarks of AT&T.

OSF, OSF/1 are trademarks of the Open Software Foundation.

PostScript is a trademark of Adobe, Inc.

Prism is a trademark of HP-Apollo.

Sequent, Symmetry are registered trademarks of Sequent Computer Systems, Inc.

Sun, SunOS, SPARC, NeWS, NFS are trademarks of Sun Microsystems, Inc.

UltraNet is a trademark of Ultra Corporation.

UTS is a trademark of Amdahl Corp.

UNICOS is a trademark of Cray Research Inc.

X Window System is a trademark of MIT.

X/Open is a registered trademark of X/Open Company, Ltd.

XENIX is a trademark of Microsoft Corporation.

Other trademarks are acknowledged.

ACKNOWLEDGEMENTS

This is the first EurOpen conference to be held in Eastern Europe. The technical programme covers a range of topical subjects, with distributed processing and networks as the main themes.

The host organisation is EurOpen Hungary, whose members have done a great deal of work to ensure the success of the conference. In particular, Maria Toth of the John von Neumann Society for Computing Sciences has been EurOpen's main point of contact and local organiser.

The programme committee, consisting of Kim Biel-Nielsen, Mike O'Dell, Johan Helsingius, Frances Brazier, and Elod Knuth, have sifted through all the papers offered and produced the technical programme. The final programme meeting lasted till midnight, during which time the sun did not set!

Thanks are due to the Conference Executive, Ernst Janich, for advice and timely reminders. Neil Todd has once again produced an excellent tutorial programme, and Helen Gibbons and her team at Owles Hall have worked hard behind the scenes to keep everything together and moving in the right direction.

Thanks also to the typesetting crew at Imperial College, who have done wonders with troff to produce the proceedings.

I hope you enjoy the conference and find it useful.

Andrew Findlay
Programme Chairman

Output on a Linotype 60 by Phil Male and Danny
Turner, Computer Newspaper Services, Howden.

These proceedings were specially produced for EurOpen at the Department of Computing, Imperial College, London, using resources generously provided by the Computing Support Group.

– sm, jsp.

Table of Contents

UNIX and Virtual Reality	1
<i>Mike Griffin; Dept. of Cybernetics, University of Reading, UK</i>	
Interactive User Interface Design – The Teleuse Approach	9
<i>Achim Brede; Bredex</i>	
The QEF/QEI Model for Software Component Consistency	11
<i>David Tilbrook; Sietec Open Systems Division</i>	
Give a Process to your Drivers!	13
<i>Francois Armand; Chorus systemes, France</i>	
Multimedia Synchronization and UNIX	29
<i>Dick C.A. Bulterman; CWI, Amsterdam, The Netherlands</i>	
Using a Wafer-Scale Component to Create Efficient Dist. Shared Memory	47
<i>Aarron Gull; City University, London, UK</i>	
Performance Evaluation: The SSBA's at AFUU	67
<i>C. Binot; AFUU, Le Kremlin-Bicetre, France</i>	
Near Real Time Measures of Unix-like Operating Systems	79
<i>Mario Cambiaso; DIST – Universita di Genova, Italy</i>	
Steppingstones: Some Remarks on Measuring X11 Performance	91
<i>Werner Kriechbaum; IBM AIX FSC, Munchen, Germany</i>	
Security and Open Working in the Networked Academic Community	101
<i>Denis Russell; University of Newcastle upon Tyne, UK</i>	
phLOGIN, Why, What and How	117
<i>Alain Williams; Parliament Hill Computers Ltd, UK</i>	
MANIFOLD: A Language for Specification of IPC	127
<i>Farhad Arbab and Ivan Herman;</i>	
<i>Centre for Mathematics and Computer Science, The Netherlands</i>	
A Distributed Concurrent Implementation of Standard ML	145
<i>David C. J. Matthews; University of Edinburgh, UK</i>	
Load Balancing Survey	157
<i>Dejan S. Milojicic;</i>	
<i>Institute "Mihajlo Pupin", Beograd, Yugoslavia</i>	
A Public Access Interface to the OSI Directory	173
<i>Paul Barker;</i>	
<i>Dept. of Computer Science, University College, London, UK</i>	
Managing the International X.500 Directory Pilot	187
<i>Colin J. Robbins; X-Tel Services Ltd, UK</i>	

A Design Overview of XLookUp	199
<i>Damanjit Mahl;</i>	
<i>Manufacturing & Engineering Systems, Brunel University, UK</i>	
An Implementation of a Process Migration Mechanism using Minix	213
<i>Sylvain R.Y. Louboutin; University College, Dublin, Ireland</i>	
HAWKS – A Toolkit for Interpreted Telematic Applications	225
<i>Carl Verhoest; Telesystemes Innovation, Paris, France</i>	
Virtual Swap Space in SunOS	237
<i>Howard Chartock; Sun Microsystems, USA</i>	
The Art of Automounting	249
<i>Martien F. van Steenbergen; Sun Microsystems Nederland B.V.</i>	
Monitoring Network Performance	267
<i>Martin Beer;</i>	
<i>Dept. of Computer Science, University of Liverpool, UK</i>	
StormCast – A Distributed Application	273
<i>Dag Johansen; University of Tromsø, Norway</i>	
Location-Independent Object Invocation in Open Distributed Systems	287
<i>Herman Moons;</i>	
<i>Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium</i>	
Communicating Database Objects	301
<i>Agnes Hernadi;</i>	
<i>Hungarian Academy of Sciences, Budapest, Hungary</i>	
Unix in Novell Environment	309
<i>Gabriella Ivanka;</i>	
<i>Computer Research and Innovation Centre, Hungary</i>	
An International Hotel Reservations System	311
<i>Gary M Bilkus; BLiX Limited, UK</i>	

Author Index

Farhad Arbab <farhad@cw.nl>	127
Francois Armand <francois@chorus.fr>	13
Paul Barker <P.Barker@cs.ucl.ac.uk>	173
Martin Beer <mdb@compsci.liverpool.ac.uk>	267
Gary M. Bilkus <gary@utell.UUCP>	311
C. Binot <ssba@afuu.fr>	67
Achim Brede <achim@bredex.uucp>	9
Dick C.A. Bulterman <dcab@cw.nl>	29
Mario Cambiaso <sumar@dist.unige.it>	79
Howard Chartock <howard@sun.com>	237
P. Dax <ssba@afuu.fr>	67
Susanna Delfino <sumar@dist.unige.it>	79
N. DoDuc <ssba@afuu.fr>	67
M. Gaudet <ssba@afuu.fr>	67
Mike Griffin <mike.griffin@cyber.reading.ac.uk>	1
Aarron Gull <aarron@cs.city.ac.uk>	47
Gunnar Hartvigsen <gunnar@cs.uit.no>	273
Ivan Herman <ivan@cw.nl>	127
Agnes Hernadi <h792her@ella.hu>	301
Shaun Hovers <shaun@compsci.liverpool.ac.uk>	267
Gabriella Ivanka	309
Ferenc Jamrik	301
Gabor Janek	301
Dag Johansen <dag@cs.uit.no>	273
Elod Knuth	301
Werner Kriechbaum <werner@ibm.de>	91
Gyorgy Leporisz	309
Sylvain R.Y. Louboutin <Louboutin@ccvax.ucd.ie>	213
Damanjit Mahl <Damanjit.Mahl@brunel.ac.uk>	199
David C.J. Matthews <dcjm@lfcs.ed.ac.uk>	145
Dejan S. Milojicic <eimp002@yubgss21.bitnet>	157
Richard Mitchell	1
Herman Moons <herman@cs.kuleuven.ac.be>	287
Milan Pjevac <eimp002@yubgss21.bitnet>	157
Colin J. Robbins <C.Robbins@xtel.co.uk>	187
Guido van Rossum	29
Denis Russell <Denis.Russell@Newcastle.ac.uk>	101
Peter Snyder <peter@sun.com>	237
Martien F. van Steenbergen <Martien.van.Steenbergen@Holland.Sun.Com>	249
Giancarlo Succi <charmi@dist.unige.it>	79
David Tilbrook <dt@snitor.uucp>	11
Dusan Velasevic <velasevic%buef78@yubgef51.bitnet>	157
Pierre Verbaeten	287
Carl Verhoest <cave@telesys-innov.fr>	225
Alain Williams <addw@phcomp.co.uk>	117
Dik Winter	29

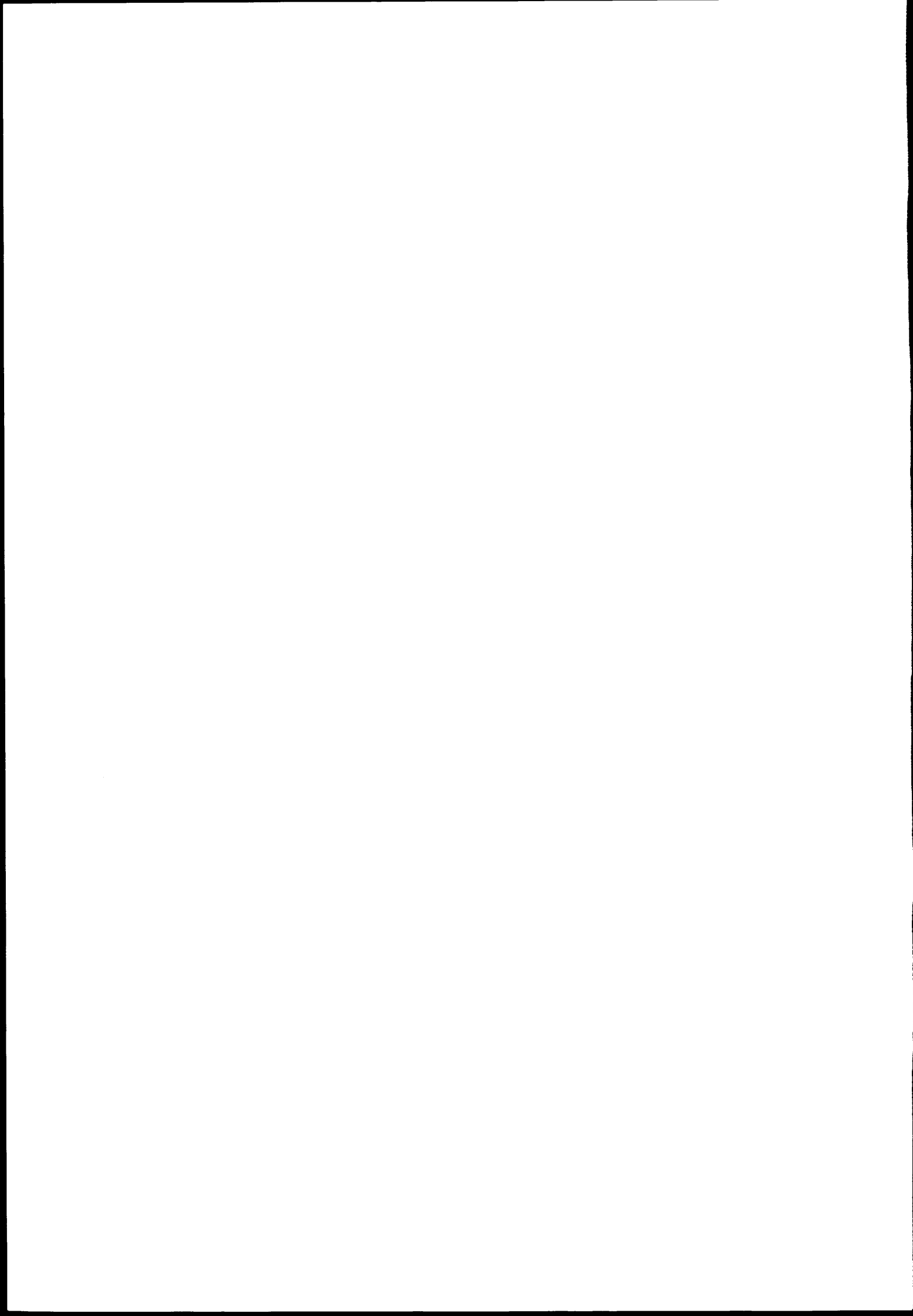
UNIX Conferences in Europe 1977–1991

UKUUG/NLUUG meetings

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

EUUG/EurOpen Meetings

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



UNIX and Virtual Reality

Mike Griffin Richard Mitchell

Department of Cybernetics

University of Reading, UK

mike.griffin@cyber.reading.ac.uk

Abstract

With the growth of interest in Virtual Reality systems, current implementations are based around customised, or experimental processing platforms. This is not an ideal state of affairs, as it tends to reduce compatibility to a bare minimum, and prevent the utilisation of cheap mass storage and processing units.

If a Virtual Reality system is based on a UNIX platform, such that support and communication layers are used, then this tends to reduce the cost of development of the overall system, and allow automatic compatibility between differing designs.

In this paper, the proposed design of a UNIX supported virtual reality system is presented. Practical implementation details are discussed, as well as the current research progress on this project to date.

Introduction

UNIX systems are becoming increasingly prevalent in both research and industrial areas. As this has become the case, the cost of such systems has reduced dramatically, and the size of the software base associated has increased in proportion. It is quite clear, that with the merger of UNIX types in the System V release, this phenomena will increase.

Virtual reality, on the other hand, is a relatively young application and research area. Although the first systems were designed as early as 1968 [Sut68a], display and tracking technology were not sufficiently well advanced to make such systems a suitable commercial proposition. The recent advances in liquid crystal display technology has nullified the first of these problems, and research into magnetic tracking systems has solved the second. Refinement is still needed, but practical presentation hardware is now commercially available.

Aside from the technical problems of developing suitable display and interaction interfaces for Virtual Reality systems, there is the difficulty of making the computer platforms, software and applications packages for these systems. Current research, both in commercial and academic environments, is towards customised, non standard implementations. These tend to be expensive in nature, and lack the basic support to grow and benefit the expanding user base.

It is quite clear that custom operating systems for this area, as well as computing platforms and software, may not be necessary if the underlying basis for such systems was a standard UNIX type platform. Some significant modifications may be necessary on higher layers to support the requirements for Virtual Reality, but a large number of core systems would still be usable. How this may be achieved, can be seen by the examination of an already existing Virtual Reality system.

Thus in the next section the subject of Virtual Reality will be discussed, followed by a section on the system developed at Reading. Then the proposed implementation using UNIX will be given.

Virtual Reality

Virtual Reality or VR is the ultimate example of man-machine interaction [Fol87a]. The basic idea is to present to the user a machine generated world which appears to the user to be real, and which allows the user to interact with that world. There are various applications where such systems may be used, some of which are given here.

Ideally, humans should not have to enter hazardous environments, so it is a good idea to send a machine instead which is controlled remotely. Generating suitable control commands can be difficult, so a good system would be one in which a model of the hazardous scene is made and in which the human operates. The movements made by the human are used to both control the remote machine, which thus changes the environment, and to update the model, so that these changes are made apparent to the user, and so provide the necessary feedback to the user [Gri91a].

Simulators have been used to good effect in the training of pilots for many years. VR systems are advanced examples of such systems, and so could be used for training pilots as well as trainee drivers of cars, etc.

Currently air traffic controllers select where planes should fly using a two dimensional display. A much better system would provide three dimensional information. A VR system would allow the user to appear to be "up in the clouds", and allow him to turn his head around and find the three dimensional position of the planes.

VR systems could also be useful for architects. They could design a building and the VR system would display the building and allow the architect to walk through it.

Another obvious application is games. These could allow both single player games and multi-user games. For the latter other people could appear in the virtual reality and interact suitably. Fun!

These are typical applications, but how are they achieved? As the user perceives a world with all senses, an ideal system would present suitable information to all these senses. Thus a visual scene should be generated, with suitable sound, tactile sensing, appropriate smells and suitable forces affecting the human. Current systems do not achieve all of these, but systems are available which generate stereo visual information, audio, tactile and some force information, thereby generating data for the most important senses.

In many VR systems the user wears a helmet containing two video displays on which information is displayed for each eye. The information on the two displays is suitably different so that the scene displayed appears to be three dimensional. Audio information is easily provided

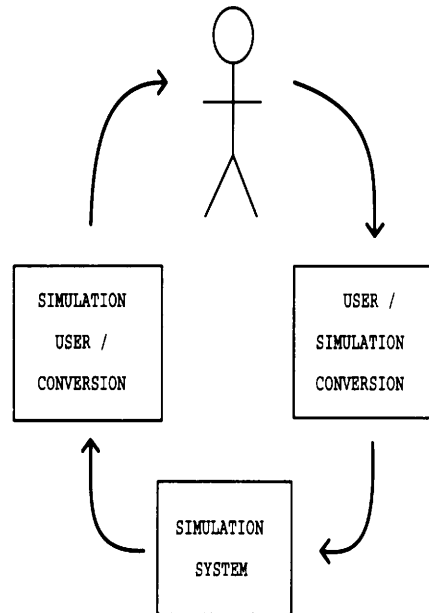


Figure 1: *Virtual Reality System*

using speakers for each ear. The tactile and force information is achieved using a dataglove which the user wears on his hand, and which allows the user to feel that he has picked up an object, for example.

These devices allow the generation of the world, but the user must interact with the world. Thus sensors are required to detect where the user is looking and his position. Thus the position and orientation of the head, and independently the position of the pupils of the eyes, are measured. The dataglove is also used to measure the position of the hand and the relative movements of the fingers [Bur91a].

As a result of these interactions, the user may change the world. Thus the world model should be updated accordingly, and its appearance to the user changed suitably. Figure 1 shows a block diagram of the scheme: this contains the user and the simulated world, with suitable interfaces between them, operating in an interacting feedback loop.

Current technology is available to provide the hardware for VR systems. What is needed now are the tools to generate the virtual realities, including the worlds and the objects within the worlds. At Reading a system called LOKI has been produced for this purpose [Gri91b].

The Reading "LOKI" System

This system is designed as a first generation development to test some of the practicalities of developing small virtual reality systems. It is based around a commonly available microcomputer system, with a custom hardware and software arrangement. Native operating system calls are utilised as often as possible, but the basic interprocess communication in the LOKI server is fundamentally semaphoring and message passing in nature.

A Virtual Reality software package is required to satisfy a number of specific requirements;

1. To enable the user to model an environment.
2. To represent this model to the user.
3. To permit the user to interact with the model.

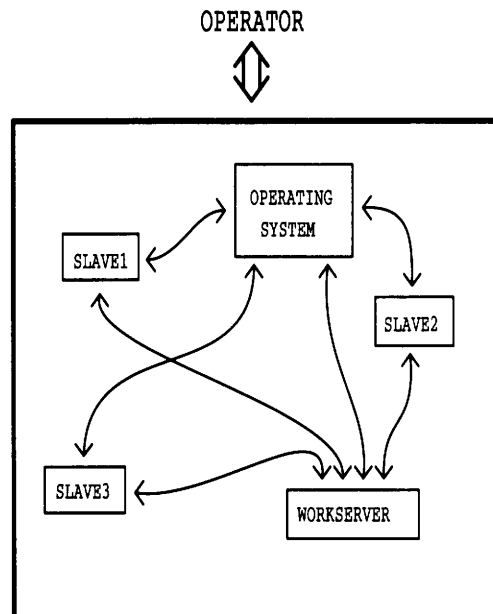


Figure 2: *Virtual Reality Computer*

These requirements are met by the LOKI system in the following manner. An initial server process is run, which acts as a central hub called the Workserver. This server is responsible for the holding of the model, rendering the model database, the supporting of applications, and the running of an object oriented language system used to describe objects in the virtual reality [Gri90a]. The applications are achieved by a number of slave processes which are controlled by and interact with the Workserver, and which use the facilities of the operating system of the machine on which the system runs. A typical application could be a CAD package used to generate the realities: this uses the operating system for file storage, interaction with the user, etc., and generates a world used by the Workserver. A block diagram of the system is shown in Figure 2.

The workserver itself is shown in more detail in Figure 3. This contains various parts. First there are standard library routines, the task handling system and the LOKI compiler system. The rest of the workserver contains the worlds with which the operator may interact, and the objects which may exist in the world. Associated with these objects are so called daemons which allow the objects to move within the world, and the means whereby the objects may be shown to the user: the graphical and sound blocks shown in Figure 3 are used for this purpose.

Slave processes may be run from the "Workserver" hub, each process being supplied with a communications pipe to talk to the main system. Library routines may be accessed to perform various functions, by passing commands to the workserver, via this pipe system. In this way, models are defined, manipulated, and rendered without any of the model information being held in the slave process. This means that failure of a slave does not crash the LOKI system, and that applications packages written tend to be small.

The slave processes work effectively on "objects" within the Workserver system. An object is an item in the simulated world which is implemented as a data structure which can contain references to a program, geometrical qualities, other objects, and so forth. These objects are contained in an object core, but are not all necessarily being used. The system contains a list of valid worlds, each of which has a number

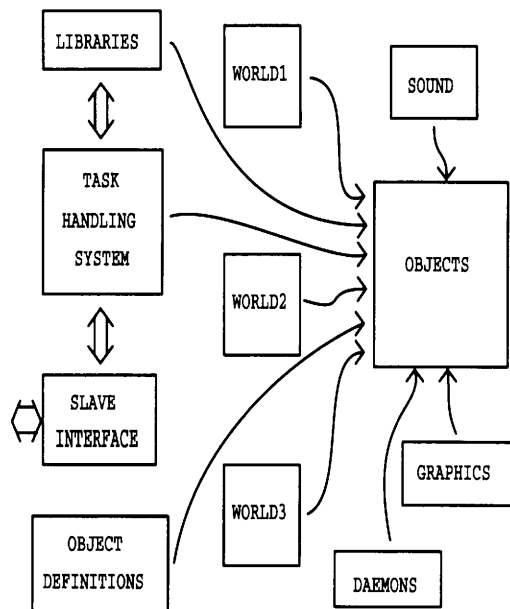


Figure 3: Workserver Block Diagram

of defined objects associated with it. The user is presented with just one world at one time, but may be switched between them at will. Objects may be referenced by more than one world, but may be called within each environment only once (because, for example, an object cannot be in two places at once).

It is possible to define objects as being just geometrical instances, i.e. having a definite shape which is then displayed to the user. This does not cover the problem of assigning behaviour to objects. There are two methods for achieving this; firstly to use the slave processes or applications to control objects directly; or secondly to use the LOKI programming language to assign objects their own programs. This is where the compiler block in Figure 3 is used: definitions of the objects are compiled and stored in the objects section of the workserver. This programming language facilitates most of the necessary behavioural control for a simple virtual reality. In use, the daemons utilise these behaviours to move objects around within the world.

The handling of objects is achieved using an object oriented method. A particular object, say a car, can exist in the world at a particular position, moving at a given speed, etc. However, there may be many cars in the world, sharing various attributes, like appearance, ways in which they move, etc. Thus there is a separate description of these common attributes which is inherited by each object instance, and used by the system to determine the position of the car, its appearance, etc. Further sophistication is allowed: suppose various cars are on a transporter lorry: when that lorry moves, so do the cars: again the inheritance properties allow for such a case.

Slave processes and the workserver are controlled by the computer's own multitasking executive. The object multitasking system is facilitated by a separate, custom arrangement. In this a non pre-emptive/pre-emptive strategy has been implemented. Programs run by objects generally execute a number of instructions, followed by an explicitly or implicitly defined "SYNC" command. This command stops the further processing of this program, and the next object in the environment is handled. If a task executes too many instructions, the Workserver forces a "SYNC" to occur: this is explained below. When all objects have been handled, then the display of the current environ-

ment is updated. This display is then presented to the user. After display presentation, the list of valid objects is then reprocessed, and this is then continued in a cyclical manner.

As display update for the scene should ideally occur twenty five times a second, the above cycle needs to occur evenly, and the individual programs attached to the objects need to pass control as quickly as possible to the next object for a smooth and fast scene display. If, however, a program has been badly written, such that control is not passed on to the next object, then the system could "stick" on that task. To prevent this a "SYNC" is forced after a certain number of instructions has been issued.

In the overall system, it was decided that tight scheduling of tasks was not necessary for the slave applications: for example, the CAD package generating the world does not need to operate in real time. Updating the computer graphics and the model database is however a tight scheduling problem, in order to achieve smooth animation for scenes for the observer. This was the reason that a separate multitasking scheduler was developed. Input handling from the user is also processed by this scheduling system, as smooth input sampling is also necessary for a satisfactory system.

Disk operations, the forking of slave processes or applications, system clean up operations and communications to other tasks on the same computer are dealt with by the native operating system calls.

Implementing LOKI on a UNIX Platform

LOKI bears a close resemblance to most research and commercially available Virtual Reality software in its functionality and general implementation. It can be seen that there are elements within the object system, as well as the graphics and user interactions, which can be said to require regular and fast scheduling. If the size and complexity of the database is to be increased and the system is to remain real time, it will almost certainly require the use of parallel processing. Coarse grained message passing systems have been attempted in other systems, and this seems quite successful.

UNIX scheduling on a standard platform is ill designed for delivering exactly even and equal time slicing to all its constituent processes [Bac86a]: the overall performance degenerates with the platform loading and specific activities like DMA, etc. Real time UNIX kernels exist, which deliver even time slicing but these are expensive and not generally available. As specific interfaces are required for Virtual Reality work, i.e. special graphics systems and input devices, it would seem a good compromise to implement the object oriented program elements on this hardware as well, and place the support elements on the UNIX system.

In the case of the LOKI system, work is underway to implement this method in the following manner. A parallel processing system, based around a dedicated shared memory architecture machine is proposed. This system processes the object oriented elements of the design. Information is then passed first to a dedicated graphics and audio rendering system, and then on from this to the operator. Another dedicated system intercepts input from the user and then passes this to the parallel processing schema.

Applications programs would be held on the UNIX platform, and dealt with as standard UNIX processes. Communication with the model sys-

tem would be facilitated by the use of named pipes [Mic90a], utilising the same form of command system as in the early version of LOKI. A process, similar to a workserver, would exist on the platform to act as receiver for these commands, and to pass information to and from the dedicated hardware. The utilisation of a filing system would be performed through this, as well as providing access to all the standard UNIX resources. With reference to Figure 3, the worlds, the objects, daemons, etc., are on the dedicated hardware, the rest is on the UNIX platform.

This system places all the real-time dependent elements onto dedicated hardware. This creates a system which is somewhat more expensive than a schema just based on the UNIX platform. Unfortunately it is not possible to use just a standard UNIX platform, for the reasons stated, but the compromise proposed here ensures that a large proportion is placed on a standard environment.

For a basic Virtual Reality system, the full complexities of UNIX are not necessary. However, a full VR system and appropriate support, having a three dimensional workstation system requiring the VR functions and all the functions needed by a normal operator, is a different proposition. Here, the highly developed electronic mail facilities, shared filing systems and general network communications, which exist on UNIX [Bou83a], provide exactly the type of facilities needed as part of the complete system.

Conclusions

For Virtual Reality systems, a large amount of specialist software and hardware is required. To date, all systems are based around either customised or non UNIX platforms. The cost of these dedicated systems is high, and for large scale commercial success, would need to be reduced.

If a system similar to the LOKI schema was to be placed onto a UNIX platform, some specialised elements would still be required, providing the necessary interaction with the operator, and suitable real-time facilities. However, many of the elements of the complete system could be placed on the UNIX platform, thereby utilising the existing facilities provided, and hence reducing the development time of the VR system.

References

- [Bac86a] M. J. Bach, *The Design Of The UNIX Operating System*, 1986.
- [Bou83a] S. R. Bourne, *The UNIX System*, Addison-Wesley (1983).
- [Bur91a] Grigore Burdea, Jiachen Zhuang, Ed Roskos, Deborah Silver, and Noshir Langrana, "Direct-drive Force Feedback Control for the Dataglove," in *Proceedings of Euricon '91*, Corfu (1991).
- [Fol87a] J. D. Foley, "Interfaces For Advanced Computing," *Scientific American* (1987).
- [Gri90a] M. P. Griffin and R. J. Mitchell, "Surround Vision Systems: The Role of Object Orientated Programming," in *IEE Colloquium on CYBERNETICS TODAY, Digest 1990/044*, London (1990).

- [Gri91a] M. P. Griffin and R. J. Mitchell, "The Use of Simulation Systems To Control Manually Operated Remote Manipulators, With Long Pure Time Delays," in *Proceedings of Euriscon '91*, Corfu (1991).
- [Gri91b] M. P. Griffin, "A Cybernetic Perspective of Virtual Reality and Telepresence Technology," PhD Thesis, University of Reading (1991).
- [Mic90a] Sun Microsystems, *STREAMS Programming*, Sun Microsystems Inc (1990).
- [Sut68a] I. E. Sutherland, "A Head Mounted Three Dimensional Display," pp. 757-764 in *Proceedings Fall Joint Comp Conf* (1968).

Interactive User Interface Design

The Teleuse Approach

Achim Brede

Bredex

achim@bredex.uucp

Abstract

Designing user interfaces for new applications is not an easy task. Not only the complexity of software components for implementing user interfaces (e.g. like the X Window System and OSF/Motif) have to be considered, human factors also play an important role. Lots of work has been done to implement tools for helping designers and developers to implement user interfaces.

A study of modern computer software came to the result that about 60% effort is needed for the user interface part and only 40% for the application code. This can be measured either in lines of code or time of development.

If one looks closer into the user interface part, it can be divided into the following three portions:

- The static part or the presentation layer
- The dynamic part or the dialog layer
- The API or the application interface layer

The paper will give examples of all three layers and discuss the benefits of each.

The presentation layer can be built by an interface graphical editor (WYSIWYG). The result can be written to a file by generating C code or generating a specific user interface language.

The usage of a dialog manager for specifying the dynamic behaviour is compared to C coding. Finally the paper looks into the application interface. The extensibility with new objects (widgets) is also discussed.

The QEF/QEI Model for Software Component Consistency, Dependency Determination, and Construction Recipe Ordering

David Tilbrook

Sietec Open Systems Division

dt@snitor.uucp

Abstract

This presentation investigates the problem of determining when and if a process within a software construction system should be invoked. Within the UNIX community, the best known approach to this problem is that employed by *make*. *make* invokes a process to update a target file **when** the files on which it depends exist and are themselves up to date, and **if** the target file is older than any of its dependents. This approach has the advantage of being very simple, both to implement and to understand. But it is far from adequate, as is discussed in this presentation.

This presentation analyses the difficulty and importance of ensuring software component consistency and completeness. It defines the problem and discusses its relevance and implications. A model and definition of consistency is presented. Problems with that definition are then examined with respect to tradeoffs and complexity, the concept of versioning, and the difficulties of strict application of the consistency model. Suggestions for circumventing some of the problems presented by the model, such as propagating gratuitous time stamp changes, are presented.

Any consistency model is highly dependent on the derivation and/or expression of the dependencies. Various approaches to both aspects are discussed.

Given models for consistency and dependencies, the problems in determining the order of execution can be discussed.

Finally, strategies for software organization that simplify or improve the performance of the construction process, while ensuring the integrity of the product, are presented.

Many of the solutions and approaches presented in this presentation are based on the author's systems for software construction and version management, Quod Erat Faciendum (*qef*) and Quod Erat Inveniendum (*qei*) respectively.

Give a Process to your Drivers!

François Armand

Chorus systèmes, France

francois@chorus.fr

Abstract

This paper presents how the modular architecture used for the CHORUS/MiX V.3.2 system enables system writers to encapsulate UNIX device drivers within UNIX processes, and the possibilities offered by this feature.

The CHORUS architecture is designed to support a new generation of open and distributed operating systems. A microkernel provides generic services allowing servers that cooperate within subsystems to offer standard interfaces: a UNIX SVR3.2 interface is available and a UNIX SVR4 interface is under development. The CHORUS microkernel provides services to manipulate actors and threads (memory management, scheduling and so on). It also offers a distributed Inter-Process Communication (IPC) facility and services to dynamically connect user provided functions to hardware interrupts and traps.

CHORUS/MiX V.3.2 is implemented as a set of the following servers: a Process Manager, a File Manager, a Terminal Manager and a Socket Manager (to provide BSD sockets). In addition to standard UNIX services (which have been transparently extended to distribution), one can also use CHORUS services to take advantage of CHORUS real-time features and multi-threaded processes.

The CHORUS/MiX V.3.2 has been experimentally enriched by the introduction of a new kind of server named a "Driver Actor" (DA). A DA allows one to encapsulate device drivers within independent actors. This new kind of server has been (experimentally) used to separate the disk driver from the File Manager.

As a consequence, this separation removes the constraint placed on the File Manager to reside into the machine supervisor address space, as privileged instructions are executed only by the driver itself. Thus, the File Manager can run as a UNIX process either in user or supervisor address space, communicating with the driver through CHORUS IPC accessible at UNIX interface level. This allows one to take advantage of standard debuggers, such as sdb and gdb, to debug the File Manager.

Using CHORUS IPC between the File Manager and the Driver Actor permits them to be transparently distributed over a network of processors. Therefore the driver may be loaded on a dedicated board while providing this driver with an environment compatible with that of a UNIX native kernel. This kind of configuration is used within the Multi-Works Esprit project.

In addition, the CHORUS/MiX subsystem permits one to dynamically load processes running in the supervisor address space usually reserved for use only by the UNIX kernel. Thus from a shell, one can dynamically load, locally or remotely, UNIX drivers as if they were "common" processes.

1. CHORUS/MiX

1.1. CHORUS Architecture

A CHORUS System is composed of a small-sized **Nucleus** and of possibly several **System Servers** that cooperate in the context of **subsystems** to provide a coherent set of services and user interface. A detailed description of the CHORUS system can be found in [Roz88a]. Some other systems have adopted similar architectures: Mach [Acc86a], V-system [Che88a] and Amoeba [Mul87a].

1.2. CHORUS Kernel

The CHORUS kernel provides the following basic abstractions:

- The **actor** defines an address space that can be either in user space or in supervisor space. In the later case, the actor has access to the privileged execution mode of the hardware. User actors have a protected address space.
- One or more **threads** (*light weight processes*) can run simultaneously within the same actor. They can communicate using the memory of the actor if they run in the same actor.
- Otherwise, they can communicate through the CHORUS IPC that enables them to exchange **messages** through **ports** designed by global unique identifiers.

A message is composed of a (optional) **body** and an (optional) **annex**. Annex size is fixed (64 bytes currently). Body size is variable.

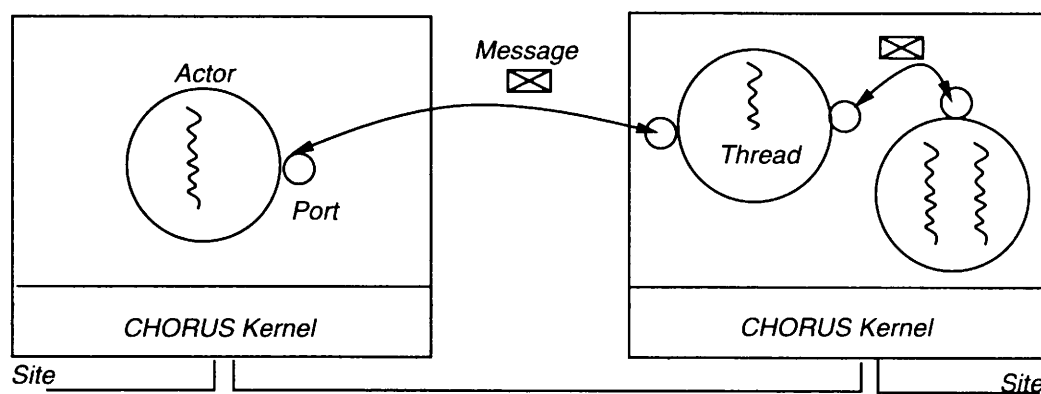


Figure 1: CHORUS basic abstractions

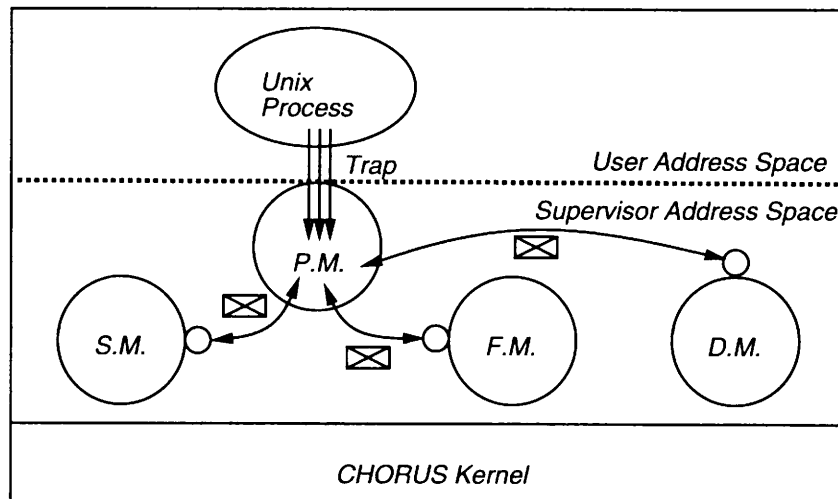


Figure 2: MiX subsystem structure

1.3. The MiX subsystem

MiX is a CHORUS subsystem providing a UNIX interface that is compatible with UNIX SVR3.2. It is binary compatible with SCO on AT/386 machines. Herrmann's paper provides more details on the implementation of the MiX subsystem. It is composed of the following servers:

- The **Process Manager (P.M.)** provides the UNIX interface to processes. It implements services for process management such as the creation and destruction of processes or the sending of signals. It manages the system context of each process that runs on its site. When the P.M. is not able to serve a UNIX system call by itself, it calls other servers, as appropriate, using CHORUS IPC.
- The **File Manager (F.M.)** performs file management services.
- The **Device Manager (D.M.)** manages asynchronous lines, keyboards, pseudo-ttys, etc and implements the UNIX line disciplines.
- The **Socket Manager (S.M.)** implements BSD 4.3 socket services, providing access to TCP/IP protocols.

For performance reasons and because they manage traps or interrupts these servers run in system space.

UNIX processes are implemented by the P.M. on top of the abstractions provided by the CHORUS kernel. Basically, a UNIX process is composed of one actor within which one thread is running.

1.4. Extensions

One goal of CHORUS/MiX is to offer the user new services:

- Providing multithreaded processes,
- Transparently extending traditional UNIX services to distribution, thus providing a distributed file system, remote execution of processes and distributed signals,
- Providing CHORUS IPC at the UNIX interface level,
- Providing access to the priority-based preemptive scheduling,

- Providing the ability to dynamically execute processes in system space.

More details on these extensions can be found in [Arm89a] and [Arm90a]. The later possibility is extremely important to the rest of this paper. The system space is partitioned between the CHORUS kernel itself and CHORUS/MiX servers. The free space is available for the user to load processes dynamically in system space. Loading processes into system space is desirable to achieve better performance or gain access to the privileged mode of execution of the hardware. This ability is restricted to the UNIX super-user.

2. History and Related Works

2.1. CHORUS/MiX and the drivers

CHORUS/MiX has always tried emphasize on modularity by splitting the main UNIX kernel functions into independent modules. In the current CHORUS/MiX V.3.2, however, one basic service has not been “separated” in such a way – device drivers:

- Drivers such as disks, floppies and tapes, offering a “block interface” are embedded within the File Manager.
- Drivers for terminals, keyboard/mouse offering only a “character interface” are included in the Device Manager.

It was deemed more important to separate file management from process management and, thus, to be able to distribute them over a network, than to separate a disk driver from the file management as the management of files is tightly coupled with the management of a disk.

However, CHORUS/MiX has already experimented with partial driver separation in its previous versions:

- *CHORUS V2*: In version V2, UNIX servers ran as “user mode” actors, and thus had no access to privileged hardware instructions. I/O operations were performed by the kernel upon reception of request messages sent by the UNIX servers. As it was a very low-level interface, UNIX drivers had to be modified. Moreover, this interface was too dependent on the machine on which the system ran.

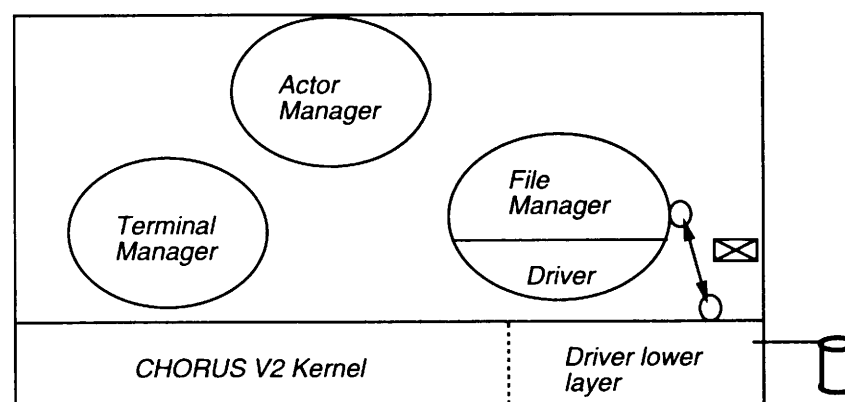


Figure 3: I/O Management in CHORUS V2

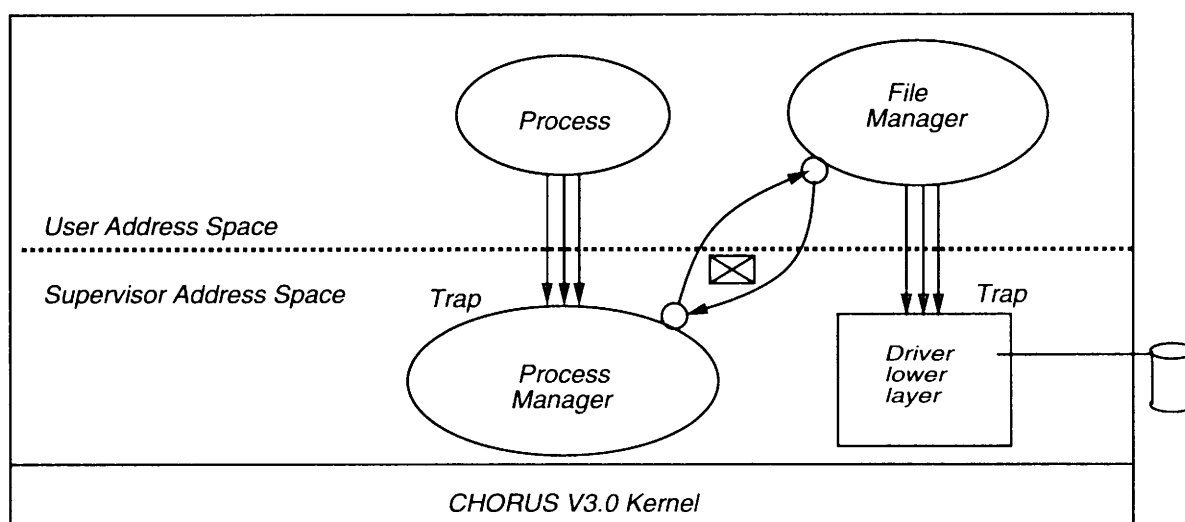


Figure 4: Previous I/O Management in CHORUS V3.0

- *CHORUS V3.0*: In the first implementation of CHORUS/MiX on Bull SPS 7/70, the File Manager executed in user mode. The sequences of instructions of the disk driver that contained privileged instructions were separated in routines running in system mode that were dynamically connected to a trap, during driver initialization. Once again, this approach involved a specific adaptation for each driver being ported.

None of these two attempts fit with the “natural” driver interface provided in UNIX kernels.

2.2. Related Works

- *MACH*: MACH 3.0 [Gol90a] also provides a UNIX implementation based on a microkernel. The UNIX emulation runs in user mode within a “single” server communicating with the drivers that are linked together with the microkernel. Communication is achieved using MACH IPC. This is similar to CHORUS V2 with the important distinction that the interface used in MACH is more natural. However, there is no provision for the dynamic loading and unloading of device drivers as drivers are embedded within the microkernel.
- *V System*: In V [Che90a] device drivers are embedded within a pseudo process that executes in the V kernel itself. Access to the devices is achieved by means of a UIO (Uniform Input/Output) interface mapped on the message based V IPC. The UIO interface is generic enough to allow access to any kind of servers, such as file servers, mail servers or device servers, in a quite uniform fashion. More details about the UIO mechanism can be found in [Che84a]. An application gets access to the file server using the UIO interface, possibly hidden within a library. In turn, the file server accesses the device driver using the same interface. However, as in MACH, it seems that there is no provision for dynamically loading a device driver.
- *SunOS*: On a Sun i386, SunOS allows one to dynamically load a driver within the kernel using a special command “modload”.

But the driver's writer needs to provide some "wrapper" code in addition to the driver itself. To our knowledge this is the system that achieves the functionality most similar to CHORUS/MiX. Unfortunately, this ability is restricted to Sun i386 machines.

2.3. Why do we need Driver Actor ?

2.3.1. Powerful Device Boards: MultiWorks Example

More and more machines provide powerful boards (processor and plenty of memory) to connect devices. The European project Multi-Works builds such a multi-media workstation.

The station is built around an EISA bus with a Main Processing Unit, based on a Intel CP486, and several boards, called Intelligent Peripheral Adaptors. I.P.A. are dedicated to device management and based on the chip Acorn ARM3. An I.P.A. board has 1, 4 or 16 megabytes of RAM memory. One of these board is used to manage a "Disk Array" through a SCSI bus.

The CHORUS kernel runs on both the CP486 and I.P.A. providing IPC over the EISA bus. In addition, a full CHORUS/MiX subsystem runs on the CP486 board. Thus, CHORUS/MiX has to manage a disk which is connected to an IPA. One could have taken advantage of the CHORUS/MiX modularity by running the File Manager on the I.P.A. But, this solution has a major drawback: a lot of UNIX requests that do not require a disk access, such as operations on pipes or on cached data, would generate undesirable accesses to the EISA bus. The additional bus references would significantly reduce the available bus bandwidth and would negatively impact the performance of both the system and the applications.

It seems more appropriate to execute only the disk driver on the I.P.A. and to let the File Manager and its buffer cache reside on the main processor.

2.3.2. Co-Existing Sub-Systems

The CHORUS kernel allows several subsystems to run simultaneously on a same machine. In such a case, various subsystems share the access to the processor and to the main memory, relying on the services provided by the CHORUS kernel. Access to other physical resources, such as devices, is not managed by the kernel, but is on the responsibility of the subsystems themselves.

Rather than partitioning the devices between the subsystems, it is more convenient to be able to share the access of a device between multiple subsystems. To achieve this, the device driver must be able to communicate with each of the subsystems. Splitting the drivers from the UNIX servers that need them, is a first step in that direction.

2.3.3. Other Needs

The encapsulation of drivers within independent servers presents some other interesting characteristics:

- The combination of this encapsulation with the ability to dynamically load programs in system space enables us to dynamically load drivers without stopping the system when adding a new device type to the machine. This can also help to reduce the size of the resident set of system servers by removing drivers that are

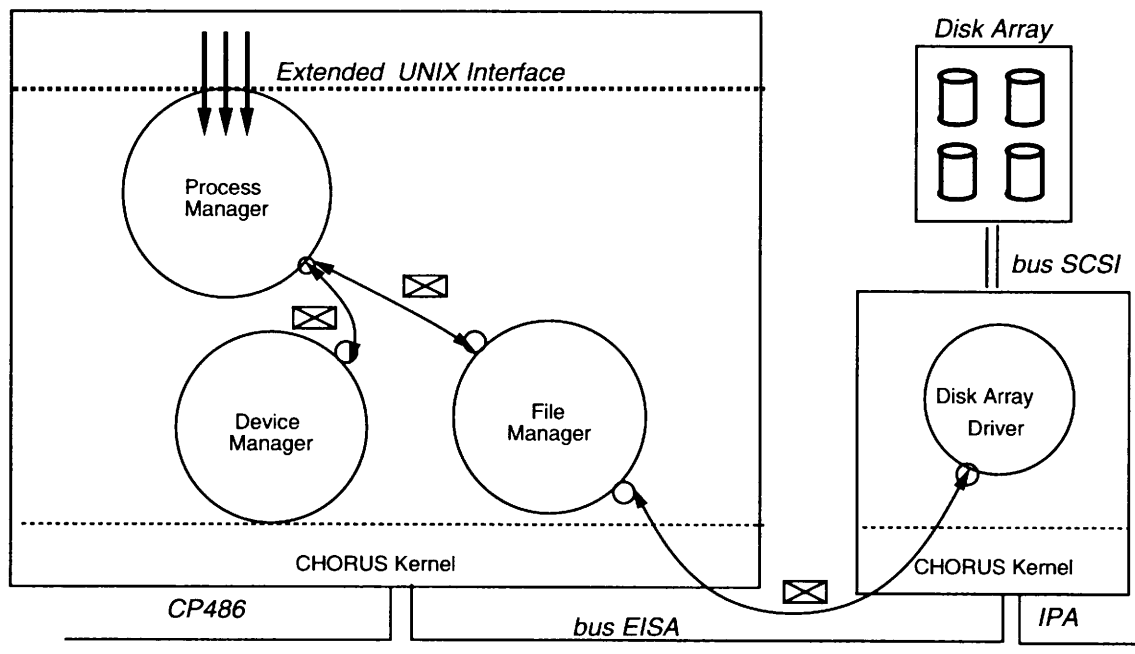


Figure 5: MultiWorks Multimedia Workstation Architecture

not frequently used from the default boot file. These drivers being loaded and unloaded on demand either manually or automatically.

- Servers such as the File Manager, being "cleansed" of privileged hardware instructions can be executed in user mode with a loss in performance the. This configuration can be used to debug the CHORUS/MiX servers as "normal" UNIX processes, using standard debuggers such as sdb, dbx or gdb.

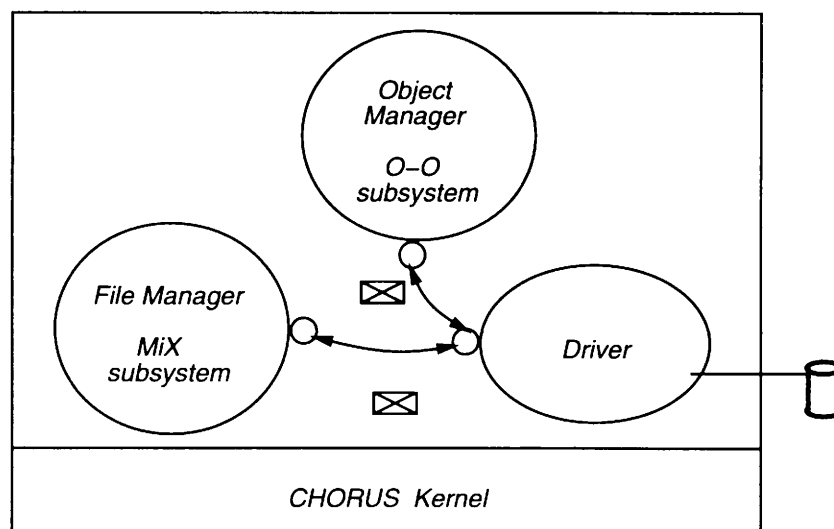


Figure 6: Sharing a Driver between two subsystems

3. General Structure of an Actor Driver

3.1. Exported Services

Services provided by a driver are well-defined within UNIX kernels, although this definition may vary from one system to another. Nevertheless, basic services remain the same, they are summarized below.

UNIX distinguishes between drivers that offer a “*block interface*” and those that offer a “*character interface*”.

Block Interface

Such a driver has to provide the following services:

- ◆ *drvopen* (device, read/write, open_type)
- ◆ *drvclose* (device, read/write, open_type)
- ◆ *drvstrategy* (buffer_header)

The most important function is the one called “*strategy*” which is used to process the I/O's on the device. The communication between the upper layers of the file system and the driver is achieved through a “*buffer header*”. The useful information for the driver is contained in the following fields of such a header:

- ◆ Device number on which the I/O must be performed
- ◆ Block number to be read/written
- ◆ Size of the I/O
- ◆ Address in memory to store/find the data
- ◆ Flags to describe the I/O (read/write, synchronous /asynchronous, etc)

Character Interface

Such a driver must export the following services:

- ◆ *drvopen* (device, read/write)
- ◆ *drvclose* (device, read/write)
- ◆ *drvread* (device)
- ◆ *drvwrite* (device)
- ◆ *drvioctl* (device, cmd, arg)

The description of the I/O to be performed is defined in a “*uio*” structure within the system context of the process.

3.2. Imported Services

Unfortunately, no standard or guide defines the services provided by a UNIX SVR3.2 kernel that can be used by a device driver. Thus, a driver can potentially use all of the kernel functions and data structures. However, in practice device drivers use a small set of services and data structures.[†] It is, thus, possible to list the most common needs of a driver:

- sleep/wakeup: wait for event, reactivate a process when the event occurs,
- spl0, spl7, splx, etc: mask/unmask interrupts,

[†] UNIX SVR4 defines precisely what can be used by a device driver within a “DDI/DK1 Reference Manual”.

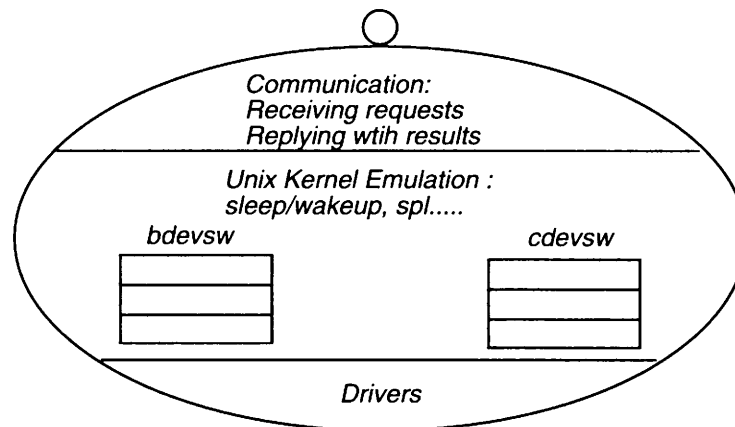


Figure 7: Driver Actor Structure

- timeout, untimeout: activate a function after a delay, cancel the activation,
- getblk, geteblk: allocate a buffer and a buffer header, etc,
- biowait, biodone: wait for the completion of an I/O, indicate the completion of an I/O.

All these services have already been emulated within CHORUS/MiX servers. Thus, it is straightforward to re-use such functions within a Driver Actor.

3.3. Layout and Communication

A Driver Actor is similar to other servers in that it is a multi-threaded server that receives requests on a port. As the real device driver is no longer embedded within the server, it is necessary to replace it by a "stub-driver" that conforms to the UNIX driver interface and that sets up request messages, sends them to the appropriate Driver Actor, and waits for the corresponding reply. This structure implies that the stub-driver is able to retrieve the UI of the port of the Driver Actor given a major number.

Upon reception of a request, the Driver Actor needs to unmarshall the message and to build a context similar to that which would have existed within a monolithic UNIX kernel so that the appropriate driver function can be left unmodified. Upon completion of the request, the Driver Actor must marshall the results in a reply message that will be interpreted by the stub-driver.

4. A Sample Case: The Disk Driver Actor

4.1. Description

This architecture has been used within CHORUS/MiX V.3.2 to separate the disk driver from the File Manager. The current implementation deals only with the block interface, the character interface of the disk driver will be implemented later. The contents of the messages being exchanged derive directly from the UNIX driver interface described previously.

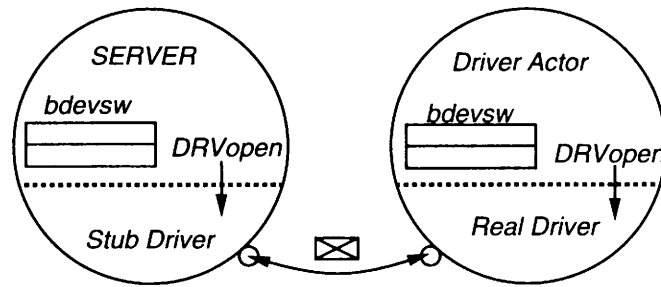


Figure 8: Communication Server ↔ Driver Actor

The “write” request passed to the stub-driver by the File Manager is in fact, according to the block interface, a “strategy” request with a write flag. The stub-driver receives a buffer header from which it extracts the information needed by the real driver and copies it into a message annex. Data to be written to the disk are simultaneously transmitted within the message body.

Thus, a synchronous write request is performed as follows:

- The *strategy* routine of the stub-driver is invoked by the File Manager. It receives a buffer header describing the I/O to be performed and pointing to the buffer containing the data to be written.
- The *stub-strategy* routine sets up a message annex with the information contained in the buffer header. The message body is the buffer itself. No copy of data is performed by the stub-driver. Then, the stub driver invokes the real driver by means of the remote procedure call offered by the CHORUS kernel.
- The CHORUS IPC Manager will perform the copy of the message body and will make the message available to the Driver Actor.
- The Driver Actor receives the message and decodes the service code found at the beginning of the annex. Its only work is to set up a request context emulating a UNIX kernel environment, and to set up a buffer header from the information received in the

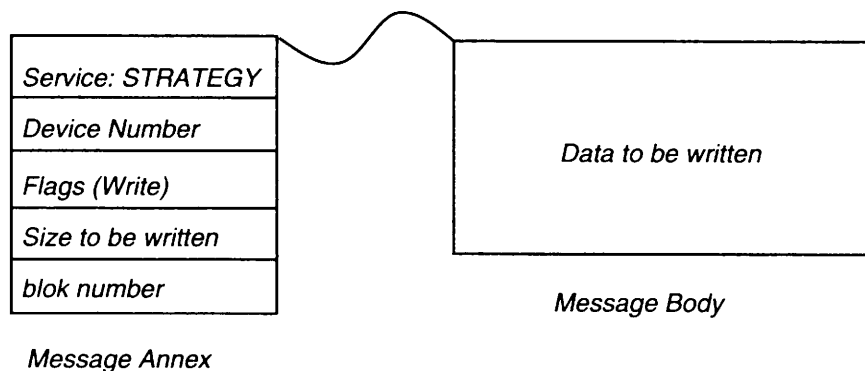


Figure 9: Disk Write Message Request

message annex. The pointer to the buffer containing the data is set to the body of the received message.

- Then, the real strategy routine is invoked. The Driver Actor then waits for the I/O completion, using *biowait*. When the I/O is completed, the real driver will wake up the request, using *biodone*, permitting the Driver Actor to reply to the File Manager.

Read requests are processed in a similar fashion.

Two important issues need more explanation:

Data Transfer

When the Driver Actor and the File Manager are on the same site, the modularity implies that an additional copy has to be performed. Of course, that copy is not necessary when the driver is running within the File Manager. The UNIX kernel algorithms guarantee that while an I/O is being processed, the buffer cannot be accessed by any other process. One could take advantage of this "property" to attribute the buffer to the driver for the duration of the I/O. Unfortunately, on AT/386 machines the page size is 4Kbytes and System V buffers are 1Kbytes long. Thus, it would be necessary either to place only one buffer in a page, wasting a lot of physical memory, or to lock four buffers at a time implying more complex synchronization and more contention in the File Manager. With file systems using 4Kbytes buffer, such as BSD or SVR4, this could be implemented.

Synchronous and Asynchronous Processing

In a UNIX kernel as well as in the CHORUS/MiX File Manager, an I/O request can be processed either synchronously or asynchronously.

- ♦ *Synchronous requests.* These kinds of requests are started by the invocation of the *strategy* routine which returns immediately. The completion of the I/O is awaited using the *biowait* function, putting the current process to sleep. Thus the duration of the I/O can be used to give the main processor to any runnable process. This behaviour is emulated by releasing the processor in the File Manager when invoking the Actor Driver. When the reply is received, the processor is re-acquired, and the stub-driver returns from its strategy routine. The following call to *biowait* that is performed later by the File Manager will just check that the I/O is done without blocking the process.
- ♦ *Asynchronous requests.* One of the characteristics of UNIX file systems is their use of asynchronous I/O operations such as deferred writes and read ahead to get better performance. In such cases, the I/O is started as usual by invocation of the strategy routine but as its result is not necessary to the current process, its completion is not awaited. The completion of the I/O is just marked within the buffer header.

The protocol between the File Manager and the Driver Actor described previously has been modified to satisfy this need. Upon reception of an asynchronous request the Driver Actor answers immediately, enabling the File Manager to continue its work. When the I/O is done, the Driver Actor sends an asynchronous message to the File Manager denoting that the I/O has completed. In addition,

a completed read request carries the data. In essence, this mechanism is similar to that of a software interrupt. In order to process these asynchronous messages from the Driver Actor, the stub-driver needs to create a thread that will process them.

4.2. Usage: Dynamic Reconfiguration

In order to validate the above design, we have imagined and successfully demonstrated a scenario in which the hardware configuration evolves. Thus, it is up to the system software to follow the evolution!

The initial configuration is composed of one autonomous machine with a processor, some memory, a disk, some terminals and a network interface. Another machine has the same configuration except that it has no disk. The first machine runs a full CHORUS/MiX system consisting of the Kernel, PM, FM, DM and SM. On the diskless machine a CHORUS/MiX system without File Manager is loaded. For demonstrational purposes, the two machines are COMPAQ 386; the disk of one of them being not used. The distribution provided by CHORUS enables one to use the diskless machine as a second processor, and thus to balance the load between the two processors. Next, the scenario script states that as the user's needs are increasing the diskless machine is equipped with a local disk. One supposes the hardware "smart" enough to allow this addition without stopping the system. Thus, one can remotely load a disk driver, from the first machine, within the supervisor address space of the second machine. This allows one to initialize the new disk and to copy on it the files needed by an autonomous machine. Finally, once the new disk has been initialized, one can remotely load on the second machine a File Manager running as a UNIX user process. The Process Manager will recognize the existence of a local File Manager and will start the execution of the "/etc/init" process loaded from the new disk. The second machine will then be completely autonomous. As the File Manager is being executed as a "normal" UNIX process, one can use UNIX debuggers to debug it, although some "obvious" precautions must be taken. When a process

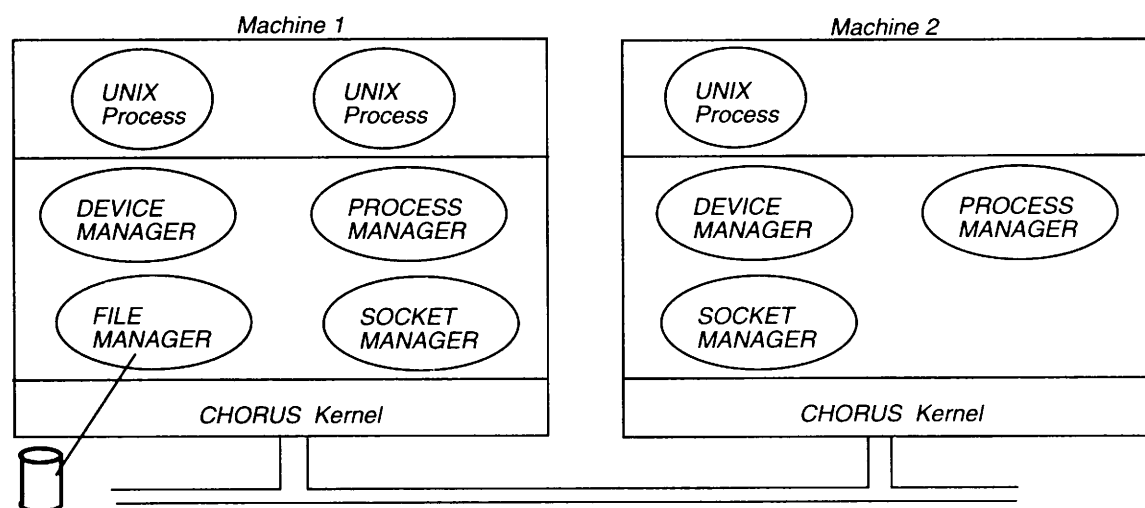


Figure 10: Using a diskless machine

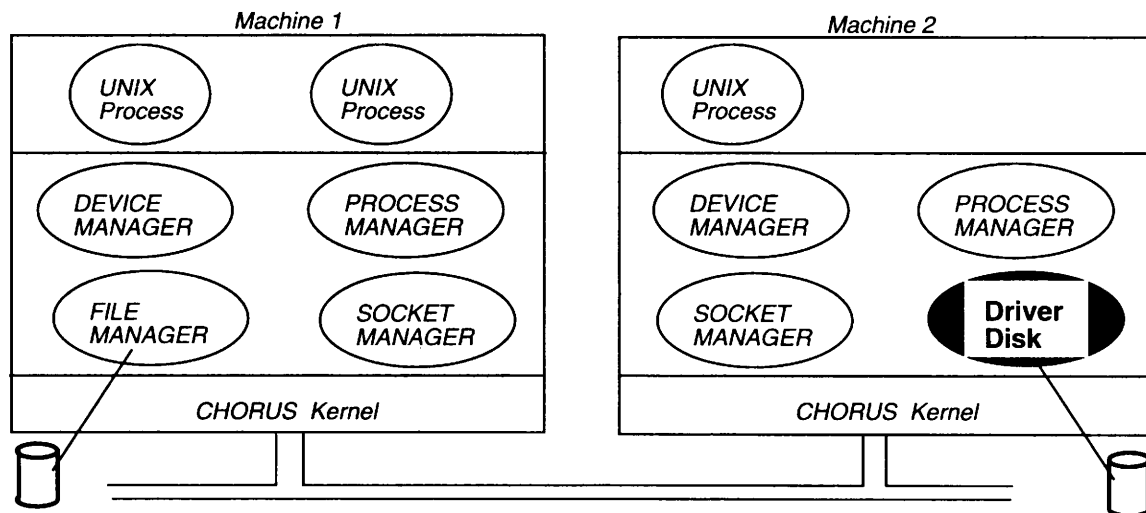


Figure 11: Dynamically Loading a Disk Driver

	"Standard" MiX	FM as a user process	FM as a supervisor process
creat	60 creat/sec	58 creat/sec	59 creat/sec
open	443 open/sec	355 open/sec	428 open/sec
write 1k	60 Kbytes/sec	43 Kbytes/sec	49 Kbytes/sec
read 1k	198 Kbytes/sec	90 Kbytes/sec	110 Kbytes/sec

Table 1: Performance figures

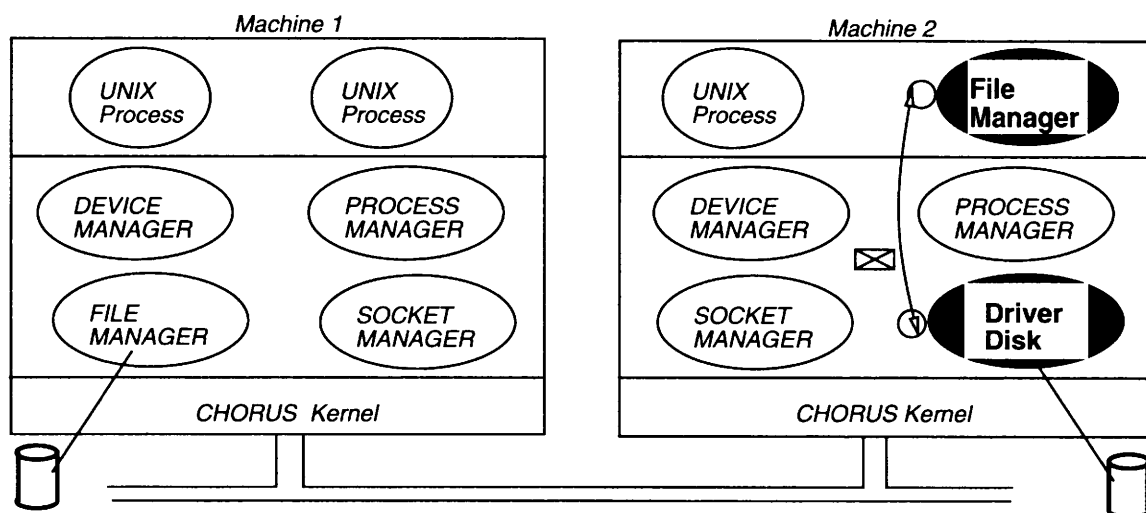


Figure 12: Dynamically Loading a File Manager

being debugged encounters a "breakpoint" it is stopped by the system. When the process being stopped is the File Manager, the requests it receives will not be processed until its execution will be resumed by the debugger. Note that some of these requests will originate from the debugger itself, if it is running locally. Thus, it is necessary to make the File Manager sources available on another machine and to run the debugger from another machine to avoid deadlock.

4.3. Performance

In order to appreciate the effect of such an architecture on the system's performance we have run a set of benchmarks in various configurations:

- Standard CHORUS/MiX with the File Manager and the disk driver linked together within one actor.
- File Manager running as UNIX process in user mode, and the disk driver running as a UNIX process in supervisor mode (as in the demonstration presented above).
- File Manager and driver disk both running as UNIX processes in supervisor mode.

The benchmarks run are briefly described:

- **creat**: measures the time needed for the pair *creat(2)/close(2)*.
- **open**: measures the time consumed by the pair *open(2)/close(2)*.
- **writelk**: measures the time consumed to write 2 megabytes in a regular file 1 kilobyte at a time.
- **readlk**: measures the time consumed to read 2 megabytes from a regular file 1 kilobyte at a time.

These benchmarks have been run on a COMPAQ386 running at 20 MHz with 5 Megabytes of main memory. The results are listed in Table 1. The case where the File Manager runs in user space is, of course, slower because data must be copied from the benchmark process to the File Manager, which is also a user process. Such a copy implies a memory context switch which is unnecessary when the copy is done from user space to supervisor space.

The stability of the "*creat*" test is probably due to the fact file creation implies synchronous writes to the disk to write the inode. Thus, this test is mostly limited by the disk transfer rate and not by the software algorithms.

The "*open*" test consist essentially in copying a pathname between the user process and the File Manager. The user mode FM is slower because of the extra memory context switch. The two other configurations where the FM executes in supervisor space achieve similar performance. The situation where the disk driver runs outside of the FM is penalized by the first open when it is necessary to load blocks from the disk.

The most important differences are observed on read/write operations. The difference between the standard MiX and the case "FM as a supervisor process" with a separated driver disk is due to two main reasons:

- First, an extra copy is performed for all disk blocks being moved. Future developments of the Disk Driver Actor will solve this handicap by using the light weight remote procedure call mechanism provided by the CHORUS kernel.
- Second, the processing of asynchronous requests is not yet fully operational and thus has not been used. All requests are being

processed synchronously, the performance gain from the deferred writes and the read-ahead has been lost.

Nevertheless, these figures prove the correctness of the CHORUS approach that consist in enabling servers to reside in supervisor space to achieve better performance.

5. Conclusion

We have been successful in splitting the disk driver from the CHORUS/MiX File Manager. These two servers have been encapsulated within UNIX processes. The disk driver can be loaded dynamically into the supervisor address space.

It has been possible to prototype all of this very quickly on top of CHORUS/MiX due to the pre-existing modularity and to the power of the tools provided by the CHORUS kernel.

In addition to the work in progress already mentioned, this prototype will enable us to implement the automatic loading of a driver upon the first open of a device, the File Manager being the parent process of the driver. The Disk Driver Actor will also be delivered to the MultiWorks project.

6. Acknowledgements

I would like to thank people that have contributed to the success of this experience: Roland Dirlwanger, Frédéric Herrmann, Denis Métral-Charvet, Didier Poirot, Marc Rozier and François Saint-Lu. I would also like to thank people that helped me with much worthy advices while writing this paper: Joëlle Madec, Allan Bricker, Michel Gien and Marc Guillemont.

References

- [Acc86a] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Summer Conference Proceedings*, USENIX Association (1986).
- [Arm89a] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier, "Revolution 89, or Distributing UNIX Brings it Back to its Original Virtues," pp. 153-174 in *Proc. of Workshop on Experiences with Building Distributed (and Multiprocessor) Systems*, Ft. Lauderdale, FL (5-6 October 1989). Chorus systemes Technical Report CS/TR-89-36.1
- [Arm90a] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier, "Multi-threaded Processes in Chorus/MIX," pp. 1-13 in *Proc. of EUUG Spring'90 Conference*, Munich, Germany (23-27 April 1990). Chorus systèmes Technical Report CS/TR-89-37.3
- [Che88a] David Cheriton, "The V Distributed System," *Communications of the ACM* **31**(3), pp. 314-333, Vsyst (March 1988).

- [Che84a] D. R. Cheriton, "A Uniform I/O Interface and Protocol for Distributed Systems," Research Report, Stanford U. (Dec 1984). Z/2 SYS1201
- [Che90a] David R. Cheriton, Gregory R. Whitehead, and Edward W. Sznyter, "Binary Emulation of UNIX using the V Kernel," pp. 73-86 in *Proc. of Summer 1990 USENIX Conference*, USENIX, Anaheim, CA (June 11-15, 1990). CS/EX-90-284 X90284
- [Gol90a] Davic Golub, Randall Dean, Alessandro Forin, and Richard Rashid, "UNIX as an Application Program," pp. 87-96 in *Proc. of Summer 1990 USENIX Conference*, USENIX, Anaheim, CA (June 11-15, 1990). CS/EX-90-285 X90285
- [Mul87a] S. J. Mullender (Editor), *The Amoeba Distributed Operating System: Selected papers 1984 - 1987*, CWI tract 41, Amsterdam (1987).
- [Roz88a] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser, "CHORUS Distributed Operating Systems," *Computing Systems Journal* 1(4), pp. 305-370, The Usenix Association, Chorus systemes Technical Report CS/TR-88-7 (December 1988).

Multimedia Synchronization and UNIX

– or –

If Multimedia Support is the Problem, Is UNIX the Solution?

Dick C.A. Bulterman

Guido van Rossum Dik Winter

CWI: Centrum voor Wiskunde en Informatica

Amsterdam, The Netherlands

dcab@cw.nl

Abstract

This paper considers the role of UNIX in supporting multimedia applications. In particular, we consider the ability of the UNIX operating system (in general) and the UNIX I/O system (in particular) to support the synchronization of a number of high-bandwidth data sets that must be combined to support generalized multimedia systems. The paper is divided into three main sections. The first section reviews the requirements and characteristics that are inherent to multimedia applications. The second section reviews the facilities provided by UNIX and the UNIX I/O model. The third section contrasts the needs of multimedia and the abilities of UNIX to support these needs, with special attention paid to UNIX's problem aspects. We close by sketching an approach we are studying to solve the multimedia processing problem: the use of a distributed operating system to provide a separate data and processing management layer for multimedia information.

1. Introduction

If a definition for "Multimedia" were to exist in an ultra-modern dictionary of computer jargon, the entry might read as follows:

mul-ti-me-di-a <buzzword; adj.>: A property of applications software that allows for the mixed use of several (chiefly output) media – such as sound, video, text, image and graphic data – in a manner that makes an unsuspecting user think that something extraordinary is happening on an otherwise conventional computing system.

Examples: multimedia mail, multimedia documents, multimedia research.

See also: *Clothes, Emperor's New.*

While serious multimedia researchers (such as the authors) would take exception to the tone of this definition, few would argue against the

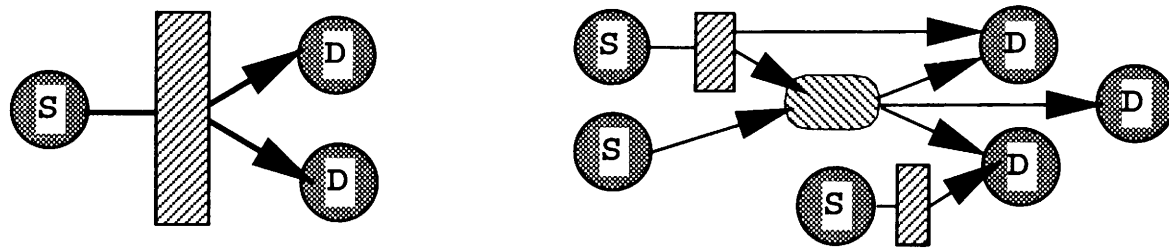


Figure 1: Two examples of multimedia data processing

description of multimedia systems as consisting of applications software that uses mixed forms of media as a basis for presenting information. Unfortunately, even this definition has a serious flaw: it places an emphasis on *applications*; this obscures the crucial role that other layers in a computing system play in providing multimedia support.

Support for multimedia data manipulation can exist at three levels: in the applications code that allows the user to access and control the flow of information, in the operating systems code through the use of device drivers and scheduling software, or at the hardware controller level. Activity at the controller level is currently constrained to providing or accepting data under control of the other layers in the system. The partitioning of activities between the operating system and the applications layer depends on the complexity of the system being supported (see Figure 1). In simple multimedia systems (Figure 1a), data consists of raw information that must be moved from one place to another at a specified rate. This is FAX-style multimedia: the information itself is uninterpreted, making content-based manipulation of the data impossible. In this case, the application layer may request that the operating system start a transfer, with most subsequent device actions controlled by a (set of) device drivers. In less simple forms of multimedia systems (Figure 1b), each of the various data paths may consist of structured information that can be manipulated individually or as a whole, and in which the *meaning* of the information may influence its processing. While the characteristics of multimedia systems are as different as the number of separate applications systems, we can generally conclude that the more manipulation of information that is required, the higher the degree of applications support that will be necessary in the system. (See [Pro89a] for a discussion of multimedia and workstations.)

At first glance, multimedia support seems to be nothing more than providing UNIX I/O drivers for a set of new device controllers for a variety of media types. The most fundamental difference between multimedia systems and other types of input/output transformations of data, however, is the fact that multimedia data is inherently multi-dimensional. This means multimedia information consists of a number of *related* components that must be gathered and/or scattered to/from a variety of devices, under a set of implicit or explicit synchronization constraints. This synchronization needs to be supported by a system layer that has access to the underlying hardware (for integrating device I/O) and is accessible to the application code (which ultimately controls the logical information flow). The synchronization also needs to be supported by the general system-wide process scheduling mechanism to ensure that all of the components at all levels are active at the desired times.

In this paper, we consider the role of the UNIX kernel in providing multimedia support. Our purpose is to consider the needs of multimedia systems and to contrast these against the facilities provided in UNIX to support these needs. While several aspects of multimedia data manipulation are considered, we focus primarily on the synchronization of independent data paths (such as separate audio and image paths) within a multimedia application. We do this because, unlike application-specific user programs or media-specific hardware controllers, it is the operating system that currently provides the only programmable place for providing the efficient synchronization primitives that both the hardware and applications software need to allow complex multimedia data interaction.

We start our consideration of the relationship between multimedia support and operating systems by discussing two classes of multimedia data models: multimedia data location models and multimedia data synchronization models. We then review the location and synchronization facilities provided by the UNIX I/O model. We then compare the needs and expressive capabilities of both UNIX and generic multimedia systems to see if UNIX is "multimedia ready." We conclude by offering an alternative approach that we are studying as part of the CWI/Multimedia project [Bul90a, Bul91a] to better support broad classes of multimedia applications. Note that while our general observations on the utility of UNIX for long-term multimedia support are not positive, we hope to provide more than an exercise in UNIX-bashing; instead, we hope to highlight a set of problems that we feel will increase in importance as the use of multimedia expands.

2. Two Classes of Models for Data Interaction in Multimedia Systems

The broadest notion of the term multimedia is simply the use of several different types of data formats to encode and/or present information. In considering methods of supporting multimedia, two issues immediately arise: first, it is important to know where the multimedia data comes from (as well as knowing where it will need to be sent), and second, it is important to consider which types of synchronization is required to ensure that the multiple data streams interact in the desired manner. The first issue deals with data location models; these models determine the sources and destinations of data streams.[†] The second issue deals with the relationships among data streams; these relationships can be the property of an application or of the data itself. Both of these issues are considered in the following sections.

2.1. Multimedia Data Location Models

The location of multimedia information determines the amount of operating systems support that are required to gather scattered data for possible processing, and to pass that data on to a set of output devices. There are four general models that can describe the sources and destinations for multimedia information. These are reviewed in Figure 2.

- *Local Single Source:* This location model has all data originating at a single source. An example is a CD-ROM that contains sound, text and picture information. Information is fetched in blocks

[†] Note that *streams* is used here in terms of a collection of information, not in terms of a particular device driver implementation technique.

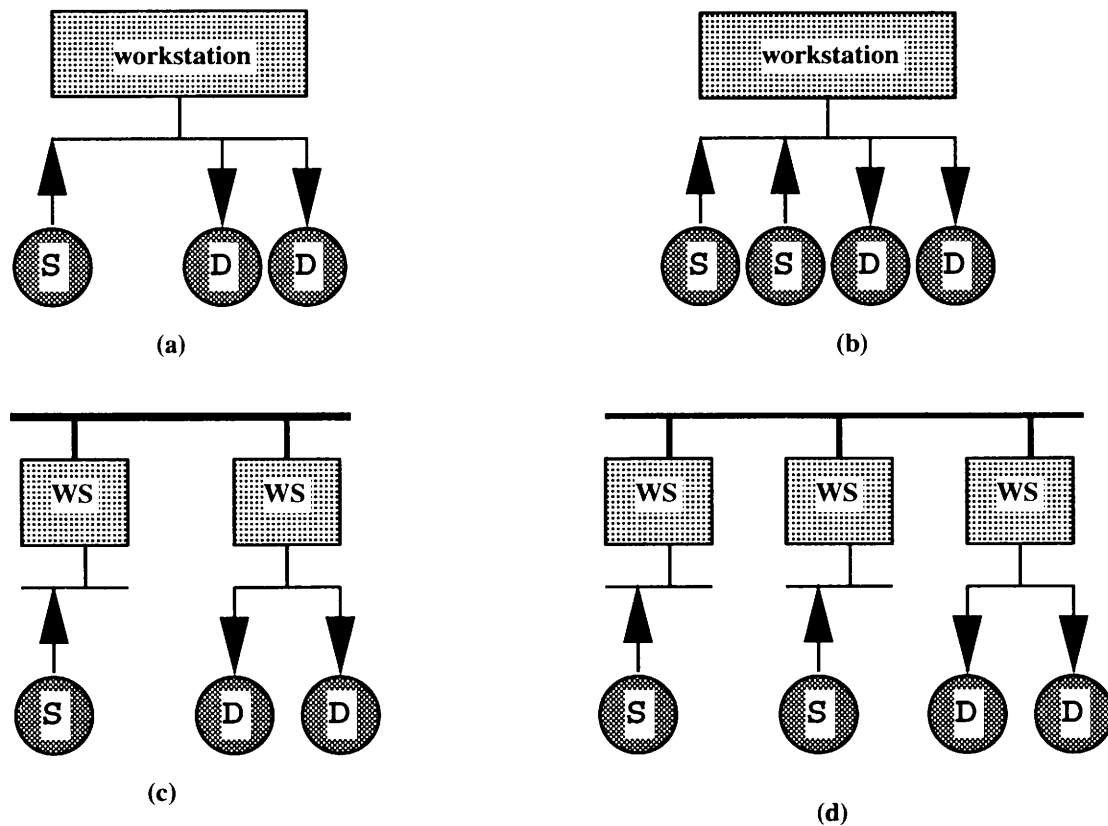


Figure 2: Location of multimedia data: (a) local single source, (b) local multiple source, (c) single distributed source, (d) multiple distributed source

from the source device and then routed (by either the device controller, the operating system kernel or the application) to one or more output devices. The primary attraction of the single source model is that all synchronization among input media is the responsibility of the source material designer. This media is typically interleaved into a single sequential stream that is fetched using conventional system calls.

- **Local Multiple Sources:** This model appears similar to that defined above, except that source data is scattered across several devices. (We again assume that output data goes to several devices as well.) An example of this type of interaction is combining voice annotation with images in an electronic slide-show. The principal difference between single source and multiple source data is the need for some sort of external synchronization between the data streams. The location of this synchronization (either in user code or in the kernel) will depend on the performance needs of the application, with a general trade-off existing between better performance (more towards the kernel) and greater flexibility (more towards the user).
- **Distributed Single Source:** In this model, we assume that a single source of information exists that is located on a remote workstation. The single-source nature of data means that no multi-stream synchronization is necessary. The difference between local and distributed models is that some account needs to be made for the transfer delays between source and destination. These delays may result in portions of the composite material arriving at non-constant rates or out of order. Control over the

data is spread over at least two kernels (the sending and receiving) as well as several protocol layers and a user layer.

- *Distributed Multiple Sources:* This is the most general model of data location. Information may be gathered from many sources on many workstations, and destinations may also be spread over several places. This model is the most interesting because it combines aspects of synchronization problems with transfer delays and raw information scheduling. While current network technologies limit short-term practical applications of this model, we expect that this model will become much more viable within five years time.

The central problem in supporting multimedia data is, then, the degree to which processing layers need to exist between the source of information and its destination(s). In general, the more flexibility required (whether in terms of number of streams or amount of post-fetch processing) the greater the delay. For simple operating systems (such as single-user, single CPU environments), the delays can be easily predicted for a given application. For multi-processing workstations, the coordination and scheduling tasks become much greater. In both cases, the level of complexity is directly related to the amount of synchronization processing required by an application.

2.2. Multimedia Data Synchronization Models

Regardless of the location of data, the data itself can contain synchronization information (that is, it can be self-synchronizing) or it may require synchronization through an external mechanism. Synchronization concerns cover a broad spectrum. In this section, we consider four aspects that affect the partitioning of tasks among the application software, the OS and the device controller(s). These are: the basic type of relationship among data streams, the scope of synchronization information, the determination of the controlling party in a synchronization relationship, and issues regarding the precision of synchronization required.

- *Synchronization Classes:* There are two basic classes of synchronization within a multimedia framework: *serial* synchronization and *parallel* synchronization. Serial synchronization requirements determine the rate at which events must occur within a single data stream; this includes the rate at which sound information is processed, or video information is fetched, etc. Parallel synchronization requirements determine the relative scheduling of separate synchronization streams. In most non-trivial multimedia applications, each stream will have a serial synchronization requirement and a parallel relationship with other streams. Note that a special case of serial synchronization can be defined as *composite* or *embedded* synchronization; in this case, each serial block of data contains information for parallel output streams. In this case, the parallel synchronization among blocks is embedded in a serial stream.
- *Synchronization Scope:* The second distinction is between point and continuous synchronization. Point synchronization requires only that a single point of one block coincides with a single point of another. Continuous synchronization requires close synchronization of two events of longer duration. In general, point synchronization can be managed by the applications layer while continuous synchronization will need to be managed by a device

controller or a high-performance, low-overhead portion of the operating system.

- *Synchronization Masters:* The third distinction regards the controlling entity in a (set of) stream(s). Sometimes we have two channels that are equally important, but sometimes one channel is the "master" and the other the "slave." It is also possible that an external clock plays the role of the master, either for all of the streams or for a subset of time-critical ones.
- *Synchronization Precision:* Finally, there are levels of precision. Stereo sound channels must be synchronized very closely (within 1 to 0.1 millisecond), because perception of the stereo effect is based on minimal phase differences. A lip-synchronous sound track to go with a video movie requires a precision of 10 to 100 milliseconds. Subtitles only require a 0.1 to 1 second of imprecision. Sometimes even longer deviations are acceptable (background music, slides). Note that in all cases the *cumulative* difference between the channels is what matters, not the speed difference.

In general, the synchronization problems make multimedia systems difficult (and interesting). Since each type of medium has its own characteristics, the level of support for a combination of media is a challenging design issue. Most vendors of current commercial equipment use embedded synchronization that is mapped onto a serial stream of data. As a result, they need to consider only point-type synchronization scope with a single master device. The precision is determined by the characteristics of the input source and the system load; most of the synchronization precision is supported by managing interrupt contention between the input and output devices. While this approach can lead to dramatic results, it is not sufficient if the user is to be given more control over the data being processed or if information needs to be combined from several sources (either locally or from distributed points in a network).

3. I/O Processing and the UNIX Kernel

This section will review the standard UNIX I/O model. We start with the layers of logical control that are possible within a UNIX environment to support processing of multiple data streams. We then describe the interaction of these layers when supporting UNIX I/O. Our purpose is to consider generic UNIX facilities rather than the particulars of any one UNIX implementation.

3.1. UNIX Processing Layers

Activity within a UNIX environment can be divided over five general layers in a system: the *thread*[†] layer, the *process* layer, the *kernel top-half* layer, the *kernel bottom-half* (or *interrupt*) layer, and the *device controller* layer. These layers are illustrated in Figure 3. An applications program typically runs on the thread level in user mode. Upon issuing a system call, the processor first switches in to system (or privileged) mode, then acts on the request and then switches back to user mode. All of this processing typically occurs in the context of the process active at the time of the system call. Occasionally, a device

[†] Note that in systems that do not support a user threads package, a process can be considered to be an entity with a single thread of control.

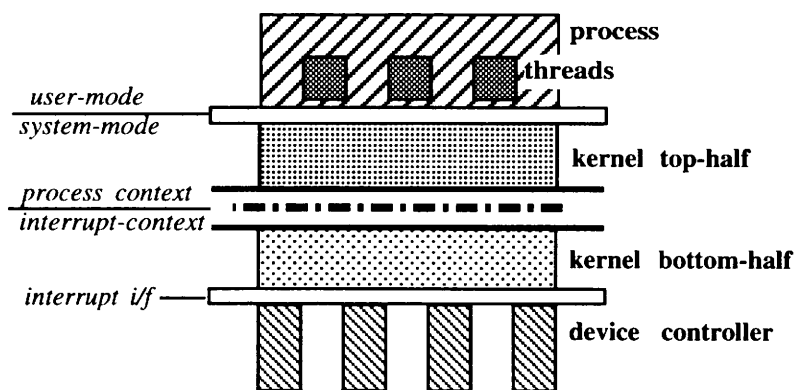


Figure 3: Elements of the UNIX processing hierarchy

will need to interrupt the running process (which is either in user mode or system mode) to service some aspect of a transfer request; such a change is known as a context switch. A typical UNIX kernel will act as the scheduler for a series of user processes, each of which will compete for time on the CPU. Within the process, one or more of its threads will be selected to execute in an implementation specific manner. (Recall that processes are entities that own resources but which do not themselves execute outside the context of their threads.) The kernel may execute its top-half code in response to a system call by the active thread. It may also execute its bottom-half code in response to an interrupt from one of the external devices (such as disks, terminals or the network) or from its system clock. Device drivers run partially as kernel top-half code and partially as kernel bottom-half code. Finally, a great deal of processing may take place within a device controller. Such processing is initiated and ultimately controlled by kernel device drivers, although much of the detailed processing happens in parallel to other processing done on the system. (Note: our partitioning of layers is based on an I/O view of UNIX processing; it is clear that other partitionings are available, but the combination we have shown is most relevant to our multimedia discussion.)

Each of the processing layers has its advantages in managing I/O transfers. The thread level provides an application program with full control over the processing of data. Unfortunately, threads represent the weakest link in a processing chain, since they may be suspended at any time when a higher-priority thread becomes available. Processing within the kernel top-half, on the other hand, is immune from suspension unless the operation suspends itself waiting for I/O completion (or resource availability) or unless an interrupt occurs. Of course, the kernel top-half code is typically not user-modifiable, making it inflexible for information processing (as opposed to information fetching, dispatching or sending). Code that executes at interrupt time has the highest probability of being allocated the CPU quickly; this can allow a fast response to time-critical events, although the amount of processing that can be done on data is usually limited by the need to serve other interrupts quickly as well and a general desire not to "hog" the CPU in interrupt mode. The best performing processing layer may well be the device controller layer. Here, activity can take place in parallel with other devices and the general CPU processing. The disadvantage of controller-layer processing is the limited information that such controllers have on other events taking place in the system. The consequences of this limited information are reflected in a cyclic shifting of process-

ing responsibility that has taken place between the kernel and the device controller in recent years. SCSI disks, for example, use embedded controllers that relieve the kernel of a great deal of overhead processing. (This is usually thought of as a positive development.) Network controllers, on the other hand, have seen a shift from on-controller processing back to in-kernel processing. This is because of the inflexibility of protocol versions that are "baked" into hardware and the difficulty of routing information among different controllers without kernel intervention.

3.2. UNIX I/O Models

The basic I/O facility that UNIX offers to its users is that of a byte-oriented transfer mechanism based on variants of the *read* and *write* system calls. When a user issues the

```
read(fd, buffer, n)
```

system call, for example, the thread is effectively suspended until upto *n* bytes of data are transferred from device *fd* into the buffer named (in this case) *buffer*. This transfer is initiated at the thread level, then processed by the kernel top-half code, then by the device controller, then by the kernel interrupt code, then by the kernel top-half code and then by the thread code. The flow of control may block within the kernel top-half (after initiating the transfer but before the transfer is completed); at this point, the thread is descheduled pending completion of the transfer. Once the transfer-completion interrupt is received from the device, the thread is made reschedulable, although it does not run again until it is the highest-priority thread in the system. If the transfer is completed in one I/O request, the thread is usually able to continue as soon as the kernel top-half code completes (that is, as soon as the system call completes). If the operation is a read/write cycle, such as:

```
read(fd, buffer, n)
```

```
**process the data here**
```

```
write(fd2, buffer, m)
```

then the output data path is similar to that described above (except, of course, in the other direction!).

The movement of data across the various processing layers can be optimized by reducing the number of layers used in each transfer. Unfortunately, if the data needs to be processed in some application dependent way, the options available for speed-up are limited. The general methods of controlling the flow of information within a UNIX environment are shown in Figure 4; each of the four general models are discussed in the following paragraphs. (Note that for purposes of simplification, both kernel top-half and kernel bottom-half processing are regarded as simply "kernel" processing in the following discussion.)

- *Controller Managed I/O*: In this class of I/O, one controller sends information directly to another controller. The kernel, a controlling thread and process data structures are typically not involved once the transfer has started. (Internal resource contention is typically handled by the system hardware.) Controller managed I/O is illustrated in Figure 4.a.
- *Kernel Managed I/O*: In this class, the device controllers transfer information to the kernel, which then dispatches data to other controllers. The controlling threads and process data structures

are typically not directly involved in the transfer. One example of kernel managed I/O is an in-kernel implementation of the IP networking protocol: a packet may arrive from one network device and processed by the kernel IP code, which may then route it to a separate network device. If this is done at interrupt mode, then the transfer can occur quite quickly. Of course, if too much network traffic is forwarded by this kernel, then over-all performance of the system will decrease. Kernel managed I/O is illustrated in Figure 4.b.

- *Thread Managed I/O:* In this class, a controlling thread is used to manage the data transfer. Information is passed from a device controller through the kernel to the thread. The kernel schedules the incoming data as well as the thread; the thread schedules the outgoing data. A single thread may schedule several transfers. The principal advantage of this technique is that it allows data that has arrived at the thread to be processed in an application-dependent manner before it is sent out to an output device. The disadvantage of this technique is that it may take a relatively long amount of time before this processing is completed – with the wait interval almost always being non-deterministic. (This, as we will see, can have very severe consequences for synchronization processing.) Thread managed I/O is illustrated in Figure 4.c.
- *Process Managed I/O:* Although activity in a process can only take place in a thread, we use this class to define multi-threaded activity. Here, UNIX synchronization (such as locking and semaphore activity) is used to merge several data streams. Process managed I/O is illustrated in Figure 4.d.

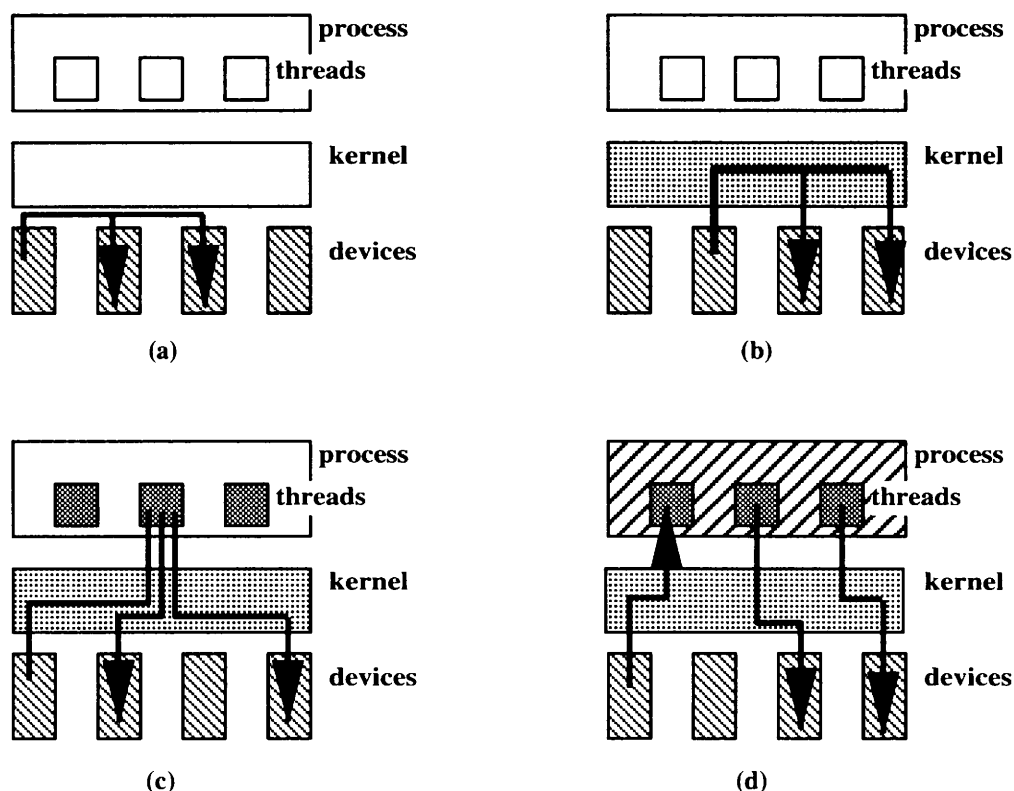


Figure 4: Interactions within UNIX for supporting I/O: (a) controller managed, (b) kernel managed, (c) thread managed, (d) process managed

If large amounts of data need to be processed quickly, controller-controller I/O is typically the only useful choice available to an applications programmer. If, on the other hand, two input or output streams need to be coordinated based on their content, then thread-based (or process-based) processing may be the only option available. This is because controllers have only limited abilities to process data and because the kernel is typically a black-box that cannot be changed by the applications programmer. We consider the ramifications to multimedia I/O of this situation in the next section.

4. The Impact of the UNIX I/O Subsystem on Multimedia Interaction

The previous sections have considered the needs of multimedia applications and the general types of I/O that can be supported by a UNIX kernel. In this section, we combine these topics to consider UNIX's impact on multimedia (and vice-versa). We begin our discussion with a comparison of data location models and then discuss synchronization topics.

4.1. Location Models

In Section 2.1, we defined four location models for multimedia data: *local single source (LSS)*, *local multiple source (LMS)*, *distributed single source (DSS)*, and *distributed multiple source (DMS)*. Given the fact that we are considering multimedia data, we assume that the information that is received from each source will need to be processed in some manner, either by routing a data stream into a number of sub-streams (one for each medium) or by moving composite data from one location to another. Our definitions have explicitly excluded consideration of output locations since the problems encountered here simply mirror those encountered on input. We can measure the effectiveness of the UNIX I/O system by considering the impact of the multimedia location models for each of the types of control considered above:

- *Controller Managed I/O*: In spite of its attractive performance characteristics, controller managed I/O can only play a minor role in multimedia processing. To be sure, information can quickly be transferred from one location to another in a system (such as from a laser disk to video memory), but the relative lack of control given to a user for this type of transfer – which is typically limited to start/stop/rewind/search – will ultimately limit its appeal. In terms of our models of location, controller managed I/O may be useful for LSS data that is self synchronizing, but as soon as any management of separate streams is required, the limited knowledge of the controller will restrict its usefulness.
- *Kernel Managed I/O*: For reasons of performance, kernel managed I/O can potentially play a dominant role in providing multimedia support. One example of this is the use of interrupt context processing to provide a high-speed manner of controlling and routing of incoming data. Another example is the use of multiplexing device drivers to coordinate the activity of a number of I/O streams. This model is especially useful for LMS data and (to a reasonable degree) with DSS and DMS data. Unfortunately, kernel managed I/O has a number of severe limitations, the most restrictive of which is that few application pro-

gram builders have the ability (or desire) to write new device drivers to cope with in-kernel I/O processing. This is especially true for applications that need to share I/O devices with other applications; in this case, driver modules simply cannot be unlinked and relinked efficiently enough to provide the flexibility required by several applications.

- *Thread Managed and Process Managed I/O:* Application-based interaction in a multi-threaded model provides the most general form of support for all types of multimedia data processing. The application programmer can dispatch as many threads as is necessary to handle each type of data. Unfortunately, there is a catch to this flexibility: performance. The non-deterministic scheduling characteristics of UNIX systems make them unreliable at the thread level for collecting and processing information. To understand the limitations of processing at the thread level, consider that it takes about 40 microseconds for a 20 MHz processor to switch from user-mode to kernel-mode in executing a system call. (This is raw system call overhead; processing time is extra.) This means that even if we provide a set of device drivers with a great deal of memory to buffer incoming and/or outgoing data, an application still loses nearly 100 microseconds just in changing the modes necessary to initiate a data transfer between an input and an output device. Since each multimedia transfer will typically cause at least three systems calls for trivial I/O (one for fetching composite data and two for writing it out to two devices), this overhead can be substantial. Add to this the perilous scheduling situation that all threads must endure and the fact that at the thread level only limited resource management facilities exist in UNIX (such as memory locking or explicit control over kernel buffer management), then the situation at the thread level is not particularly encouraging.

The conclusion that can be drawn from this discussion is that the level that offers the best performance in the processing hierarchy (the controller level) is least useful in the general case, while the level that offers the most flexibility may not be suitable for the high-performance needs of multimedia applications. In reaching this conclusion, we have concentrated on the LSS and LMS models. The situation is even worse for the DSS and DMS models, since here multiple thread layers and multiple kernel layers (and, of course, multiple controller layers) must be transited by data as it moves from one machine to the other. We return to this point in Section 5.

4.2. Synchronization Models

While the discussion above focused on the abstract gathering, processing and scattering of multimedia data, in this section we focus on the particular problems that arise in a UNIX environment for handling synchronization processing. In considering the degree to which each of the UNIX processing layers can contribute (or hinder) synchronization of parallel multimedia streams, one immediate problem with which we are confronted is the UNIX scheduler. The priority-based scheduling mechanism offered by most kernel implementations is inflexible in responding to short-term constraints that can occur while synchronizing multiple data streams. This is a consequence of basic UNIX design constraints; even so-called real-time scheduling classes within recent implementations of UNIX do not provide a user with a great deal of dynamic scheduling control to respond to transient critical conditions.

Although the scheduler could conceivably be changed, most users will not be able to do this easily. This means that the synchronization of, say, lip movements to sound data will be very difficult to guarantee unless application-specific processing is included as part of the interrupt service mechanism. As was mentioned above, however, it is unrealistic to expect that a device driver (such as, say, an audio controller) will have the ability to respond to broad synchronization requirements from a variety of applications programs. Instead, we can expect that the audio controller will simply try to push out audio information at a specified rate with only minimal regard for processing in other output streams.

The problems associated with scheduling delays present themselves most clearly when considering the synchronization problems associated with multiple multimedia streams. The facilities provided by UNIX for allowing the close synchronization of even two streams are limited by the processing granularity provided to an application layer. If we assume that a 20MHz CPU will allow approximately 25,000 system calls per second under ideal situations, then it is clear that synchronizing two input streams and then sending them to two output devices has a theoretical upper limit of an average of 6,250 samples per second. In reality, there will also be controller overhead, interrupt overhead, device driver overhead and scheduling overhead, so that the actual number of samples will probably be considerably less. If we assume that we wish to synchronize two high-quality audio streams with each other (requiring no other processing than simply collecting samples and sending them out), then the highest quality we could achieve would be approximately 6 KHz sampling – which is less than 15% of that available with compact discs. If we wanted to do any processing in addition to the collecting and routing of samples, the situation would only be worse. In the case of non-local data fetches, fetch delay associated with a network would need to be added to the processing delays that would accumulate at each of the contributing or consuming hosts. (This example is obviously contrived: in an actual implementation, buffering would take place within kernel processing and actual sampling would be controlled by an external clock; the purpose of the example is to illustrate that even the theoretical limits of applications control granularity do not offer a tremendous amount of processing latitude to the applications designer.)

In terms of our detailed list of synchronization types, we can make the following observations about the ability of UNIX to support multimedia processing:

- *Synchronization classes:* UNIX can do reasonably well in supporting serial synchronization of data if the sampling rates are sufficiently low to not cause a burden on the system. The block-oriented fetching of data can significantly increase the number of samples processed by an application, although the limited scheduling control of each thread will not ensure the constancy required by high-bandwidth devices. For parallel synchronization, the prospects are less promising: the sequential nature of UNIX I/O will result in either a loss of data resolution or in a limit on the number of parallel tracks that can be processed. One reason for this is the form of the generic I/O system call; all I/O is done on a single file descriptor at a time, with separate file descriptors requiring separate system calls. It may be possible to build multiplexing drivers to combine I/O on a number of file descriptors, but this will not offer a general solution to most applications builders. Another possibility may be the develop-

ment of multi-file I/O system calls (with particular synchronization semantics defined in the system call argument list), but even if these were to become accepted by the growing list of standards organizations, most languages would be unable to cope with the notions of parallel I/O accesses. For the time being, the best one can hope to do is to provide either an applications-based multi-threaded scheduling solution to parallel stream synchronization (with all of the performance limitations discussed above) or to rely on smarter controllers to by-pass the CPU altogether.

- *Synchronization scope:* Of the two types of synchronization scope defined above, point synchronization can be relatively well-managed by the thread level, but continuous synchronization can only be managed if the input and output data rates are sufficiently low. Once again, the scope of the synchronization is not only restricted by the implementation concerns of the UNIX I/O system, but also by the ability of applications code to flexibly access data at a low-enough layer in the system.
- *Synchronization masters:* The easiest way to support synchronization within a UNIX environment is to have a master clock regulate the gathering of samples and the dispatching of samples to various output devices. In order for such a clock to function, it will need to be able to influence processing in a number of threads in the same way that real-time clock can influence the scheduling of various real-time processes. (The problems are, of course, not simply similar, they are identical.) Unfortunately, the level of real-time support in UNIX systems has never been particularly good. As for peer-level synchronization, the problems with guaranteed scheduling time under UNIX once again limit the amount of coordinated processing that can be realistically accomplished.
- *Synchronization precision:* Depending on the level of precision, processing can be implemented at any of the five layers in the UNIX hierarchy. If stereo channels need to be synchronized, then it can only occur at the controller or interrupt level (unless the data need only be resynchronized at a much lower rate). If, on the other hand, subtitles need to be added to a running video sequence, then this can easily be done at the thread level.

The general dilemma of processing multimedia data remains that those applications requiring the most processing support are probably the least likely to get it in a general UNIX environment. This is not really surprising: manufacturers of high-performance output devices (such as graphics controllers or even disk subsystems) have long realized that the only way to really improve over-all system performance is to migrate this processing out of the UNIX subsystems. Unfortunately, doing so is difficult for multimedia applications, since the type of processing required over a number of input and output streams is usually beyond the scope of the implementation of any one special-purpose I/O processor. In the next section we discuss a general approach that we are investigating for providing both good performance and reasonable flexibility to multimedia applications.

5. Conclusions and an Alternative Approach

The discussions in the preceding sections can lead us to two general conclusions: The fastest processing layers in the UNIX hierarchy are the device controller and the interrupt layers; these layers enjoy high-priority scheduling and can be invoked with relatively little overhead. In terms of efficient multimedia processing, it can be argued that once you reach either the kernel top-half code or the thread/process layers, it is probably too difficult to provide efficient and deterministic multimedia processing. It can be assumed that for all but the most trivial types of fetch-and-deposit multimedia operations, it is both desirable and necessary to provide a layer of applications support to manage the interactions among the various incoming and outgoing data streams. (Recall that the entire reason for having computerized multimedia systems is the measure of control a user can have over the sequencing and presentation of pieces of data.) This type of processing is "easily" done at the thread/process layers, it is possible (but often impractical) at the device driver layer, it is improbable at the interrupt layer and it is usually totally unavailable at the controller layer.

The net effect of these conclusions is that it is desirable to supply a new programmable layer in the UNIX hierarchy that combine the performance benefits of the existing lower layers with the flexibility of the existing upper layers. In providing this layer, it is probably not useful to simply steal cycles from the CPU – doing this is, in effect, only replacing the existing UNIX scheduler with a semi-real-time one. If we assume that all of the normal services available to a user must continue in addition to multimedia processing, then some form of co-processing will be required to satisfy both the UNIX user and the multimedia application. In closing this paper, we provide a brief description of an approach being studied at CWI for providing multimedia applications support. This approach, which is based on a distributed I/O and processing architecture, is a generalization of existing approaches for offering high-performance graphics and computation processing on a workstation: the special-purpose co-processor.

Figure 5 illustrates the placement of a *multimedia co-processor* (MmCP) as a component of a workstation architecture. The MmCP is assumed to be a programmable device that can be cross-loaded from the master processor. It is assumed that the MmCP can execute arbitrarily complex processing sequences, and that it will have access to all

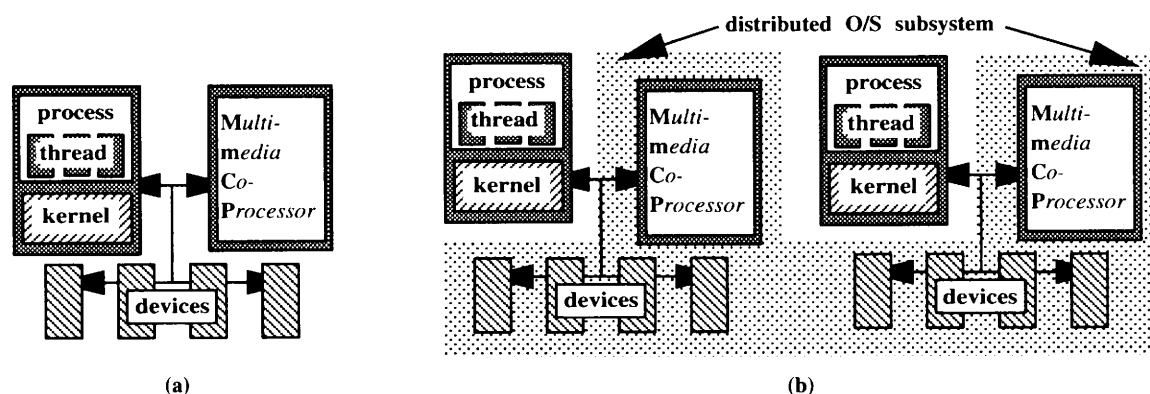


Figure 5: The Multimedia Co-Processor (MmCP):
(a) local architecture, (b) as a distributed operating system

or a part of the workstation's memory. As with arithmetic co-processors, a simple interface should exist to control information flow from the UNIX processor. Unlike normal co-processors, however, we assume that the MmCP will be driven by a distributed operating system that will provide communications support between its hosting workstation and other workstations in a network environment. This distributed support (Figure 5b) will provide for coordination among the various sources and destinations in the DSS and DMS models discussed above.

In the previous sections, not much direct mention has been made of the DSS and DMS models. This is, in part, due to their relative scarcity in current multimedia systems. It is clear to us, however, that there is a great need for coordinated data transfer among various agents in a networked multimedia system. This coordination may consist of bandwidth reservation algorithms for efficient network use or intelligent algorithms for information transfer. An example of the latter type of algorithm may be a transport-style communications layer that knows to bias its service towards one type of media – such as audio – at the expense of others – such as video – if bandwidth become limited during a transmission. If one were to try this in a typical UNIX kernel, then the process and mode switching time may well be longer than the adaptive period of transmission delay!

Our work is currently centered around evaluating the use of the Amoeba operating system as the basis for an MmCP [Mul90a, Ren88a]. Amoeba has two main advantages in our research: first, it has excellent communications characteristics that appear to make it suitable for light-weight protocol development; second, it is a mature but relatively unused system – meaning that it is still an open, experimental system (unencumbered by hundreds of users or thousands of standardization committee members). It should be pointed out that we are investigating *basing* our work on Amoeba, but that we do not intend to replace Amoeba. Also, unlike other operating systems research projects [Acc86a, Dal90a], we are not intending to develop a “micro-kernel” as such (that is, a kernel with core services for use in controlling activity on a workstation), but rather something which could be called a “nano-kernel” i.e. a kernel that handles a particular subset of services that can be allocated to one or more users of on a general workstation (Figure 6). In this sense, our work is aimed at replacing the partitioned intelligence in device controllers with a layer of shared intelligence at a super-controller level. This has the advantages of providing a full (and

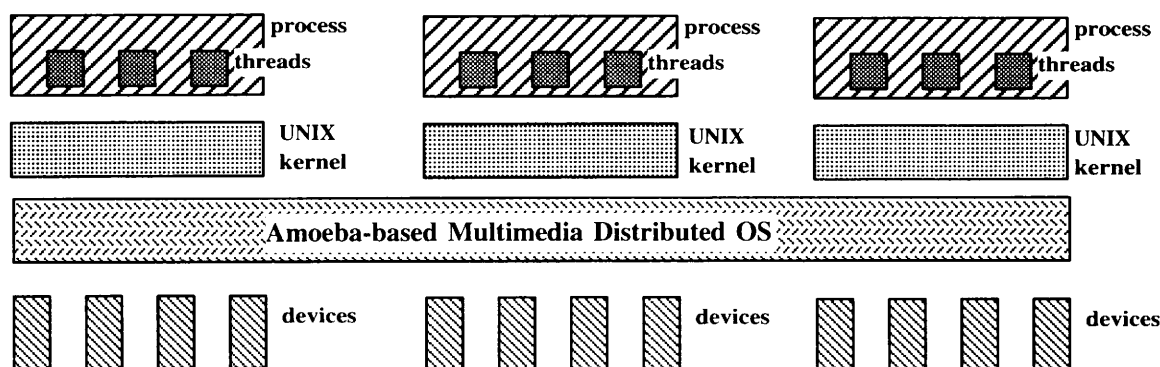


Figure 6: Multimedia support using an embedded distributed operating system

standard) UNIX environment plus a programmable interface layer for high-performance support.

Although we feel that a strong case can be made for the development of the MmCP (either based on Amoeba or otherwise defined), we are only starting a detailed investigation of the resource and functional partitioning requirements needed to support general multimedia systems. This research is driven by two observations:

- First, it should be clear from the sections above that the general motivation for a programmable, high-performance processing layer exists. It may be argued that this need will reduce with faster processors, although we feel that such processors will only stimulate the requirements for even higher processing rather than satisfying it.
- A second development that encourages our work is the rapid development of multi-processor workstation architectures. Although many of these systems are little more than trade-press rumors, several systems (such as the SGI PowerSeries) already provide moderate-cost multiprocessor workstations coupled with a wide array of input and output subsystems. There is no inherent reason why these systems can not simultaneously support multiple operating systems (one for the MmCP and one the remaining processors). An initial port of Amoeba to a Silicon Graphics platform (in our case, a 4D25) has been successfully completed as a proof-of-concept project.

We view these reasons as providing a basis for further work at the applications, kernel interface and architecture layers of the workstation.

6. Summary

We have attempted to argue that the conventional UNIX environment for workstation computing – as useful as it is for many applications – may not be ideally suited for high-performance multimedia computing. Although some of the factors that constrain UNIX are technology dependent, much of this problem lies with fundamental design issues that were a part of the original uniprocessor, sequential serial I/O model developed for UNIX in the 1970's. The approach of the multimedia co-processor that we have presented here is an attempt to overcome many of these problems without sacrificing the positive aspects of a uniform UNIX interface.

Our work is being done as part of the CWI/Multimedia research project, an interdisciplinary effort to study various related aspects of the multimedia problem. Please note that organizations and commercial firms mentioned in this paper have been selected as examples of architectures and approaches to supporting workstation computing. No attempt has been made to survey all relevant manufacturers of multimedia workstations and no particular endorsement is intended or implied for those companies referenced.

References

- [Acc86a] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 Usenix Conference*, Atlanta, GA (July 1986).

- [Bul90a] D. C. A. Bulterman, "The CWI van Gogh Multimedia Research Project: Goals and Objectives," CST-90.1004, CWI (1990).
- [Bul91a] D. C. A. Bulterman, G. van Rossum, and R. van Liere, "A Structure for Transportable, Dynamic Multimedia Documents," *Proceedings of the Summer 1991 Usenix Conference*, pp. 137-156 (June 1991).
- [Dal90a] P. Dale and I. Goldstein, *Realizing the Full Potential of Mach*, OSF Internal Paper, Open Software Foundation, Cambridge, Mass. (1990).
- [Mul90a] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and J. M. van Staveren, "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer Magazine* **23**(5), pp. 44-53 (May 1990).
- [Pro89a] SIGGRAPH-89 Panel Proceedings, "The Multi-Media Workstation," *Computer Graphics*, Boston, Mass. **23**(5), pp. 93-109 (Dec 1989).
- [Ren88a] R. van Renesse, H. van Staveren, and A. S. Tanenbaum, "Performance of the World's Fastest Distributed Operating System," *Operating Systems Review* **22**(4), pp. 25-34 (Oct 1988).

Using a Wafer-Scale Component to Create an Efficient Distributed Shared Memory

Aarron Gull

City University

London, UK

aarron@cs.city.ac.uk

Abstract

Within a decade a revolution will have occurred in computing: for by then, byte for byte, the cost of semi-conductor storage will be lower than that of magnetic discs. When this happens, it is argued that magnetic storage will be totally replaced by wafer-scale integrated silicon-based mass-storage devices.

In preparation for this day, a method of creating Dynamic Random Access Memory (DRAM) wafers which combine shared storage and communication properties is proposed. These can be stacked like conventional disc platters to form storage devices called Wafer Discs.

This paper describes Wafer Disc and shows how it can be used to construct a scalable multi-computer based on a shared optical disc cache.

1. Introduction

It is widely accepted that significant increases in computer throughput can only be achieved through the use of multiprocessing. The principal rule when designing multiprocessor systems is to achieve a balance between the number of processors, the degree of communication between them and the bandwidth of the communications medium used to connect them. Two types of multiprocessor system are common:

Tightly-coupled multiprocessors

The processors in tightly-coupled multiprocessors systems are typically linked by a common very high bandwidth bus to a shared memory. However, as there is a high degree of communication between the processors, such systems are limited in size; even the largest multiprocessor systems rarely have more than 30 processors [Sys87a].

Loosely-coupled multicomputers

The processors in loosely-coupled multicomputers are linked by relatively low bandwidth communication channels. However, the processors have private memories and, if there is sufficient

locality, communicate relatively rarely. This enables such systems to be very large.

Multicomputers, although scalable, are notoriously hard to program. Such systems typically use the message passing parallel programming paradigm. This requires programs and data to be partitioned so that the maximum multiprocessing can be achieved with the minimum of communication. This proves to be difficult and, as a consequence, multi-computer operating systems tend to be complex, employing many different mechanisms and caching policies [Win89a].

Multiprocessor systems, though limited in size, are often very easy to program. These use the common memory to implement the shared variable programming paradigm. There is no need to partition data in such systems, as all of the processors have access to it. Multiprocessor operating systems often use the shared memory to simplify many of their mechanisms and caching policies. Mach, for example, unifies interprocess communication, virtual memory and backing storage [Acc86a].

A number of people, notably Li [Li86a], Delp and Farber [Del86a] and Fleisch and Popek [Fle89a], have attempted to combine the programmability of multiprocessors with the scalability of multicomputers by implementing large-grained shared memories on multicomputers. The efficiency of these systems usually suffers, however, network and server contention.

City University has proposed a method of combining a large scalable shared memory and a high bandwidth communication network in a single wafer-scale component called Wafer Disc. This paper describes this device and suggests it could be used as a medium for the construction of efficient shared memory multicomputers.

2. Background Information

In 1990, Hennessy and Patterson [Hen90a] observed three trends in the memory hierarchy:

- *DRAM-Growth Rule:* Density increases at about 60% per year, roughly quadrupling in 3 years.
- *Disc-Growth Rule:* Density increases at about 25% per year, roughly doubling in 3 years.
- *Address-Consumption Rule:* The memory needed by the average program grows by a factor of 1.5 to 2 per year.

If the DRAM and Disc-Growth trends observed by Hennessy and Patterson continue, one Giga-bit silicon memory devices will exist within 9 years. As the cost of storage is typically inversely proportional to its density, within a decade these will cost under \$25 each.[†] About this time, byte for byte, silicon memory will provide cheaper storage than magnetic discs and will probably replace them entirely. Non-volatile storage could be performed by cheap, but slow, archival mediums such as CD/writable optical discs and high density tapes.

Even though silicon memory is decreasing in cost, according to Hennessy and Patterson's Address-Consumption rule, the memory requirements of programs are growing at about the same rate. Today the typical mini-computer has at least one Gbyte of magnetic disc storage.

[†] DRAM currently retails for around \$6000 per Giga-bit of 80ns memory. A 760 Mbyte magnetic disc can be purchased for \$2800, around \$470 per Giga-bit.

Assuming that magnetic memory requirements are proportional to program sizes, in 12 years 130 Gbytes of silicon memory will be needed simply to replace the magnetic storage in a typical system. Even using one Gbit DRAMs this will require over one thousand chips.

Hardware reliability problems are most often caused by defective connections, either in the form of the soldered joints connecting the chip pins to the printed circuit boards (PCBs), or the bond wires inside chip packages which connect the pins to the silicon. In effect, a system's mean time to failure is proportional to its number of pins. Assuming that a 1 Gbit DRAM chip will require 24 pins, the memory alone in a typical mini-computer will have around 25,000 pins. Consequently, such systems will have considerable reliability problems.

Wafer-scale integration (WSI), the process by which integrated circuits are packaged as whole wafers rather than as individual chips, is an appealing solution to this problem. Most of the interconnections in a wafer are made by aluminium alloy metalisation on silicon. This is inherently more reliable than both PCB interconnections and chip bonds. In addition, Wafer-scale integration has the further advantages of lower cost per function, higher performance and reduced PCB footprint. However, it has the disadvantage that successful WSI requires built-in fault-tolerance as the yield of defect-free wafers is essentially zero.

As a storage medium, Wafer-scale silicon storage has several advantages over conventional magnetic storage devices: it is faster; it can be accessed randomly while magnetic storage only allows pseudo random access to data; it is more reliable and, finally, it has better handling properties. It has the disadvantage, however, of being volatile.

Although Carlson and Neugebauer [Car86a] have dismissed memory as a WSI candidate, work by Chesley [Che87a], Anamartic [Cur89a] and Anderson et al [And89a] have shown it to have considerable promise.

- *Chesley's Wafer Virtual Memory Proposal.* Chesley has proposed a non-redundant approach to constructing wafers of DRAM. By not employing a redundancy scheme, Chesley uses 100% of the area of the wafer. Thus a wafer yield of around 70 per cent is obtained without incurring the penalties usually associated with error detection and correction.
- *Anamartic's Wafer Stack.* One of the first commercial attempts to provide Wafer-scale memory is the Wafer Stack device produced by Anamartic (formally Sinclair Research). This uses six inch wafers containing 202 one-megabit DRAMs. It had two main disadvantages:
 - ♦ It was relatively expensive. This was because it did not employ state-of-the-art memory chips.
 - ♦ It was slow; it only employed a single 8 bit wide Catt Spiral [Aub78a] data path and had a slow processor acting as an intelligent interface. Consequently it had a latency of around 200 μ s and a peak transfer bandwidth of 800 Kbytes per second.
- *City University's Wafer Disc Proposal.* City University has proposed a method of creating DRAM wafers, called Wafer Discs, which combine shared storage and communication properties.



3. The Wafer Disc Proposal

The Wafer Disc proposal is based on a set of five wafer-scale integration design rules formulated at City University during the Cobweb project [And90a]:

- Successful wafer-scale integrated devices should comprise a large number of independent payload blocks embedded in a reliable communications network. This will result in high fault-tolerance.
- The communication and payload circuitry should be separated so that the communication architecture is general purpose and reusable. This will lead to faster product design and lower costs.
- The wafer functions should be kept as simple as possible. Any complex processing should be done off the wafer. This will improve the yield of usable components.
- To reduce production costs, the minimum amount of defect repair should take place. Only power drains, which otherwise would be fatal, should be patched prior to wafer packaging.
- Wafer testing should be accomplished by external circuitry which will inject test patterns into the edge of the wafer which then percolate.

The main components envisaged in a Wafer Disc system are Wafer Disc Platters (WDPs) and Wafer Disc Interfaces (WDIs). A system may contain large numbers of both types of component:

- *Wafer Disc Platters.* A Wafer Disc Platter is a wafer which contains a regular mesh of Communication Elements (CEs). Each CE is connected to its nearest neighbours by busses, forming a four-connected communication network. Each CE is also connected to a Payload Element (PE) comprising a Memory Access Controller (MAC) and some DRAM. The Wafer Disc architecture is illustrated in Figure 1.

At the edge of the wafer the busses are connected to pads. These are bonded to the pins in the wafer packaging to create external busses. The wafer contains an excess of pads to guarantee that all of the pins can be bonded to functional CEs. Future technology may allow high bandwidth optical connectors and fibres to be used for external communication.

The dimensions of the components in a Wafer Disc Platter are determined by two compromises:

- ♦ *The communication bus width.* The width of the communication bus is a compromise between employing wide data paths (which provide higher communication bandwidth) or narrow data paths (which have a higher yield, and consume less of the wafer).

The main problem with wafer-scale communication is that although a wafer's internal bandwidth is high, it is ultimately connected to the outside world which employs relatively low rates. As it is pointless to provide higher internal bandwidth than can be utilised, it was decided to employ narrow data busses comprising 32 bit data-paths and 4 signal lines.

- ♦ *The number of CEs,* in a wafer is a compromise between having few CEs (thus allowing more of the wafer area to

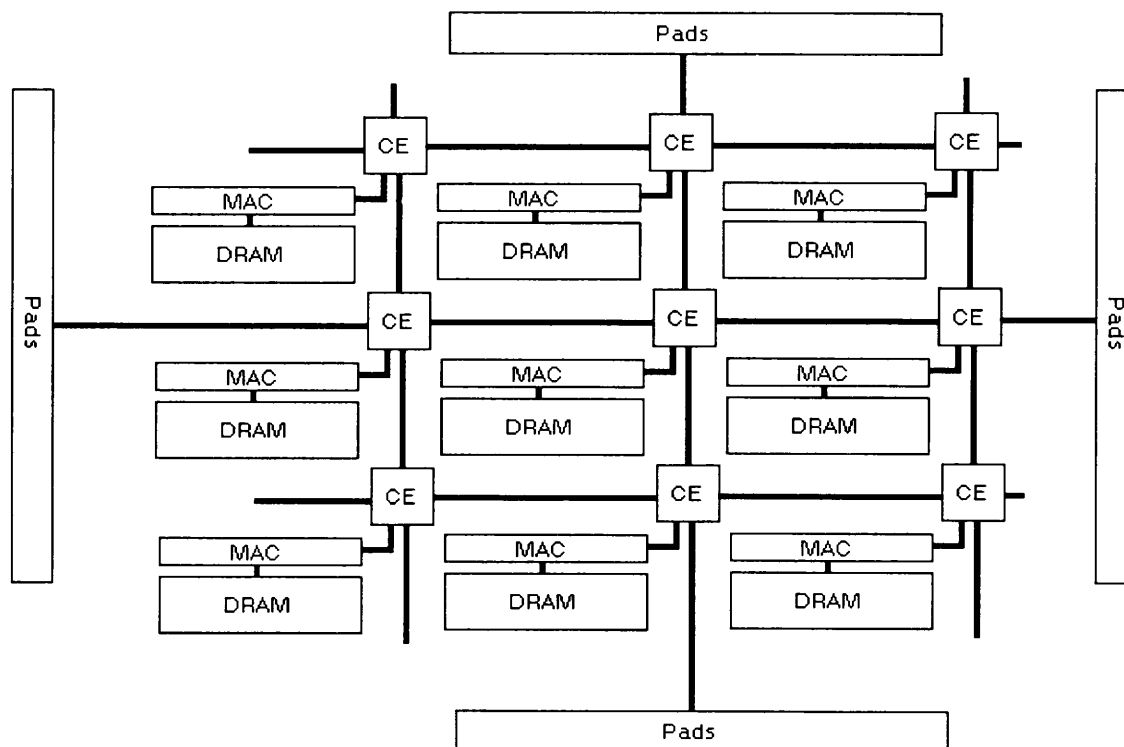


Figure 1: The Wafer Disc Architecture

be devoted to memory and the average communication path length to be low) and having many CEs (resulting in less contention in the wafer).

As the scalability of any shared memory is highly dependent upon contention and the communication network is compact and very fast, it was decided that it was appropriate to employ a large number of CEs. Assuming that the smallest amount of DRAM that is economic has dimensions $5 \times 10^6 \mu$ by $1.3 \times 10^6 \mu$ and CEs occupy approximately $4.1 \times 10^6 \mu^2$ an eight inch wafer can contain about 2,800 CE and PE pairs in a 33 wide by 107 high oval.

- **Wafer Disc Interfaces.** Each processor in the computer system will have a dedicated chip on its PCB called a Wafer Disc Interface (WDI). A WDI is effectively a chip-mounted CE which is responsible for interfacing between the processor and the external busses of up to four WDPs. A WDI, however, does not necessarily have to be connected to a processor; it can simply be used to link a number of WDPs. In this way, WDIs act as "glue", allowing WDPs to be chained together to form potentially massive storage systems. A 2-connected system is shown in Figure 2.

3.1. The Communication Element

One of the major benefits of the Wafer Disc Architecture is that the storage capacity of a WDP is similar to that of a wafer containing conventional DRAM chips. This is because the communication network takes up the space normally reserved for pads, test dies and scribe lines.

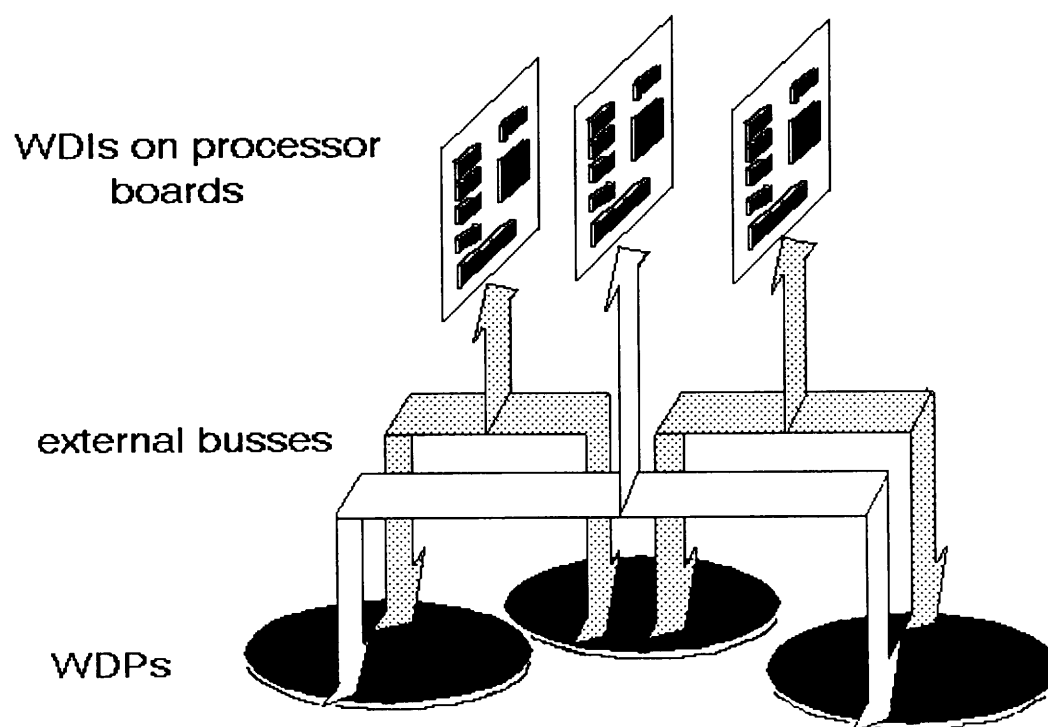


Figure 2: *Connecting WDP to Computers*

This makes Wafer Disc an economical alternative to conventional silicon memory.

Circuit-switched communication appears most attractive for Wafer Disc. (This technology was investigated by Winterbottom [Win87a, Win88a, Win89a].) In order to reduce the routing circuitry on the wafer, communication is always initiated by a Wafer Disc Interface.

Communication occurs in three stages: circuit construction, data transfer, and circuit collapse.

During circuit construction, an electronic circuit is established between the WDI and the destination CE. The head of the circuit starts at the WDI and contains routing information. This is a series of two bit pairs which specify the course the head must take at each CE, relative to the current direction of travel, to get to the destination. The routing scheme is illustrated in Figure 3a.

The top two bits of the routing information are stripped off by the CE connected to the WDI and examined. If possible, the bus in the requested direction is allocated to the circuit and the head of the circuit passes to the next CE in the route. This procedure is repeated until the circuit head reaches its destination.

Up to two circuits can be routed through each CE as long as they use different busses. Several possible combinations are shown in Figure 3b. If, at any point, a bus required by the circuit is already in use, the circuit back to the source CE collapses immediately, thereby freeing the allocated busses and providing freedom from deadlocks. When a circuit collapses, the sending CE must back off and retry after waiting some random interval.

When a circuit has been established, data transfer can take place. The circuit appears to be an auto-simplex parallel (32 bit wide) shift regis-

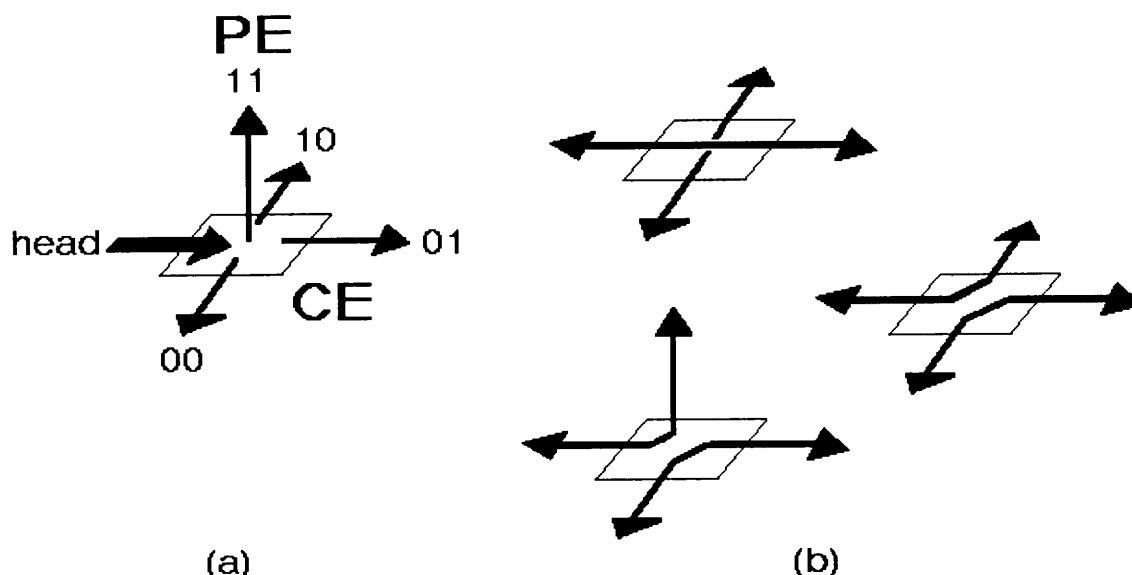


Figure 3: Circuit-switched routing

ter. If this is clocked by the source CE at 40 MHz, a hop time of 25ns and a raw data transfer rate of 152 Mbyte/sec will be obtained.

When the communication is complete, the source CE dissolves the connection which in turn causes circuit collapse. The collapse frees the allocated busses.

Routes are held off wafer in look-up tables in the Wafer Disc Interface. When the system is powered-up, the Wafer Disc Interface tests the wafer, locating the defective CEs and building routes to the working ones around the duds.

In general, the number paths blocked by a circuit is minimised when the head makes as few direction changes as possible. This usually results when the head travels straight into the wafer until it is level with the destination CE, then turns left or right as appropriate.

The advantages of this communication scheme include:

- The communication protocol has been proved to be deadlock free by Whobrey [Who88a].
- A circuit-switched CE is much smaller than a packet switched one. This results in an exceptional communications network yield.
- When a circuit is established, exclusive communication with the CE is guaranteed for its lifetime. This assures atomicity, a property often desirable in multiprocessing systems.
- It uses well-understood technology which has already been applied practically at City University.

The main disadvantage of this scheme is:

- Making and holding a circuit ties up a number of data paths in the wafer. As the number of circuits increases, the probability of being able to make an additional one drops. This is largely offset, however, by the huge number of possible data paths in the network.

A gate equivalence scheme has been used to compute the area of a CE using very conservative 1.5 micron technology. The values obtained are shown in Table 1. The majority of the wafer area is consumed by memory and power rails.

Structure	Components	Area ($\times 10^6 \mu^2$)
CE	Control Logic ($\approx 2,000$ gates)	0.9
	Routing Bus	1.3
	Power Rails	1.9
PE	Memory	6.3

Table 1: Areas of the Wafer Disc Cell Components

Assuming a memory region has dimensions $5 \times 10^6 \mu^2$ by $1.3 \times 10^6 \mu^2$, CE and PE pairs will occupy approximately $5.8 \times 10^6 \mu^2$ by $1.8 \times 10^6 \mu^2$, the CEs consuming 39% of the wafer and the PEs 61%. The total CE area is $4.1 \times 10^6 \mu^2$. The CE yield has been predicted as 94% by the negative binomial model, assuming logic fault rates of 0.03 defects/ mm^2 , metal fault rates of 0.01 defects/ mm^2 and a clustering parameter of 0.75 to show a high degree of defect locality.

The harvest of CEs is defined as the proportion of the total CEs which can be configured into a connected network attached to the wafer's pads. If a wafer has several disjoint networks, the harvest refers to the largest. When leaving room for pads, an eight inch wafer can contain about 2,800 CE and PE pairs in a 33 wide by 107 high oval. With this large number of elements and the exceptional yield, Wafer Disc's fault tolerant architecture ensures that the harvest is very close (within a fraction of a percent) to the communications yield. Using 1 Gbit technology, each PE will contain about 8 Mbytes of DRAM. If a harvest of 94% is achieved, the raw storage capacity of a single eight inch wafer will be around 22 Gbytes.

It is assumed that a wafer can reasonably have 1,000 pins. Allowing two power, four control and 32 data lines per bus, up to 25 Wafer Disc Interfaces could be connected to each wafer.

Simulations have shown that wafers will have an average of 4,880 functional busses and the mean circuit length between the WDIs and the PEs will be 62 hops. The mean circuit length between WDIs will also be very similar to this. However, as a wafer is effectively oval in shape, the mean circuit length from a WDI will be highly dependent upon its position in the wafer's major axis. Consequently some WDIs will be faster at constructing circuits than others.

The implications of bus load on constructing a circuit are shown in Figure 4. If a multicomputer's I/O subsystem is balanced so that typically two average length circuits are active simultaneously (consuming 124 busses), there is a 20% chance of making a third. If the retry period is equal to the average transfer time, the expected number of attempts at making a circuit will be 5. Consequently, if each hop takes 25ns, the expected time to make a circuit will be around 7.8 μs . Dropping a circuit takes no time.

3.2. The Payload Element

Each Wafer Disc Payload Element will contain a rectangle of DRAM and a Memory Access Controller. It is envisaged that a single PE will hold as small an amount of memory as economically possible. This will minimise data contention within the wafer.

The DRAM payload blocks will be very similar in form to one of the arrays of memory which comprise conventional DRAM chips. The

Probability(making circuit)

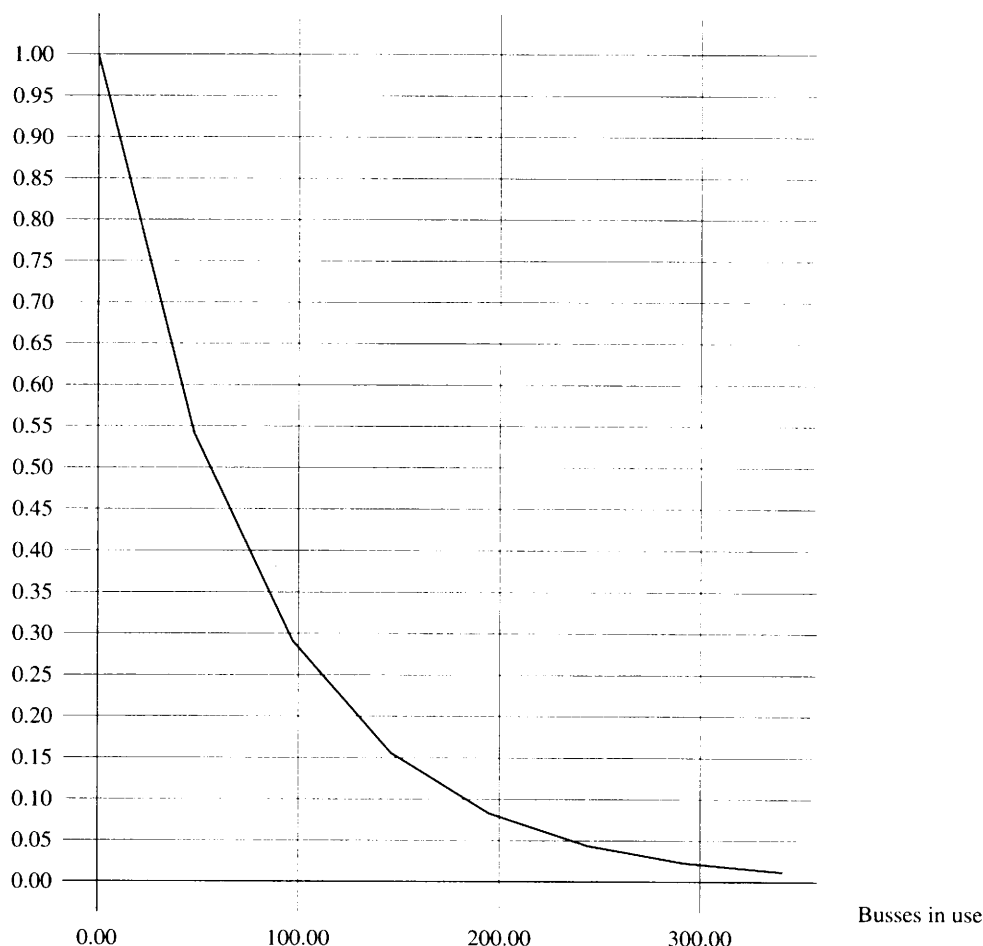


Figure 4: Graph to show the probability of making a new circuit given that circuits already exist

dimensions of these are typically determined by a combination of the length of their sense lines, the size of their sense amplifiers and the amount of memory access logic required. An array could reasonably measure 0.5 cm \times 0.125 cm. Using 1 Gbit technology, each would contain just over 8 Mbytes of memory.

Many of the DRAM blocks on a platter will contain defects. Generally, these fall into two categories:

- **Hard Defects** are permanent faults caused by physical, usually visual, anomalies in the wafer such as lithographic errors or contaminations. Hard defects can affect single bits, whole rows or columns or even whole arrays of memory.

To increase the yield of usable memory, the DRAM block in each PE can be divided into logical fragments which are accessed independently. Each DRAM array is divided into sixteen subarrays, or fragments, each 1,024 by 64 bytes. On power-up, the Wafer Disc Interface tests the memory in the wafer and locates the defect-free fragments. Fragments which contain hard defects, even a single bit, are not used for storage. This is illustrated in Figure 5.

- **Soft Errors.** Soft defects are non-permanent transitory faults which occur randomly due, for example, to alpha particle decay or chip degradation. Due to their nature, soft defects tend to affect isolated bits within the memory. As a consequence of this, over a period of time the contents of a silicon based storage dev-

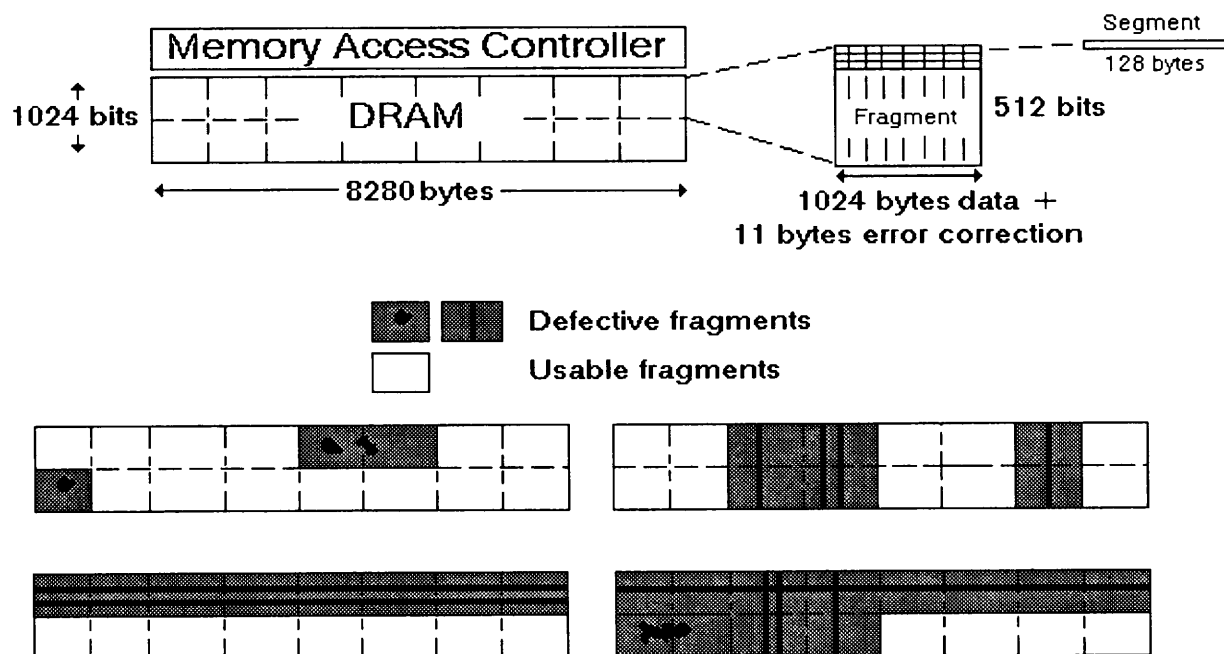


Figure 5: Increasing memory yield by fragmentation

ice will slowly degrade, in contrast with magnetic media. It is necessary to employ a mechanism in Wafer Disc which can correct, rather than just detect, soft errors.

The low occurrence of soft errors suggests that a light-weight error correcting code, such as Hamming Code [Gol86a], is suitable for Wafer Disc. The Wafer Disc Interface applies the error correcting code to the data written to the wafer. This introduces redundancy which allows occasional bit errors to be corrected when the data is read.

Each fragment row can be divided into eight 128 byte segments which are accessed independently. To error correct a single bit error in a segment using Hamming Code requires 11 bits, a 1% overhead. These extra bits are incorporated into the segments, making each fragment row 8,280 bits long; an 8 Mbyte PE requires an additional 88 Kbytes for error correction.

The contents of a fragment are read and written as a series of segments. When writing a row to the wafer, the WDI buffers the data to be written until it has amassed a full 1,024 bits. It then inserts the error correcting code and sends the 1,035 bit result to the wafer. When reading from the wafer, the reverse happens; the WDI buffers the data read until it has received a full 1,035 bits. It is then possible to error correct it. The error correction circuitry contained within the WDI so as to simplify the MACs.

A WDI connected to the wafer will be assigned the task of cycling through the segments, sequentially reading and rewriting them. This will have the effect of repairing soft errors caused by alpha-particle decay, thus reducing the probability that a block will be irreparably affected. If a segment is found to be frequently in error, it is likely that it contains an intermittent fault. The contents of the fragment containing the segment are copied to a spare fragment in the wafer. As this processes is effectively

refreshing the memory twice, an attractive possibility would be to place the error decoding circuitry in the MACs, allowing them to perform "smart refresh". This would make external error correction circuitry unnecessary, and at the expense of storage yield.

The Memory Access Controller (MAC) is responsible for transferring data to and from its DRAM payload. The MAC supports the reading and writing of complete 1,035 bit fragment segments. When a WDI initially makes a connection with a MAC, it sends it a single 32 bit request. This has the following format:

Request	Fragment	Start row	Start segment	End row	End segment	Unused
(R/W)	(0-15)	(0-511)	(0-7)	(0-511)	(0-7)	
1 bit	4 bits	9 bits	3 bits	9 bits	3 bits	3 bits

If the request is for segments to be read from the payload, the MAC reads the requested data from the memory and sends them to the WDI, where the error correction code is removed. If the request is for segments to be written to the wafer, the WDI inserts the error correcting code into the data and sends them to the MAC, which writes them to its payload.

When the transfer has been completed, the WDI may initiate new transmissions by sending additional requests.

If 32 bit words are clocked across the data bus using a 25 ns clock, the time taken to transfer a single 128 byte segment to or from the wafer will be 800 ns. It is estimated that this could be error corrected in around 400 ns. If 2,048 byte data blocks are employed, the overall transfer latency will be about 19.2 μ s, or a block transfer rate of 100 Mbytes/second. It is argued that the computer bus will be able to support this rate.

When the memory access controller is idle, it must refresh the DRAM. Unfortunately, the power required to refresh a wafer is large. Consequently, it is difficult to provide WDPs with the battery backup needed to make them non-volatile.

4. Important Optical Disc Cache Design Issues

Three issues are considered important when designing the Wafer Disc optical disc cache:

Storage Semantics

One of the features of current optical discs is that once blocks are written they are effectively immutable. This poses a problem as these semantics differ considerably from those of conventional memory; an optical disc block which is modified cannot be overwritten with its new contents.

This problem is solved by introducing a level of indirection into the caching which maps logical optical disc block addresses on to physical locations on the optical disc. When the contents of a logical optical disc block are modified, the indexes are updated to point to its new physical location on disc. This allows the semantics of conventional memory to be emulated at the expense of sacrificing inexpensive optical storage capacity.

Cache Organisation

There are three common categories of data organisation which are employed in caches. In order of increasing performance and hardware cost these are: direct mapped, where a block can only appear in one location in the cache; set associative, where a block can appear within a restricted set of places; and fully associative, where it can appear anywhere. In reality all three are really set associative caches with different set sizes.

Bugge et al [Bug90a] have shown that the most important factor effecting the performance of a set associative cache is its size. The degree of associativity employed is of lesser importance, but when it is high, Random block replacement schemes, which are cheap to implement, perform as well as the more popular, but expensive, Least Recently Used schemes. Wafer Disc allows the creation of a Gbyte-sized cache with a high degree of set associativity and, therefore, should provide excellent performance at a low cost.

It is suggested that each 512 Kbyte Wafer Disc fragment should contain three cache sets, each of which would have 64 two-Kbyte blocks.

Data Coherence

As the number of active circuits in a system at any one time should be minimised and the majority of cache accesses will be reads, it is suggested that processors should be allowed to cache local copies of optical disc blocks. A mechanism is required, therefore, which maintains the coherence of these.

It is suggested that Wafer Disc should employ a multiple reader, single writer block coherence scheme.

Cache Size

One of the major factors in determining the performance of a Wafer Disc cache will be the communication circuit construction time. Due to the increased chance of collisions, the time taken to make a circuit is exponentially proportional to its length. Constructing long circuits, such as those which pass from one wafer to another, is much slower than making short ones. This suggests that some buffering scheme could be appropriate.

The variation in the potential block access latency gives Wafer Disc Non-Uniform Memory Access (NUMA) characteristics. To prevent the disparity in block access latencies from becoming too large, a hard limit must be set on the maximum circuit length in a system. This effectively limits the number of WDPs that can be linked together.

For example: a five wafer system, in which wafers are mutually connected by two WDIs, would have 100 Gbytes of raw storage and up to 60 processors. The maximum circuit length in this system is 214 hops, the minimum is 1 and the average is 124. The variation in block access latency this creates is not thought to be excessive.

The relatively long block access latency is also a significant factor in determining the granularity of memory access which can be realistically employed in Wafer Disc.

5. Optical Disc Cache Mechanisms

The implementation of the optical disc cache can be divided into three distinct mechanisms: address mapping, coherence maintenance and block caching.

5.1. Address Mapping Mechanism

Each defect-free PE fragment in the wafer acts as a cache for sets of logical optical disc blocks. For simplicity, a modulo-based hashing scheme is used to map logical optical disc blocks onto PE fragments and cache sets. This scheme distributes contiguous logical optical disc blocks widely amongst the PEs which, assuming that there is some locality of block reference, should reduce contention.

The scheme requires each Wafer Disc Interface to have two maps, which in a five wafer system will consume up to 128 Kbytes of storage.

Hash Map

To implement the modulo-based hashing scheme, the WDIs need to know the number of defect-free fragments in each PE. This information is contained in a Hash Map, an identical copy of which is held in each WDI. Each CE is assigned an unique Logical CE Number by which it can be globally referenced. This allows the WDIs to map logical optical disc blocks onto the logical CEs, fragments and sets which cache them.

For example: the simplified system illustrated in Figure 9 has 4 CEs, each with 6 fragments. The total number of defect-fragments is 20. Optical disc block 115 is cached by logical CE 4 (lookup $115 \text{ modulo } 20$) in fragment 0 ($115 \text{ modulo } 20 = 15$). Assuming there are three coherence sets this block would be held in set 2 ($\lfloor 115 \div 20 \rfloor \text{ modulo } 3$) with a reference tag of 1 ($\lfloor 115 \div [20 \times 3] \rfloor$).

In reality, a five wafer system can have up to 14,000 CEs and 224,000 fragments. Assuming each CE requires a five byte entry, a Hash Map will be around 70 Kbytes long.

Wafer Route Map

Once the logical CE containing the fragment responsible for caching a logical optical disc block has been determined, a WDI must make a circuit to it. The CE's Logical Number is used as an index into the Wafer Route Map. This gives the route from the WDI to the CE. The routes to the WDIs are also contained in this map.

It is estimated that the routing information for an average 62 hop circuit, normally requiring 124 bits of information, could be

Logical CE	Fragment bitmaps	Addresses cached modulo 20
1	011111 (5)	0-4
2	111111 (6)	5-10
3	100111 (4)	11-14
4	111011 (5)	15-19

Table 2: An example Hash Map

compressed to 2 bytes. When indexing information is included, a 5 wafer system containing around 14,000 CEs will require a 55 Kbyte Wafer Route Map.

As the wafer routing scheme is relative, the Wafer Route Maps are unique for each WDI.

5.2. Coherence Mechanism

After a Wafer Disc Interface has connected to the PE which is responsible for caching a logical optical disc block it must ensure that accessing the block in the requested manner will not violate the coherency of any data cached by other processors. This is accomplished by employing a coherence mechanism which only allows a logical optical disc block to be cached read-only by multiple processors, or read-write by a single processor.

When a processor generates a read or a write fault on a block, it may have to invalidate the cached copies of it held by other processors:

- If the processor write faulted, its WDI must invalidate all other cached copies of the block.
- If the processor read faulted on a block which another processor has a read-write copy of, its WDI must restrict the other's access to read-only.

Once coherency has been guaranteed, the processor is allowed to make, and work from, its own cached copy of the block until it wishes to relinquish access, or another processor invalidates it, whereupon, if modified, it must be written back to the wafer. A simplified (non-optimised) version of the coherence algorithm is given in the appendix.

The data structures required to maintain coherence are stored on the wafer along with the blocks they reference. This has several advantages:

- The structures are distributed evenly over the system, thereby allowing concurrent access and spreading load.
- Each WDI acts as its own block server, thus eliminating server contention and making the WDIs accountable for their own actions.
- The number of circuits that have to be constructed when accessing blocks is minimised.

Two types of tables are employed by the coherence mechanism: Block Lists and Copy Lists. Both types of table are of a fixed length. This simplifies their access at the expense of being less efficient in their storage.

- *Block Lists.* Each of the three coherence sets in a fragment has a Block List held in a well-known 128 byte segment. This list contains a valid entry for each of the set's logical blocks which is currently cached by a processor. Each list can contain up to 64 two-byte entries, the format of which are:

Valid (1 bit)	Busy (1 bit)	Address Tag (14 bits)
------------------	-----------------	--------------------------

The Valid field is set when an entry holds authentic data. The Busy field can be used to lock the entry when a WDI drops the connection to it. This prevents other WDIs from interfering with

it. The Address Tag field is used to uniquely identify the block referred to.

- **Copy Lists.** For each entry in a Block List there is a corresponding segment which contains a Copy List. The Copy Lists identify the processors which have cached copies of this block. Each Copy List holds 32 four-byte entries, the format of which are:

Valid (1 bit)	Processor's logical WDI (30 bits)	Permissions (1 bit)
------------------	--------------------------------------	------------------------

The Valid field is set when an entry holds authentic data. The Processor Address field holds the Wafer and Logical CE number of the WDI belonging to the processor. The Permissions field says whether it is a readable or a writable copy.

The layout of the data structures was designed so that only two segments need be read to obtain a list of the cached copies of a given block. As a consequence, each set employs 64-way associativity and the maximum number of copies of a block which may be cached by processors is limited to 32.

The three sets in a fragment require 384 bytes of Block Lists and 24 Kbytes of Copy Lists. When a set runs out of free Block Lists and Copy Sets to use, a random invalidation and replacement policy is used.

5.3. Caching Mechanism

When a Wafer Disc Interface has been granted access to a block, it may read or write it freely until it either gives up the right, or has it taken away. However, even though an entry for the block will exist in the Block Lists, it may not be held in the cache. This is because the contents of the cache are maintained independently from the Block Lists. This makes more efficient use of the cache memory.

The WDI examines the Cache List of the set responsible for caching the required block. Each Cache List is held in a single 128 byte fragment and is composed of entries with the following fields:

Valid (1 bit)	Busy (1 bit)	Address Tag (14 bits)
------------------	-----------------	--------------------------

The Valid field is set when an entry holds authentic data. The Busy field can be used to lock the entry when a WDI drops the connection to it, for instance when a block is being loaded from optical disc. This prevents other WDIs from interfering with it. The Address Tag field is used to uniquely identify the block referred to.

- If the required block is in the cache set, the WDI can read or write it as required.
- If the block is not in the cache set and a write is requested, space is located in the cache, if necessary by flushing an existing block to optical disc using a random replacement policy, and the data is copied into it from memory.
- If the block is not in the cache set and a read is requested, the logical to physical optical block indexes held in the fragment are examined to locate the current copy of the block on the optical disc. These contain the physical optical disc addresses which correspond to the logical blocks.

The WDI transfers a block to or from optical storage by connecting to the processor with the optical disk and requesting it to perform the transfer. When this is completed, the processor acknowledges the fact by reestablishing the connection to the WDI.

The optical block indexes are arranged so that only blocks with the same hash value are referenced by a given index block. As a consequence, they have to be restructured whenever the number of PEs in the system changes, when fragments fail or wafers are added or removed, for instance. A few fragments are left unused on each wafer as spares. These can be reconfigured to take the place of failed cells.

The three cache sets require 384 bytes of Cache Lists and 103 Kbytes of logical to physical optical disc block translation tables. If each optical disc block address is eight bytes long, there is enough memory to address 12 K of optical disc blocks, or around 24 Mbytes of data. There are sufficient bits in the Address Tags to make full use of this.

5.4. Performance Evaluation

When estimating the performance of Wafer Disc transactions, it is necessary to specify the steps involved because of their potentially huge differences in latencies. Ignoring processing costs, the following times have been estimated for a five wafer system clocked at 40 MHz:

- The average time to construct a circuit between a WDI and a PE is 11 μ s.
- The time to locate and read a set's Block List and the block's Copy List, assuming it already exists, is 2.4 μ s. This is also the time required to write them back.
- The time to determine whether a block exists within a cache set is 1.2 μ s. This is also the time taken to perform a logical to physical optical disc block address mapping.
- The time taken to transfer a 2 Kbyte block between the Wafer Disc and processor memory, given that no blocks need be flushed from the cache, is 19.2 μ s.
- The time taken to determine the physical location of a logical block on the optical disc is 1.2 μ s.
- The time taken to transfer a 2 Kbyte block between the Wafer Disc and an optical disc, assuming a one Mbyte per second transfer bandwidth and a 80 ms seek and rotational delay, is 81.9 ms.
- It is assumed that once a circuit has been constructed to a processor, block invalidations can take place in around 10 μ s using custom WDI hardware.

From these, the time taken to obtain a copy of a block in the cache which requires no invalidations is estimated as 33 μ s. If the block is not in the cache, an extra 81.9 ms are required to perform the transaction required to load it from optical disc. Each invalidation will add a further 17.8 μ s to this and an extra 19.2 μ s is needed if a block of data needs to be transferred.

Assuming a 100% cache hit rate, Wafer Disc can support a peak of 30,000 transactions per second, a data transfer rate of about 58 Mbytes/second.

6. Conclusion

Wafer Disc is a proposed WSI device which combines shared storage and communication properties with high reliability. The device is well suited for use as a shared set associative cache for optical discs. As such it will provide high performance and scalability at a low cost. A Wafer Disc cache, however, is limited in size and does not support fine-grained data sharing.

Wafer Disc will be suitable for bridging the gap between multiprocessors and multicomputers. It will enable the construction of medium-scale shared memory multicomputer systems containing around 60 processors and 100 Gbytes of wafer memory.

Acknowledgement

The author would like to thank his supervisor, Professor Peter Osmon, for providing the basis of many of the ideas presented in this paper.

Appendix

A simplified (non-optimised) version of the coherence algorithm is shown below:

```
COHERENT_LOAD(block, required access permissions)
BEGIN
  make connection to PE;
  read Block List;

  IF an entry's ADDRESS field = block THEN BEGIN
    IF entry's BUSY field = true THEN
      drop connection and retry later;
    ELSE BEGIN
      entry's BUSY field := true;
      write Block List;
      read entry's Copy List;
    END
  ELSE
    IF all entries BUSY fields = true THEN
      drop connection and retry later;
    ELSE BEGIN
      IF all entries' VALID field = true THEN BEGIN
        select random entry from set;
        read entry's Copy List;
        entry's BUSY field := true;
        CHANGE_ACCESS(Copy List, none);
      END
      entry's ADDRESS field := block;
      entry's VALID field := true;
      entry's Copy List VALID fields := false;
    END
  END
  IF processor already has entry in Copy List THEN
    remove it from Copy List;
```



```

IF required permissions = none THEN BEGIN
  entry's Copy List VALID field := false;
  IF all VALID fields in Copy List = false THEN
    entry's VALID field in Block List := false;
ELSE BEGIN
  IF required permissions = read THEN BEGIN
    IF an entry's PERMISSIONS field = write THEN
      CHANGE_ACCESS(writer, read);
    ELSE IF all entries' VALID field = true in Copy List THEN
      CHANGE_ACCESS(random entry, none);
  END ELSE IF required permissions = write THEN
    CHANGE_ACCESS(Copy List, none);
  insert processor and permissions into Copy List;
END
entry's BUSY field := false;
write Block List to wafer;
write Copy List to wafer;

CACHE_LOAD(block);
END

CHANGE_ACCESS(list, new permissions)
BEGIN
  drop connection to PE;
  for (entries in list with VALID fields = true) BEGIN
    connect to processor;
    IF PERMISSIONS field = write AND block dirty THEN
      take copy of block;
    alter processor permissions on block;
    drop connection to processor;
  END
  make connection to PE;
  IF took copy of block THEN
    write block to cache;
END

```

References

- [Acc86a] M. Accetta et al, "MACH: A New Kernel Foundation for UNIX Development," pp. 64-75 in *Proceedings of the Summer USENIX Technical Conference* (June 1986).
- [And89a] P. Anderson et al, "Wafer Disk: A New Systems Architecture Component," Department of Computer Science, City University (April 1989).
- [And90a] P. Anderson et al, "The Feasibility of a General-purpose Parallel Computer using WSI," Internal Report; Computer Science Department; City University (1990).
- [Aub78a] R. Aubusson and I. Catt, "Wafer-Scale Integration – A Fault-Tolerant Procedure," *IEEE Journal of Solid State Circuits* SC-13 (June 1978).
- [Bug90a] H. Bugge et al, "Trace-Driven Simulations for a Two-level cache design in open bus systems," CH2887-8/90/0000/0250, pp. 250-259 (1990).
- [Car86a] R. O. Carlson and C. A. Neugebauer, "Future Trends in Wafer Scale Integration," pp. 1741-1752 in *Proceedings of the IEEE* (December 1986).
- [Che87a] G. Chesley, "Addressable WSI: A non-redundant approach," pp. 73-80 in *Computer Architecture News* (March 1987).

- [Cur89a] L. Curran, "Wafer-scale integration arrives in "disk" form," pp. 51-54 in *Electronic Design* (October 1989).
- [Del86a] G. Delp and D. Farber, "MemNet: An Experiment on High-Speed Memory Mapped Network Interface," Technical Report; 85-11-1R University of Delaware; Computer Science Department (1986).
- [Fle89a] B. D. Fleisch and G. J. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Operating Systems Review* (December 1989).
- [Gol86a] S. W. Golomb, "Optical Disk Error Correction," *Byte* (May 1986).
- [Hen90a] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc (1990).
- [Li86a] Kai Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. Thesis; Yale University; Department of Computer Science (1986).
- [Sys87a] Sequent Computer Systems, *Symmetry Technical Summary*, 1003-44447 Rev. A; Sequent Computer Systems, Beaverton, Oregon, 1987.
- [Who88a] D. Whobrey, "A Communications Chip for Multiprocessors," Computer Science Department; City University (June 1988).
- [Win87a] P. Winterbottom, "NCU: Network Control Unit. Preliminary Data Sheet," Computer Science Department; City University (December 1987).
- [Win88a] P. Winterbottom, "BIC: Buffer Interface Controller. Preliminary Data Sheet," Computer Science Department; City University (February 1988).
- [Win89a] P. Winterbottom and P. Osmon, "Topsy: An Extensible UNIX Multicomputer," Internal Report; Computer Science Department; City University (1989).

Performance Evaluation: The SSBA's at AFUU

C. Binot P. Dax N. DoDuc M. Gaudet

AFUU

Le Kremlin-Bicetre, France

ssba@afuu.fr

Abstract

The SSBA, Suite Synthetique des Benchmarks de l'AFUU, is an ever flourishing activity. The SSBA 1.21 is still widely appreciated and its influence is increasing everyday. The SSBA 2.0, its multiprocessing companion, is about to start its life. We present here a new status report about Benchmarking activities at AFUU.

The sudden birth of benchmarks is clearly a reaction to the appearance of open and standard systems, hardware and software together. Thus, within AFUU, the SSBA's are definitely an user's answer to the arrival of UNIX workstations, as a mean to (help) evaluate and purchase these machines [Bin88a].

The BENCHMARKS group of AFUU, created in March 87, then immediately proceeded to the creation of the SSBA: version 1.21 was "up and running" by the end of 1988, and as well with others developments of the group, its activities have been, in due time, reported here [Bin89a].

Right now, we wish to set again another time mark and present a new status report, which still deals with this SSBA 1.21 at its cruise speed, but also with the SSBA 2.0 at its early take-off.

1. SSBA 1.21

The SSBA 1.21 is now seriously and widely known and appreciated [Bin90a, Bor90a], but not thoroughly in the U.S. as we expected, or might understand, although its existence is fully exposed in specialised media (but perhaps not intensely enough?): comp.benchmarks, [Uni90a, Fad91a].

1.1. The Positive Sides

Globally, the acceptance by domestic then European users is really positive – it seems that there was an empty space and that the SSBA was filling it – and constitutes more than an encouragement to the work done. These users' comments and ratings reflect most truthfully their reactions, including their participations and views in [Tri90a, Tri91a].

Among the successes of the SSBA, the most striking one is its well-recognised status, domestically at least, as the one-and-only valuable test for UNIX machines, whatever the size (workstation or mainframe). More explicitly, the following argument is often put forward: a smooth and successful SSBA test on any machine may give some results not only about the quantitative performance of said machine, but also about the reliability and solidity of the operating system; on the other hand, if the test fails, then the worse can be expected and extra attention should be exercised toward this machine. In fact, in more than one case, neither the machine nor its operating system was at fault, but the quality of the human support in running the test, which may shed some light about the software maintenance and support of the vendor.

The second success is the scope of the SSBA: The SSBA 1.21 is meant to measure a wide range of characteristics of any UNIX machine [Wen91a], and up to now, is apparently the only suite of its importance and size and scope to do so if we are to exclude commercial suites whose values are very unequal.

The third interesting point is the specific test protocol, unique to the SSBA because of its charter. While SPEC tests are usually and normally run within vendor labs, while PERFECT results are usually obtained on well known systems (eg. dedicated machines well mastered by computer scientists), SSBA are always run on delivery machines, with released-version software, which is naturally the case since these runs are usually performed in France, with a few exceptions (always indicated) concerning too-new machines. However, since vendors rightfully want the best results, they are permitted to prepare the machines as best as they can, and stuff them with whatever software they may have, still the value of the run comes from the fact that the run is done in the presence of an AFUU representative who has to record eventual modifications to the SSBA source and the result of the run. In summary, such SSBA result may not represent the best value obtained (which is **not** the avowed purpose of the SSBA), but this value has been faithfully obtained and can be trusted, at least within the SSBA's context.

1.2. The Shortcomings

However, the SSBA does have a few shortcomings.

The most benign but not necessary innocuous is the Joy Mips value which is displayed firstly among the numerical results: when other values between competing systems are more or less equivalent, then this value may become a focus of attention, and this otherwise commonplace non-discriminant has become a real headache for many among us. The Joy Mips incident is only laughable if it is not a catalyst that reveals much more important underlying problems.

This may be an illustration of the following double fact: (1) the present display of the results, "la fiche synthétique", is too dense with raw data to be conveniently interpreted and easily understood; (2) no data reduction has been done (yet). While we're very conscious of fact (1) and on the way to solve it, we're still looking for an acceptable solution to fact (2). Graphic display is now a fact of life and we'll not be exempt of it; we're even more quite concerned that we definitely need some new way to present our results and that we need it very quickly, given the huge amount of pending results (we may already be too late right now). Furthermore, data reduction, and generally speaking data processing, is not an innocent or trusted operation [Gau90a]. SPEC results

for each machine hold entirely within one page: this is a graceful situation for the SPEC 1.0 suite which focuses only on CPU performances; however, the single value of merit, SPECmark, now become three, with two components: SPEC integer and floating point marks. PERFECT results now are also one full page, a-la-SPEC we may say so, but even when the acknowledged targets are only the CPU performances, we are presented with multiple values of merits: arithmetic, geometric, harmonic means, megaflops, instability. It might be that since such occurrences of presentations are happening and staying, we may take example from them as we're not going to reinvent the wheels when the learned and professional peoples have looked at the way and lead the path.

Thus to summarise, we may say that up to now, too much emphasis is put on data collection (we do have by now more than a hundred official and published results, listed in Appendix 1) and too few attention given to data interpretation, and this situation at present time becomes really critic. Furthermore, while the "synthetic results" can be understood only by sophisticated users, these results often land on some deciding manager's or non technical user's desk, generating hazardous and regrettable consequences and after-effects.

Other bugs are unveiled throughout intensive runs by users and partners. The first serious difficulty stems from the exposure of the SSBA to POSIX-compliant systems having an ANSI/C compiler: we may remember that, up to then, the SSBA has been (sucessfully) run in numerous tests and can de facto claim to be SVID (and X/OPEN) compliant [Dje90a], but then, a deficiency is still a bug. Another bug was discovered only recently, about the timing and thus the validity of the Saxer component of the SSBA, fortunately affecting only bsd-based systems [Gau91a].

On the whole, many other minor bugs or deficiencies are uncovered, mostly at the early stages of the SSBA (versions 1.1 then 1.2), and especially concerning the shell scripts which are the added value part of the SSBA, but which are not the easiest one to deal with. Nevertheless, although the importance of the problems may be mitigated somehow thanks to the fact that "we are learning by walking", we're not exactly glad of that situation nor inconscious of its seriousness: we're very constrained by our specific situation as will be explained some more down here.

1.3. The Specifics

There are many elements that can probably best illustrate the fundamental difference between our group and most (if not all) other Benchmarking groups.

The global and initial purpose of the SSBA is to measure the overall performance of a "mini computer" or a personal workstation running under an UNIX operating system. From our understanding, SPEC wants to highlight the Risc microprocessor (CPU) performances; PERFECT wants, initially at least, to squeeze the maximum performance out of a couple {algorithm and hardware} for a given big scientific application, while EURO BEN's main target is still the benchmarking of supercomputers, vector and parallel ones... (However, we are pleased to recognize that now that the CPU parameters are taken care, every group is looking at I/O and others parameters...).

Next, most if not all people working at SPEC, PERFECT, EURO BEN are dedicated and competent professionals whose main, and perhaps full

time, job is to understand and benchmark systems. SSBA peoples are very few and in a not-for-profit context, a la GNU, a very small team of dedicated individuals, who may be (or become) competent on the subject, but who, in no way, are allocated to even work part-time on this subject. Parts of the SSBA shortcomings surely come from this situation: no hardware or software, no testing procedure, no support (documentation or technical writing), no grant..., only good will and willingness to succeed.

All the above show that the environment within which the SSBA is working, "la vie associative", may be not the most efficient one. Nevertheless, the proof is done that something can grow out of this environment, and we're not the only ones that judge satisfactorily of all that has been done there.

2. The SSBA 2.0

2.1. The Creation Process

During the last 2 years we ran the SSBA 1.21, we observed two phenomenons related with the evolution of the technology and the UNIX Market. First, due to the phenomenal increase of power, some of the tests of the SSBA 1.21 became more and more inaccurate on a 60 mips machine, this is the sense of the history. On the other side, the UNIX operating systems provided by the manufacturers evolved towards better fonctionnalités (OLTP, real time...), sophisticated mechanisms (memory mapped files, disk arrays, multiprocessing...) and standardized interfaces (POSIX.1, XPG3...). Then, the UNIX market moved to satisfy more and more users' needs in regard with the commercial world. A new release of the SSBA had to be elaborated to take into account all these factors, so the release 2.0 project began.

We adopted the same methodology as the previous releases: preselection of interesting benchmarks all around the world, live runs on various machines with various configurations, presentation and discussion of the results inside the BENCHMARKS group, enhancements of the source codes to match the specifications of the SSBA, implementation into the suite, live runs and huge amount of testing. In parallel, we conducted the task of modifying or removing inaccurate tests from the 1.21 release, with the respect of providing a smooth migration path towards the 2.0 one, and we agreed to be more directive in the output of the results. During one year, until today, we worked together to achieve this goal, the result is here.

2.2. The Specifications

Our main concerns were to have a better I/O performance characterization and to take into account multiprocessor architectures which are clearly growing in the commercial market. We wanted to provide a more accurate tool for performance evaluation of UNIX systems and a public domain tool created by an open process without constraint and royalty, we are a users' group.

We received submissions of ideas and codes from Europe and the U.S., from end-users, universities and manufacturers, from 25 to 7000 lines of source.

At the end of the process our criteria to see if the SSBA 2.0 matches the initial specifications was to run it on 3 architectures, based on the same

CPU (80486), and coming from the same manufacturer: a PC/Workstation, a server and the same server with 2 cpu boards. The results we obtained encourage us to say that we succeeded.

2.3. The Benchmark

The SSBA 2.0 was designed to be POSIX.1 compliant with XPG3 extensions (like `fsync()` for example), the ANSI C compliance is currently not implemented. We continue to use some BSD stuff (like `gettimeofday()` for example) because there is no equivalent in the standard. As we have run the SSBA 1.21 on about 25 flavours of UNIX, we have seen what are the portability problems and we have fixed them in the 2.0 release.

Concerning the existing benchmarks, we move from the Dhrystone v1.1 to the Dhrystone v2.1, we slightly change the Whetstone and Linpack to give a little work to the optimizers, we added a C version of the Whetstone to care that Whetstone is "built-in" into some Fortran compilers. We increased some parameters in the Bsd, Utah and Tools stuff (we also add a compress, tar and dd component). We now use a 32 users simulation in the Musbus, instead of 8. The Mips benchmark is run several times during the execution of the SSBA and we calculate the average. We suppress the Musbus disk tests and we use `fsync()` to flush the buffers in the Saxer. We change the output of the results and we improve the configuration part. This kept us a little bit busy!

The new benchmarks are the Andrew (to stress in a real situation the filesystem), the Smith (the nasty fortran benchmark which send 200 Mb of data in a pipe), the Bonnie I/O test (which gives accurate and stable measure of the disk throughput), and a new Benchio developed by ourselves. This is the status at the end of June, when this paper was written; but the work is still in progress (in particular with the Iobench from Prime).

There are now 2 ways of running the SSBA: in the sequential mode: all the tests run one after another, as with the SSBA 1.21, for calibration and comparison; or in the "parallel" mode: all the tests, except the I/O ones and the Musbus, run together in background. In the parallel mode the global workload on the SSBA is distributed among the different processors, when they are available and as managed transparently by the operating system, and the total elapsed time gives an idea of multi-processing capabilities of the machine.

3. What Else?

The SSBA's are of course the most prominent item of our BENCHMARKS group's activities, but not the only ones. Beside the much needed and usual household works like collecting, analysing of other benchmarks for our library, gathering printed matters about benchmarking for our bibliography, giving advices and helps to newcomers (be it from small or big companies, vendors or big accounts, end-users or marketers...), we are going forward to collaborate more closely with our friends accross the Rhein and we hope that the result of this collaboration will be interesting and successful enough to be included in a next status report; hopefully soon.

References

- [Bin88a] C. Binot, P. Dax, and N. DoDuc, "Benchmarking in the AFUU," *EUUG Newsletter* 8(1) (Spring 1988).
- [Bin89a] C. Binot, P. Dax, and N. DoDuc, "Performance Evaluation: The SSBA at AFUU," in *EUUG Conference Proceedings (Brussels)* (Spring 1989).
- [Bin90a] C. Binot and Minis & Micros, *Les Benchmarks: Pour évaluer les performances des systèmes informatiques*, Feb 1990.
- [Bor90a] D. Borchers, "Suite in F," *iX Magazine*, pp. 102-105 (Feb 1990).
- [Dje90a] C. Djebbari, "La compatibilité des standards entre-eux: POSIX et ANSI/C versus SVID et X/OPEN, un exemple la SSBA 1.21," *Tribunix* 6(33), Data General France (7-8/90).
- [Fad91a] M. Faden, "User groups congregate under EurOpen "umbrella"," *Unix Today!*, Unix in Europe supplement (June 1991).
- [Gau90a] M. Gaudet, "Quid de l'Indice de Performance?," *Dossier Spécial: Benchmarks, Tribunix*, pp. 5-6 (March 1990).
- [Gau91a] M. Gaudet, "Pan et Rataplan sur le bec," *Tribunix* 7 (6-7-8/91).
- [Tri90a] de Tribunix, *Dossier Spécial: Benchmarks*, March 1990.
- [Tri91a] de Tribunix, *Dossier Spécial: Benchmarks*, March 1991.
- [Uni90a] Unigram-X, *French Bench User Group launches Benchmarking Suite*, March 1990.
- [Wen91a] M. Wenig, "Heilige Saeulen," *iX Magazine* (April 1991).

4. Appendix 1: SSBA 1.21 Results

Machine	Processor	Tribunix		Dossier Special	iX Magazine	
		no	date		no	date
Abacom Highspeed 386	i386				2	3-4/90
Acer 12000	i486				5	9-10/90
Alcatel APX 1000	i386	32	5-6/90	Mars 90		
Alcatel APX 2000	i386	32	5-6/90	Mars 90		
Alcatel APX 3000	i386	32	5-6/90	Mars 90		
Altos 486/S1000-008	i486				5	9-10/90
Apple Macintosh IIfx	M68030				6	11/90
Apricot	i486				5	9-10/90
Bull DPX 1000/30	M68020	27	7-8/89	Mars 90		
Bull DPX 2000/20	M68020	27	7-8/89	Mars 90		
Bull DPX 2000/27	M68030	27	7-8/89	Mars 90		
Bull DPX 5000/25	Risc 6	27	7-8/89	Mars 90		
Bull DPX/2 100	M68030	33	7-8/90	Mars 91		
Bull DPX/2 210	M68030	29	11-12/89	Mars 90		
Bull DPX/2 210	M68030	29	11-12/89	Mars 90		
Bull DPX/2 250	M68040	38	6-7-8/91			
Bull DPX/2 320	M68030	29	11-12/89	Mars 90		
Bull DPX/2 340	M68030	33	7-8/90	Mars 91		
Bull DPX/2 360	M68040	38	6-7-8/91			
Bull DPX/2 510	R6000	36	1-2/91	Mars 91		6/91
Cetia VMTV2c-25	M68030	36	1-2/91	Mars 91		
Cetia VMTV2c-33	M68030	36	1-2/91	Mars 91		
Cetia VMTV2d	M68040	36	1-2/91	Mars 91		
Cetia VMCB2	M88100	36	1-2/91	Mars 91		
Cheetah Gold 486/25	i486				5	9-10/90
Compaq 486/25	i486	34	9-10/90	Mars 91		
Compaq 486/33L	i486	34	9-10/90	Mars 91		
Compaq DeskPro 386/33L	i386	34	9-10/90	Mars 91		
Compaq DeskPro 486/33L	i486	34	9-10/90	Mars 91		
Compaq SystemPro 486	i486	34	9-10/90	Mars 91		
Control Data 4320	R3000	37	3-4-5/91	Mars 91		
Control Data 4360	R3000	31	3-4/90	Mars 90		
Control Data 4680	R6000	37	3-4-5/91	Mars 91		6/91
DG AV 300	M88100	33	7-8/90	Mars 91		
DG AV 310C	M88100	33	7-8/90	Mars 91		
DG AV 500	M88100				5	9-10/90
DG AV 5200	M88100	33	7-8/90	Mars 91		
DEC DS 2100	R2000	35	11-12/90	Mars 91		
DEC DS 5000/200	R3000	33	7-8/90	Mars 91	6	11/90
DEC Vax 9000		38	6-7-8/91			
Dell 433E	i486					1/91
DE SDX 500/25	i386	37	3-4-5/91	Mars 91		

Machine	Processor	Tribunix		Dossier Special	iX Magazine	
		no	date		no	date
ESD SDX 1000	i386	30	1-2/90	Mars 90		
ESD SDX 2000	i386	30	1-2/90	Mars 90		
ESD SDX 3000	i386	30	1-2/90	Mars 90		
ESD SDX 3400	i486	35	11-12/90	Mars 91		
Evans Sutherland ESV	R3000	37	3-4-5/91			
Goupil G50DX-33	i386	34	9-10/90	Mars 91		
Goupil G60-25	i486	34	9-10/90	Mars 91		
Goupil G60-33	i486	37	3-4-5/91	Mars 91		
HP/Apollo DN 10000	Prism	35	11-12/90	Mars 91		
HP 9000/340	M68030	31	3-4/90	Mars 90		
HP 9000/360	M68030	31	3-4/90	Mars 90		
HP 9000/370	M68030	31	3-4/90	Mars 90		
HP 9000/375	M68030	31	3-4/90	Mars 90		
HP 9000/400-S	M68030	35	11-12/90	Mars 91		
HP 9000/400-T	M68030	35	11-12/90	Mars 91		
HP 9000/425-S	M68040	36	1-2/91			
HP 9000/425-T	M68040			Mars 91		
HP 9000/720	PA 1.1	37	3-4-5/91	Mars 91		
HP 9000/720	PA 1.1	38	6-7-8/91			
HP 9000/815	HP-PA	31	3-4/90	Mars 90		
HP 9000/822	HP-PA	34	9-10/90	Mars 91		
HP 9000/825	HP-PA	28	9-10/89	Mars 90		
HP 9000/832	HP-PA	34	9-10/90	Mars 91		
HP 9000/835	HP-PA	28	9-10/89	Mars 90		
HP 9000/842	HP-PA	36	1-2/91			
HP 9000/845	HP-PA	31	3-4/90	Mars 90		
HP 9000/850	HP-PA	28	9-10/89	Mars 90		
HP 9000/852	HP-PA	36	1-2/91			
HP 9000/855	HP-PA	31	3-4/90	Mars 90		
HP 9000/860	HP-PA	35	11-12/90	Mars 91		
HP 9000/870-100	HP-PA	34	9-10/90	Mars 91		
HP Vectra 486-25	i486	34	9-10/90	Mars 91		
HP Vectra 486-33	i486	37	3-4-5/91	Mars 91		
Harris NH3800	M68030	31	3-4/90	Mars 90		
IBM 3090/180S		37	3-4-5/91	Mars 91		
IBM 6150-125	ROMP	27	7-8/89	Mars 90		
IBM 6150-135	ROMP	27	7-8/89	Mars 90		
IBM 6000-320	Power	34	9-10/90	Mars 91	3	5-6/90
IBM 6000-520	Power	34	9-10/90	Mars 91		
IBM 6000-530	Power	34	9-10/90	Mars 91	3	5-6/90
IBM 6000-540	Power	34	9-10/90	Mars 91		
IBM 6000-930	Power	34	9-10/90	Mars 91		
ICL DRS 6000	Sparc				4	7-8/90
IN2 IN6130	R2000	34	9-10/90	Mars 91		
IN2 IN6230	R3000	34	9-10/90	Mars 91		
IN2 IN6600	R3000	34	9-10/90	Mars 91		
IQUE 486/25T	i486				5	9-10/90
Itos 3000 WS	R3000					3/91

Machine	Processor	Tribunix		Dossier Special	iX Magazine	
		no	date		no	date
LERIS LRS-3025-16	R3000	34	9-10/90	Mars 91		
MCS IQU 486/25	i486				2	3-4/90
Mips M120_5	R3000	29	11-12/89	Mars 90		
Mips M2000_8	R3000	27	7-8/89	Mars 90		
Mips RS2030	R2000	27	7-8/89	Mars 90		
Mips RC6280	R6000	36	1-2/91			6/91
Mips Magnum	R3000	37	3-4-5/91	Mars 91		
Nixdorf 8810/90	i486				5	9-10/90
Nixdorf Targon/31M15	M68030	33	7-8/90	Mars 91		
Nixdorf Targon/31M45Mono	M68030	33	7-8/90	Mars 91		
Norsk Data Uniline	88100	36	1-2/91	Mars 91		
Olivetti CP 486/25	i486			3/91		
PRIME EXL 7330	R3000	34	9-10/90	Mars 91		
PRIME EXL 7360	R3000	34	9-10/90	Mars 91		
RegsX 8/32	M68030				4	7-8/90
SGI 4D/25	R3000	31	3-4/90	Mars 90	3	5-6/90
SGI 4D/80GT	R2000	27	7-8/89	Mars 90		
SGI 4D/240	R3000	27	7-8/89	Mars 90		
SGI 4D/260	R3000	35	11-12/90	Mars 91		
SGI 4D/320	R3000	35	11-12/90	Mars 91		
Scotty	i486				5	9-10/90
Solbourne 5/502	Sparc	31	3-4/90	Mars 90		
Solbourne 5/600	Sparc				3	5-6/90
Sony NWS-1580	M68030			Mars 90		
Sony NWS-1750	M68030			Mars 90		
Sony NWS-1850	M68030			Mars 90		
Sony NWS-3260	R3000					6/91
Sony NWS-3410	R3000				5	9-10/90
Sony NWS-3860	R3000			Mars 90		
Sun 3/260	M68020			Mars 90		
Sun 3/470	M68030	32	5-6/90	Mars 90		
Sun 4/370	Sparc	32	5-6/90	Mars 90		
Sun 4/470	Sparc	36	1-2/91	Mars 91		
Sun 4/490	Sparc			Mars 91		
Sun 1 (4/60)	Sparc	32	5-6/90	Mars 90	3	5-6/90
Sun 1+ (4/65)	Sparc	35	11-12/90	Mars 91		
Sun 2 (4/75)	Sparc	36	1-2/91	Mars 91		
Sun SLC (4/20)	Sparc	35	11-12/90			
		36	1-2/91	Mars 91		
Sun IPC (4/40)	Sparc	35	11-12/90			
		36	1-2/91	Mars 91		
Stardent 3010	R3000	37	3-4-5/91	Mars 91		
Tektronix XD88-30	M88100	30	1-2/90	Mars 90		
Telmat T2000/STE-30	M68020	32	5-6/90	Mars 90		
Telmat T3000	M68020	32	5-6/90	Mars 90		
Telmat T4000	M68020	32	5-6/90	Mars 90		
Telmat TR5000	M88100	37	3-4-5/91	Mars 91		
Terra 486	i486				5	9-10/90
Unisys 6000/55-B	i386	30	1-2/90	Mars 90		
SCO, IX, AIX, ATT, Eurix	i386				2	3-4/90

Tribunix is AFUU's newsletter;

Dossier Special refers to two annual Tribunix's special issues about Benchmarks;

iX Magazin is the leading German UNIX magazine.

5. Appendix 2: SSBA 2.0 Sample Results

RESULTS SYNTHESIS OF THE SSBA 2.0E (06/04/91)

=====

CONFIGURATION

Name: HP9000/825	CPU type: HP-PA	FPU type: BIT
Clock rate: 12.5 Mhz	Cache: 16Kb	RAM: 16 Mb
Disk: 600 Mb	Disk Controllers: 1 HP-IB	File System: BSD
O/S: HP-UX 7.00	C compiler: standard	F77 compiler: standard

SSBA 2.0E (06/04/91) run No. 1 : BEGIN at Fri Jun 14 16:34:55 METDST 1991
 Command C : cc -D_XPG2 -O I Command Fortran : f77 -O
 THE SSBA IS RUNNING IN THE SEQUENTIAL MODE
 unix : hp-ux SVR3
 define : -DTERMIO -DSysV
 machine : HP-UX hpuxe A.B7.00 U 9000/825 16754 (uname)
 whoami : root ttyr0 Jun 14 16:13
 value of HZ = 100 /* Ticks = 100 (times method) */ (calculation)
 SIGALRM check : 12 x 5 sec delays takes 60.00 wallclock secs (error 0.00%)
 number of Processes running on the system when the SSBA starts = 24
 number of Processes available for the user = 205
 memory available for a process = 16321 Kbytes
 compile time : 1182 Seconds
 executable average size : 89242.2 bytes
 number of loops done during the run of the SSBA = 257
 average number of logged users on the system = 1
 average number of processes created by the SSBA = 27
 SSBA 2.0E (06/04/91) run No. 1 : END at Fri Jun 14 21:10:45 METDST 1991

 total time for the SSBA complete run : 16550 seconds

GENERAL

dhrynr(without reg,without optimisation,1000000 iter): 11621 Dhrystones/sec
 dhrynr(with reg,with optimisation,1000000 iter): 17307 Dhrystones/sec
 dhryr(with reg,without optimisation,1000000 iter): 13635 Dhrystones/sec
 dhryr(with reg,with optimisation,1000000 iter): 17120 Dhrystones/sec
 average mips : 5.1 Mips/Joy
 total time for utah in seconds : real: 587 user: 269.53 syst: 83.51
 total time for tools in seconds : real: 425 user: 241.16 syst: 103.22
 total time for byte in seconds : real: 103 user: 6.97 syst: 68.17
 disk: 10.000 Mb (512 bytes io) throughput 274.20 Kbytes/sec 36.5 seconds
 the throughput mesured with a 10 Mb dd is : 151.51 Kbytes/sec
 total time for testc in seconds : real: 151.00 user: 98.06 syst: 15.58
 total time for memory in seconds : real: 315 user: 294.77 syst: 10.72
 total time for calls in seconds : real: 111 user: 9.43 syst: 83.9
 total time for pipes in seconds : real: 188 user: 1.62 syst: 108.31
 total time for fork/exec in seconds : real: 799 user: 9.73 syst: 676.52

TECHNICAL

cwhetd(double precision,with optimization,100M inst): 2568 KWhetstones/sec
 cwhets(simple precision,with optimization,100M inst): 2001 KWhetstones/sec
 whetd(double precision,with optimization,100M inst): 2744 KWhetstones/sec
 whets(simple precision,with optimization,100M inst): 3798 KWhetstones/sec
 linpackrd(double precision,with optimization,rolled): 0.4769 MFLOPS
 linpackrs(simple precision,with optimization,rolled): 0.6077 MFLOPS
 linpackud(double precision,with optimization,unroll): 0.5068 MFLOPS
 linpackus(simple precision,with optimization,unroll): 0.6603 MFLOPS
 doducd : precision=50.00049685 itera=5485 (correct) time=845.62 R=57
 ECH CASE ELNUM DLEN DTYPE CASETIME user syst
 smith: 1.0 78 0 0 0 3.7870 2393.93 112.71

COMMERCIAL

bytes	100			1000			4000		
BLOCKS									
t in s	Usr	Sys	Ela	Usr	Sys	Ela	Usr	Sys	Ela
Lib Wri	3.2	11.0	35.5	2.1	11.1	35.5	2.2	11.4	35.5
Worst	24.6	460.4	520.0	4.1	51.3	78.0	2.9	14.7	53.0
Best	2.1	7.4	35.0	1.6	7.0	36.0	1.9	6.4	35.0
Sys Wri	1.5	125.4	131.0	0.2	24.3	36.5	0.1	13.9	37.5
Worst	3.8	121.0	147.0	0.4	19.1	55.0	0.1	9.2	47.0
Best	2.2	109.3	115.0	0.2	16.8	32.0	0.1	8.8	34.0

```

-----Sequential Output----- ---Sequential Input-- --Random--
-Per Char- --Block--- -Rewrite-- -Per Char- --Block--- --Seeks---
bonnie      MB K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU /sec %CPU
           50  264 95.1  271 30.4  124 16.3  231 94.2  326 23.0  18.2 18.5
total time for andrew in seconds :   real: 3940   user: 54.36   syst: 354.41
Simulated Multi-user Workload Test :
32 Concurrent Users, each with Input Keyboard Rate of 2 chars/sec
Elapsed Time: 1238.67 seconds (standard deviation 4.249 sec)
CPU Time: 1045.80 seconds [ 345.23u + 700.56s ] (standard deviation 1.966 sec)

```

Near Real Time Measures of UNIX-like Operating Systems

Mario Cambiaso Susanna Delfino
Giancarlo Succi

DIST – Università di Genova, Italy
sumar@dist.unige.it

Abstract

UNIX was originally designed for multitasking and timesharing; however presently there is a lot of research going on trying to extend it to real time both for handling real time situations and because real time capabilities add robustness and reliability also to non real time applications. Several designs are proposed; this paper approaches the problem of benchmarking them. Standard tools as well as specific real time ones are not suitable for these systems: the former because too generic and the latter because too specific. Therefore a new set of tests is introduced and it is applied to a wide range of architectures. The results that have been obtained are interesting and, under certain point of views, quite surprising.

1. Introduction

UNIX was originally designed for multitasking and timesharing, therefore it did not have the features to meet the requirements of Real Time; however its wide usage and the fact that real time capabilities can add robustness, reliability and better performances also to non real time applications have led researchers to develop a *Real Time UNIX* system. Always more people are agreeing that in the long run this kind of systems will be the majority, and the one we have now just a subset of them [Mar90a]: the MASSCOMP 53 running RTU, HP9000/720 running HP-UX subset of them: the MASSCOMP 53 running RTU, HP9000/720 running HP-UX Version8, HP9000/350 running HP-UX Version 7, RISC System/6000 (RS/6000) running AIX 3 as real-time machines while DEC 3100 running ULTRIX 3.2, DEC 5000 running ULTRIX 4.0, Sun 4/280 running SunOS4.0.3 and Sparc 330 running SunOS4.1 for traditional implementations. It tries to distinguish their features together with their advantages and weakness mostly under the point of view of the user rather than of the architectural designer.

Traditional UNIX benchmarks cannot be used here, since they are not suitable for real time applications [Zed90a] nor it is feasible to use real time benchmarks, as they are too specific. Therefore this paper designs a set of new tests which should take into consideration both general purpose features and real time capabilities (introduced in Section 2).

The starting point consists in defining expressive parameters for the desired characterization: Section 3 is devoted to this task. From Section 4 to Section 7 there is a deep description of each single test together with the results which have been obtained on the considered architectures and their analysis; they have been elaborated after an analysis of the current solutions and according with POSIX [IEE88a] rules. Section 8 discusses globally the collected results and Section 9 draws some conclusions and outlines the planned future research. The appendix provides the results of a standard benchmark (Dhrystone).

2. Summary of Real Time Features

This section is a general and brief introduction to the features a system must present in order to be suited to real time. Its knowledge is the background which is assumed in order to fully understand this paper because this work is focussed on the performances rather than on the design strategies [Boh90a]. For these reasons this section can be skipped by those already inside this topic.

The requirements for real time operating system can be summarized as follows [Fur91a].

- *Support for scheduling of real-time processes.*
- *Preemptive scheduling.*
- *Interprocess communication and synchronization.*
- *High speed data acquisition.*
- *I/O support.*
- *User control of system resources.*

3. Near Real Time Measures

In order to understand fully the strategy which has been adopted it is necessary to know the limitations imposed by the working framework:

- First of all the measures and the tests must be fully portable, since it is needed that the result be significant and comparable on any machine, no matter the presence of real time capabilities;
- The measures slots must be performed using the only means inside the machines to provide easy and simple replications of the tests on different sites; this limitation has a heavy impact on the resolution.

All the primitives used in the tests refer to the POSIX 1003.4 Standard Committee [Hae90a]. The time measurements are performed using the interval timer, and, in particular, the system calls *setitimer()*, *getitimer()* and *gettimeofday()* [X/O87a]. Therefore the obtained resolution is connected to the system clock whose value is uniformed to 10 milliseconds for all tested machines.

Since the resolution does not allow to directly measure the timing of the analyzed events, like the *fork()* and the *exec()* system calls, or the context switches, most of the commonly used benchmarks for UNIX-like O/S take the approach of considering as atomic result the execution time of a high number of these events. Collecting a set of this kind of values, under certain hypotheses, it is possible to extract informations such as average, variance and distribution of these events. In this context this approach cannot be taken for two main reasons:

1. The above mentioned hypotheses to determine the distribution are usually not satisfied [Gla88a];
2. Under a real time point of view, the interest is on worst cases rather than on the average ones.

Consequently the approach taken in these tests is to focus only on the statistics of the measurable cases: they correspond to the worst cases. A possible result is then the distribution of the events over a threshold which is determined by the timer granularity.

The set of chosen measures are called *near real time* to distinguish them from those already existent, stressing their peculiarity of being able to work both on standard UNIX systems and on the real time ones. The description of the target system examines the following topics:

1. The reliability of the *signal* mechanism,
2. The context switch latency in critical situations,
3. The I/O throughput,
4. The computational speed of the processor.

4. Reliability of Signals

There may be cases in which a process must handle asynchronous events with a deterministic behavior: delays, omissions or mistakes may result in crashing the application [Tan87a]. This is especially true in a real time framework. Actually there are critical situations in which it is possible that the system is not able to assure a correct behavior. The standard POSIX interface, for instance, handles at most one signal, while the system is already serving another one. Furthermore other problems arise when more signals are sent to the same process inside the same quantum or when the process is not scheduled between two sends.

The test, called **2KILL**, is organized in the following way.

1. A C executable which sends two signals, one immediately after the other, to the same process and counts the number of calls to the manipulator induced by them: its result is regard as **SUCCESS** if both the signals are handled, **FAILURE** otherwise.
2. A shell script which iterates n times the C executable and determines the number of **FAILURES**. This operation is repeated m^\dagger times in order to build a file containing the number of failures.
3. An *awk* script which produces the **FAILURES** statistics.

Figure 2 can help understanding this mechanism. A process forks a son which communicates with it using *kill()* and is synchronized with a pipe. When the son has completed to set up its manipulators for the signals with the *sigaction()* system call, the father sends it two *SIGUSR1* and one *SIGUSR2*. The manipulator associated with *SIGUSR1* increments an integer variable which value is printed by the one associated with *SIGUSR2*. The benchmarks is executed in different conditions to force the system to schedule the process which has to serve the signals, such as giving the son a higher priority than the father (with the *nice()* system call) or as imposing real time priorities (when allowed) [Qua85a]. The results of this test are quite interesting for some machine: the Masscomp RTU, for instance, showed a remarkable difference between what happens with standard priorities and with the

[†] The n and m value are assigned on the basis of statistic considerations.

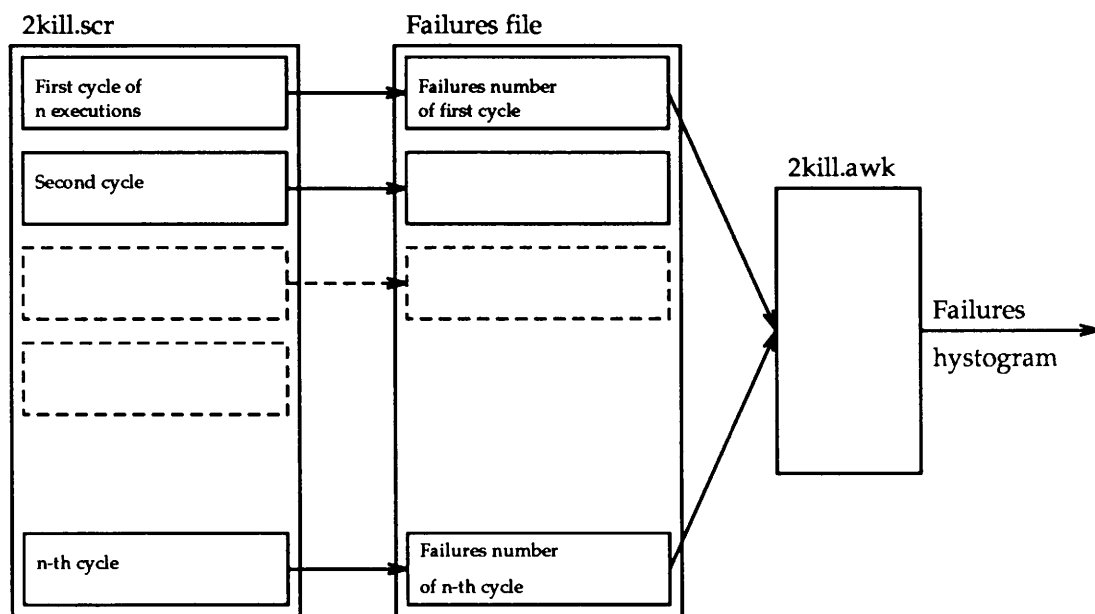


Figure 1: Diagram of test 2KILL

real time ones while in all the other machines the difference between the two cases is not so evident. For some machines the values range is more wide than other ones, showing a less uniform behavior. Furthermore in all the results got using real time priorities there are not *FAILURES*.

The results are presented through the histograms that show the distribution of the failures in comparison with the number of failures.

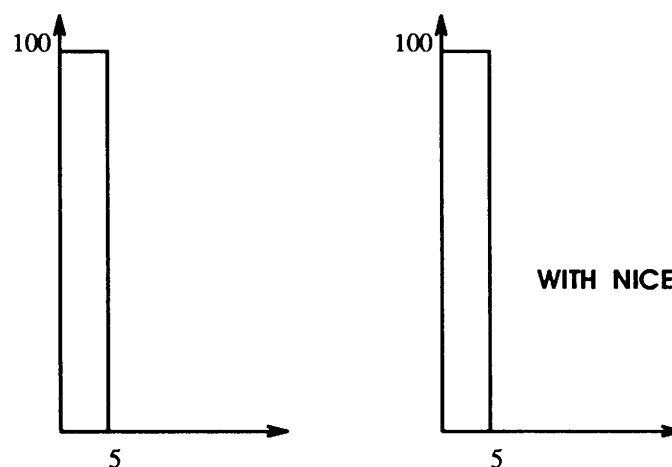


Figure 2: HP-UX V 8 failures histogram

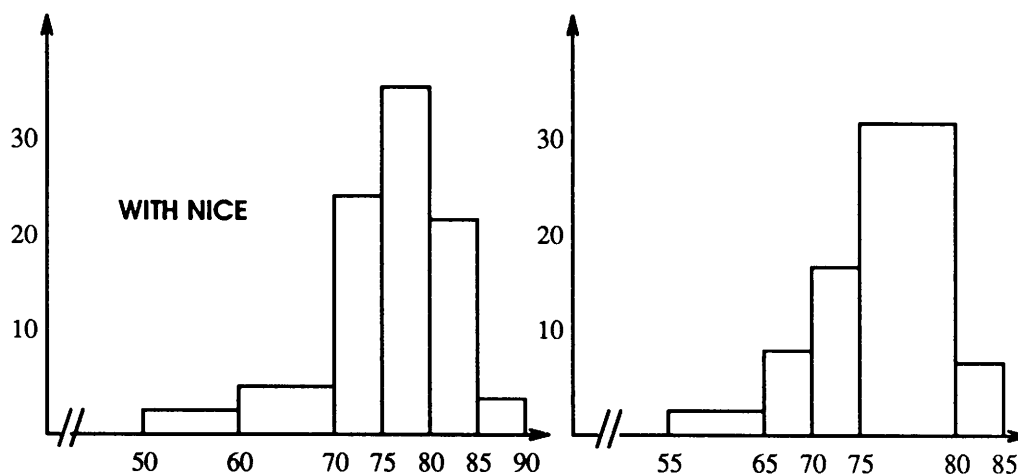


Figure 3: SunOS 4.0.3 failures histogram

5. Context Switch Latency

The time needed for a context switch [Zim89a] is usually lower than the granularity of the timer, therefore its value can be measured only under certain load conditions. One of the heaviest situations occurs when the processes competing for the CPU must execute time consuming system calls. The test called **1CSPIPE** tries to reproduce this situation: the context switch occurs between two processes, while other processes are executing increased system calls, like the **open()** on a long path. As explained before, here the interest is on worst cases: the result is the percentage of latency time over 10 milliseconds and their distribution.

The test is organized in two blocks: the first induces the context switches and measures the latencies, while the second loads the system. The first part is organized, like the tests for the signals, in 3 scripts: a C one which computes some values, a shell one which repeats the C one and an awk one to compute the statistics. Here the C code induces a context switch making two processes depending one on another and

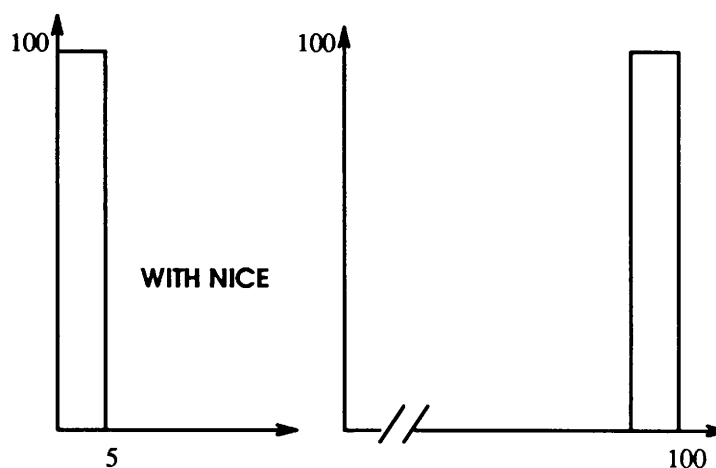


Figure 4: MASSCOMP 53 failures histogram

Brand	Machine	O/S	Percentage
HP	HP9000/720	HP-UX 8	0.0 %
HP	HP9000/350	HP-UX 7	1.5 %
Sun Micros	Sun 4/280	SunOS4.0.3	2.4 %
IBM	RS/6000	AIX 3	2.8 %
Sun Micros	Sparc 330	SunOS4.1	6.5 %
Digital	DEC 3100	ULTRIX 3.2	7.3 %
Masscomp	MASSC. 53	RTU	17.0 %

Table 1:

forcing any of the two to be waiting for the other to write on a pipe. The system call *gettimeofday()* is used to measure the timing. To store the results of these two tests, which are repeated several times, two arrays are used, and the arrays are written at the beginning of the test with dummy values to prevent page faults and extra context switches.

The second part is formed by shell scripts which load a process that opens a file on the tree, which was previously built with the depth 100. The results are show in Table 1.

To complete the description of this test the following figures in which are showed the histograms of the worst cases. Is interesting to mark

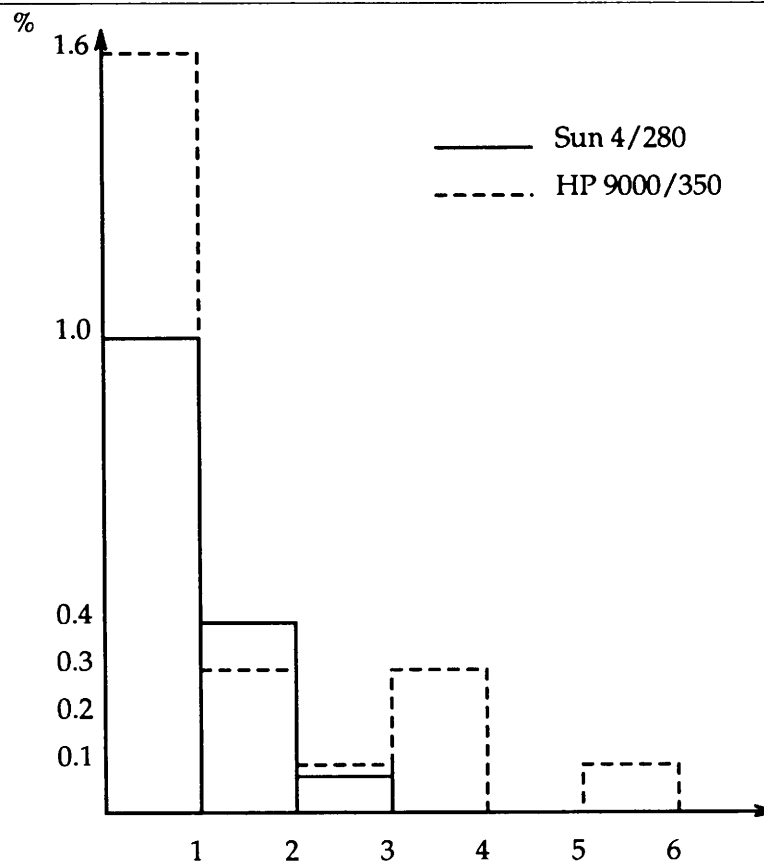


Figure 5: HP 9000/350 and Sun 4/280 latency times

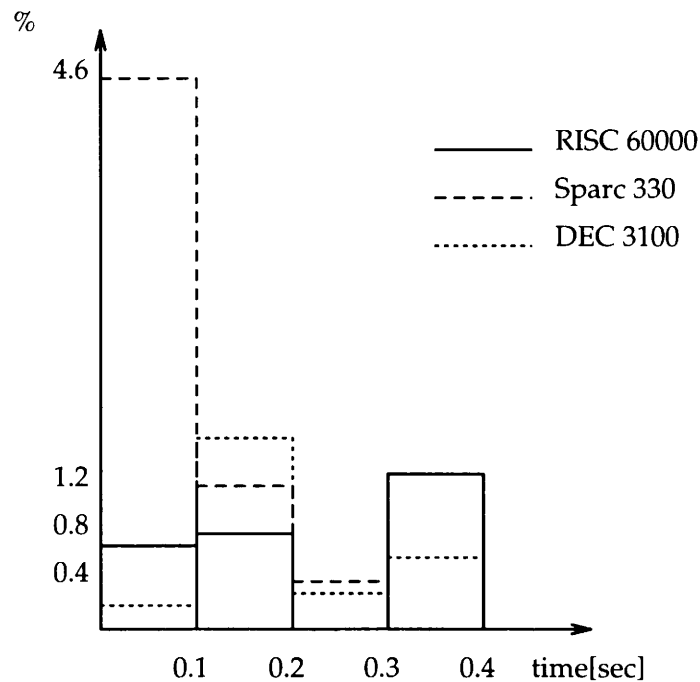


Figure 6: RS/6000, Sparc 330 and DEC 3100 latency times

the HP 9000/720 results: all the latency times are smaller than the threshold value because the time needed to the *open()* system call is very short.

6. Throughput Rate

As it is mentioned, I/O may play a critical role in real time applications: there may be the need of transferring big portions of data between primary and secondary memory in a fast and reliable way. Therefore a measure of how well this feature can be exploited, helps understanding the feasibility of a system for real time. The selected approach consists of measuring the amount of data that can be transferred in a fixed amount of time. The test is called **THPUT** and uses the system call *write()* for asynchronous writing. It is organized in three modules:

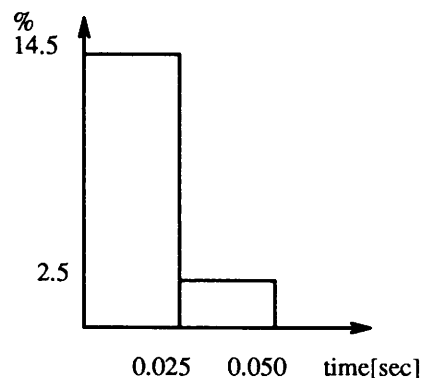


Figure 7: MASSCOMP 53 latency time

Machine	S.O.	Throughput Kbyte/sec	Saturation Mbyte
MASSC. 53	RTU	260	0.15
HP9000/350	HP-UX 7	600	1.00
DEC 3100	ULTRIX 32	800	1.00
Sparc 330	SunOS4.1	1000	10.00
Sun 4/280	SunOS4.0.3	1800	16.00
DEC 5000	ULTRIX 0.4	2200	2.00
RS/6000	AIX 3	3000	10.00
HP90000/720	HP-UX 8	4500	7.40

Table 2: I/O Throughput

- A C code having as input the length of the time interval for the test and producing as output the throughput for that interval. The writing for a fixed amount of time is performed by an infinite loop of *write()*s of 1024 bytes blocks, interrupted at the end of the interval by a signal.
- A shell script iterating the previous code for a fixed interval several times so that to reduce the effect of the I/O noise, since even a low level of I/O noise may have a high impact.
- An awk script that extracts statistics informations about the transfer rate between RAM and disks.

The results shows the presence of saturations connected to the dimensions of the buffer cache. The results are show in Table 2.

7. CPU Speed

In order to give an exhaustive paradigm of analysis to be used by any UNIX user, some means for computing the performances of this machines also in terms of computing speed are here introduced. For the sake of completeness the results of the standard Dhrystone benchmark are presented in Appendix A. Two are the tests that are used here. The first one **CYCLE** computes the time needed for a cycle (test, decrement and jump) under different conditions, like the optimized compilation or the usage of registers. The second test is called **MINSQUARE** and gives the minimum squares estimate of the time needed to execute a single cycle of an empty while loop, corresponding again to a test, a decrement and a jump. **MINSQUARE** is performed repeating **CYCLE** for many different values of decrements. The results is placed in a (number of cycles, time) cartesian system and then the regression line is determined using the recursive minimum squares method: the intersection of the line with the y axis gives the estimate of the timer overhead, since in this case nothing else is done, and the slope of the line represent the time needed to perform a single cycle. The algorithm for this test is organized in three phases:

Brand	Machine	O/S	Slopes
HP	HP9000/720	HP-UX	4.01e-8
Digital	DEC 5000	ULTRIX 4.0	8.12e-8
IBM	RS/6000	AIX 3	1.00e-7
Sun Micros	Sun 4/280	SunOS4.0.3	1.80e-7
Sun Micros	Sparc 330	SunOS4.1	1.20e-7
HP	HP9000/350	HP-UX 7	4.10e-7
Masscomp	MASSC. 53	RTU	5.08e-7

Table 3:

1. The array for containing the data is loaded into primary memory through writing in its different block (to be sure to not have any page fault or context switch during the test);
2. The *CYCLE* test is repeated for different values of iterations;
3. The collection of values obtained is given as input to the function which computes the regression line.

The *CYCLE* test supply information like *MINSQUARE* test, then in this paper there are only the obtained values for slopes; the timer values are not introduced because they are not stable since too sensitive to any perturbation. The results are show in Table 3.

The last test is performed on the whole *MINSQUARES*, since it is itself a set of both floating point and integer operations, so that can give some figures of the computational power of the systems.

8. Discussion

This tests have quite interesting results. Usually the real time systems presents better indices than the non real time ones, but not always. For instance the MASSCOMP 53 had excellent results in the *2KILL* test with *nice()*, but it behaved quite poorly in the *THPUT* and in the *ICSPIPE* one were was outperformed by any other system. In *ICSPIPE* also the IBM RS/6000 was exceeded by the Sun 4/280 which is not real time.

The HP 9000/720 performed excellently in almost all the tests, besides in those about the computing rate (*CYCLE*, *MINSQUARE* and Dhrystone[†]).

It is interesting to observe behaviors of different versions of the same Brand. The new HP-UX V8 has much better performances of the HU-UX V7, however this is not always true for SunOS 4.1, which in some cases is outperformed by SunOS 4.0.3.

9. Conclusion and Future Research

The aim of this paper was to give a paradigm for comparing real time UNIX like operating systems and the standard ones under a user point of view. Some tests has been introduced and their results explained.

[†] Look at Appendix A.

The plan is now to broad further the set of machine under tests, despite here a wide range has been considered. The next step is to take into account also the scheduling algorithms; some work has already started in this direction.

10. Acknowledgments

This work has been performed in the LISA[‡] framework. The authors thank Prof. Joy Marino for his enlightning ideas and the HP and DEC for allowing us using their machine and IBM for providing reference materials. An especial thanks goes to HP which allowed to test its HP 9000/720 before its market presentation.

Appendix A – Dhrystone

In this section are presented the Dhrystone benchmark results. Each execution is a cycle of 500000 iterations and it is repeated with four conditions:

- Using normal and optimized compilation;
- Declaring variables as normal and register.

The values showed in the following table represent this four different execution and are expressed in dhrystone/second. Again the abbreviations "+reg" and "-reg" indicate the present or not of the variables defined as register; "+O" and "-O" refer to the optimized compilation.

Machine	S.O	+O +reg	+O -reg	-O +reg	-O -reg
MASSC. 53	RTU	5425	5419	5280	5289
HP9000/350	HP-UX 7	9012	9028	4617	4341
Sun 4/280	SunOS4.0.3	20433	20292	12531	10475
Sparc 330	SunOS4.1	24578	24502	16897	12649
DEC 5000	ULTRIX 0.4	46681	43940	37306	34763
RS/6000	AIX 3	54768	54871	24548	25203
HP9000/720	HP-UX 8	91407	92764	71123	56306

References

- [Boh90a] K. A. Bohrer and J. T. O'Quin, *Enhancements to the AIX Kernel for Support of Real Time Applications*, International Business Machines Corporation (1990).
- [Fur91a] B. Furht, D. Grostick, D. Gluch, G. Rabbat, J. Parker, and M. McRoberts, *REAL-TIME UNIX SYSTEM Design and application Guide*, Kluwer Academic Publishers (1991).
- [Gla88a] S. A. Glanz, *Statistica per discipline bio-mediche*, McGraw-Hill Libri Italia (1988).

[‡] The Laboratory for the Interoperability of Open Systems.

- [Hae90a] J. S. Haemer, "Standard Update, IEEE 1003.4: Real Time Extension," *comp.std.unix* 18(492@longway.TIC.COM) (1990).
- [IEE88a] IEEE, *IEEE Standard Portable Operating System Interface for Computer Environment*, The Institute of Electrical and Electronic Engineers Inc (1988).
- [Mar90a] G. A. Marino, "Sistemi Aperti e Tempo Reale: dalle soluzioni ad hoc agli standard," in *The Automated Factory Show*, DIST – Università di Genova (February 1990).
- [Qua85a] J. S. Quarterman, A. Silberschatz, and J. Peterson, *4.2BSD and 4.3BSD as Examples of the UNIX System*, Department of Computer Sciences, University of Texas (1985).
- [Tan87a] A. S. Tanenbaum, *OPERATING SYSTEMS: Design and Implementation*, Prentice Hall Inc (1987).
- [X/O87a] X/OPEN, *X/OPEN Portability Guide, System V Specification, system calls and libraries*, The X/OPEN Group Members (1987).
- [Zed90a] H. Zedan, *REAL-TIME System: Theory and Applications*, York University (1990).
- [Zim89a] M. Zimmerman and N. Nachiappan, "What's Real With Real Time UNIX System?," *The UNIX Technology Advisor* 1(1) (1989).

Steppingstones: Some Remarks on Measuring X11 Performance

Werner Kriechbaum

IBM AIX FSC, München, Germany

werner@ibm.de

Abstract

After a short overview of *xbench* and *x11perf* this paper gives an early report on *steppingstones*, an Xperformance tool under development at IBM's AIX FSC in Munich. Line-drawing performance is used to illustrate the approach to Xperformance taken by *steppingstones*, as well as to illustrate some aspects of Xservers relevant to performance but missed with other performance-measurement tools. The final sections briefly describe multi-user extensions currently under development and the data-reduction techniques currently in plan for further refinement of the measured results.

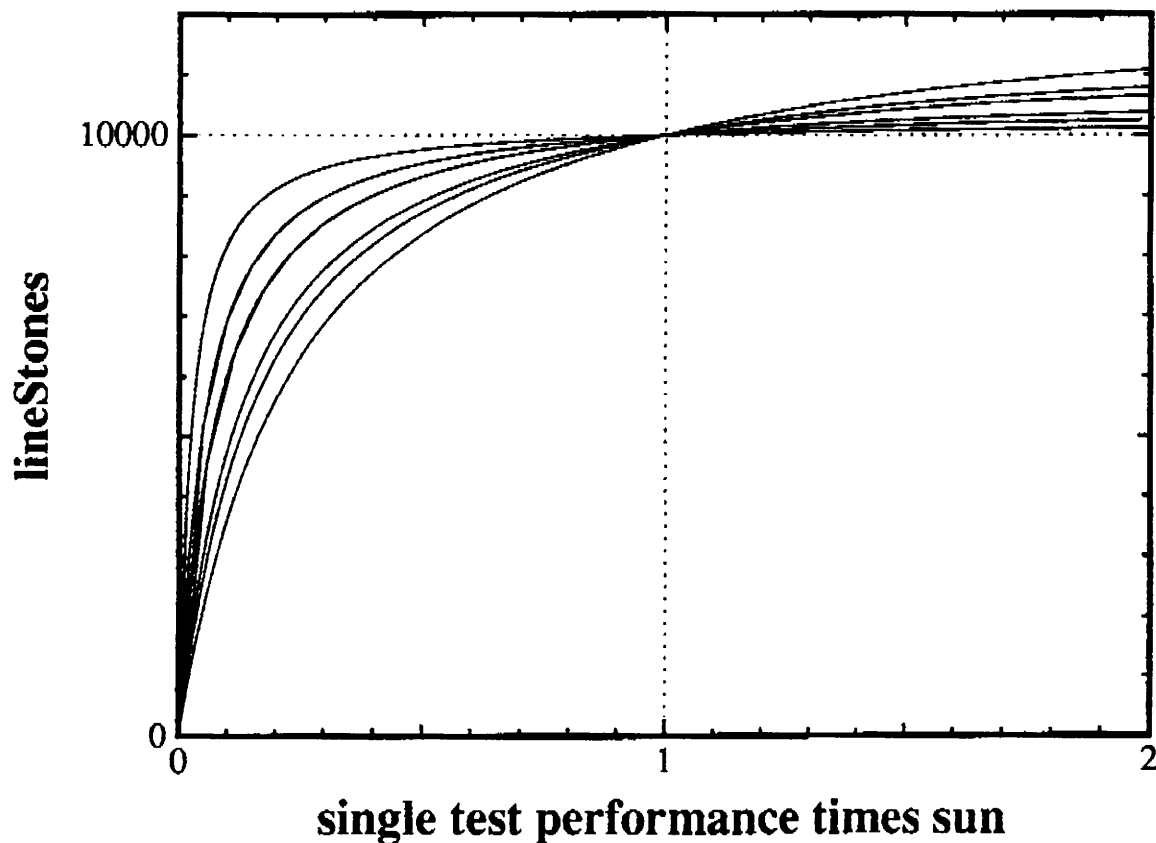
1. Introduction

Performance measurements are used for a variety of goals, conflicting at times: Developers want detailed measurements of the timing behaviour of their code under a variety of conditions to give them hints for further performance-tuning. End-users on the other hand tend to focus on a single "magic number" which should sum up system performance under all conceivable instances and should allow them to do a realistic comparison of different platforms as well as an estimation of the runtime requirements of their application. These trends are reflected by two X-performance-measurement tools currently in widespread use:

- *xbench* tries to sum up its results by providing Xstones as a single magic number, which is supposed to characterize all aspects of an X-protocol implementation in a fair and comparable way.
- *x11perf* measures a variety of performance aspects of individual X protocol requests without attempting to summarize its results in a few performance figures and thus does not provide simple means of comparison.

1.1. *xbench*

xbench performs up to 88 tests executing various Xprotocol calls. 40 of these tests are used to compute a summary value called Xstones. Xstones attempt to rate an Xserver relative to a SUN 3/50 running untuned MIT release 3 code, the performance of which is set to an arbitrary value of 10000 Xstones, which are computed using the formula

Figure 1: *lineStone Sensitivity*

$$\text{stones} = \frac{\sum_{i=1}^n \text{weight}_i}{\sum_{i=1}^n \frac{\text{sun}_i}{\text{measured}_i} \times \text{weight}_i} \times 10000$$

where i is a test-index, sun_i the performance of a SUN 3/50 (e.g. lines/sec), measured_i the performance of the tested server, and weight_i a set of weights adding up to 10000 and assigned to match the frequency with which certain Xcalls are used (text-performance e.g. has a weight of 3000 and line-performance for simple lines a weight of 1000).

Despite the arbitrary setting of weights which is unlikely to match a variety of different applications,[†] the formula shows a somewhat non-linear behaviour (*cf* Figure 1). Let us assume an Xserver which has the same or a multiple rating than a SUN 3/50 in all tests except one. In such a case, the server receives only a very small benefit in its Xstone rating when it exceeds the SUN's performance in the differing test. But it suffers a rather huge penalty for poorer performance.

[†] To be fair it should be noted that this short-coming is already discussed in the manual accompanying *xbench*: "Of course, depending on your applications, the rating could differ from your personal feeling. If you are running a CAD application doing only line-drawings, the line weight is too small compared to text/bitblt rating. ... I'd better not included the weighting stuff - I can see the flames on the net ..."

Both, the arbitrary weighting scheme and the non-linear behaviour, motivated by the performance profile of a SUN 3/50, make Xstone-ratings difficult to interpret and allow only a limited prediction of an application's performance. Used as a single "magic number" by people not aware of the intricacies of Xperformance, they may be misleading if not meaningless.

1.2. x11perf

x11perf exercises most of the Xprotocol calls using 147 elementary tests. Since its authors believe, as stated in the documentation, that "No single number or small set of numbers are sufficient to characterize how an X implementation will perform ..." *x11perf* just reports the results for each individual test giving performance values (e.g. lines/second and milliseconds per line) for each second and average values for the complete run (taking 5 seconds in the default case). This huge tabular output (almost 1000 lines using the default settings) may be appropriate for developers, but is a little bit difficult to digest for normal beings. Despite this excessive testing *x11perf* leaves some areas crucial for an application's performance unexplored: lines and segments are always tested with polycalls (e.g. *XDrawLines* instead of *XDrawLine*) using almost always a repeat count of 1000 elements. Using smaller repeat counts can unveil rather dramatic performance changes, at least with some server implementations (cf Figure 5).

1.3. steppingstones

steppingstones is an X11 performance test suite – still under development – based on *x11perf* and attempts not only to characterize the server performance but to allow a more reliable assessment of an application's performance. The major design goals of the test-suite are

1. The tests should exercise almost all Xprotocol calls using a reasonable number of parameters to collect not only isolated performance values (like e.g. performance for segments of length 1, 10, 100 and 500) but to allow an estimate of the functional dependencies between performance values and the input parameters.
2. Results should be provided not in a tabular form, but in a simple, perceivable and comparable form like line-drawings or scatter-plots
3. Besides a graphical representation, the data should be subjected to a statistical analysis which reduces the large number of timing results to a small set of numbers which characterize performance. These techniques should not make any assumptions such as assigning arbitrary weights to measured timing results.
4. Since most application programs written today don't use native Xprotocol calls but widget sets and toolkits like e.g. Motif, a layer of the test-suite should exercise such X11-derived functions.
5. The tests should address the multi-user aspect of Xperformance.

At the time of this writing, the current version of *steppingstones* consists of a set of yet-to-be integrated modules addressing above mentioned design goals (1), (2) and partially (3) and (5). Routines testing the Motif-toolkit, developed by Franz Pestenhofer from our group, will be presented at the GUUG '91. We hope that a beta-release of the Xprotocol set will be available to the public domain either late this or early next year.

The following section discusses some of the results obtained with the test-suite so far.

2. Phenomenology of Xperformance: Characterizing Line-Segment Performance

Lines, or to be more specific, solid line segments of linewidth 0, are used as an example to illustrate the approach taken by *steppingstones*. Lines are by no means the most important or most often used datatype in X, but they are rather simple and should exhibit less complexity like e.g the drawing of ellipses or display operations.

2.1. Performance Effects of the Segment Size

Figure 2 shows for 7 different server implementations the time in milliseconds taken to draw lines of various length. Besides the bizarre but reproducible behaviour of server A, all other Xservers show upto two distinct components in their timing behaviour:

- For short lines, the time needed to draw a line is independent of the length of the line.

Line Performance vs Length

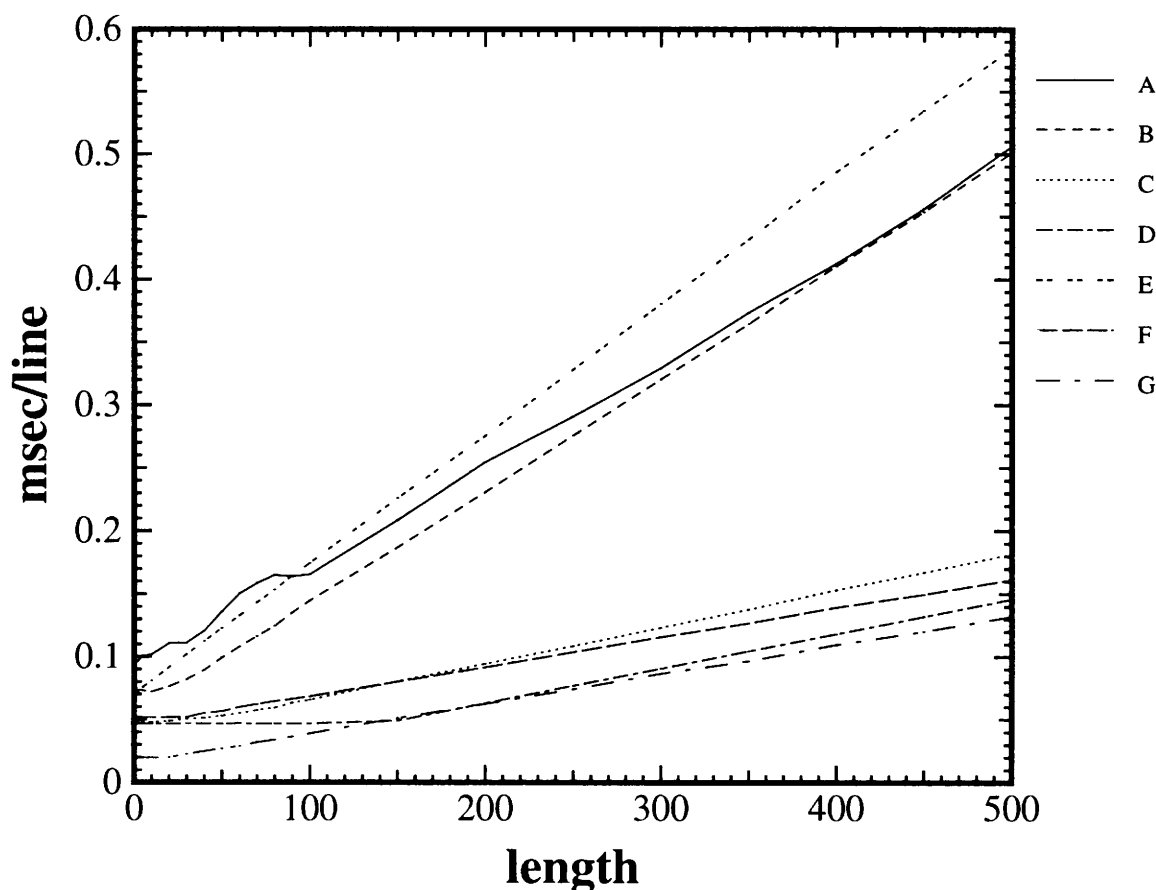


Figure 2: Line Performance vs. Length

Linear spline approximation

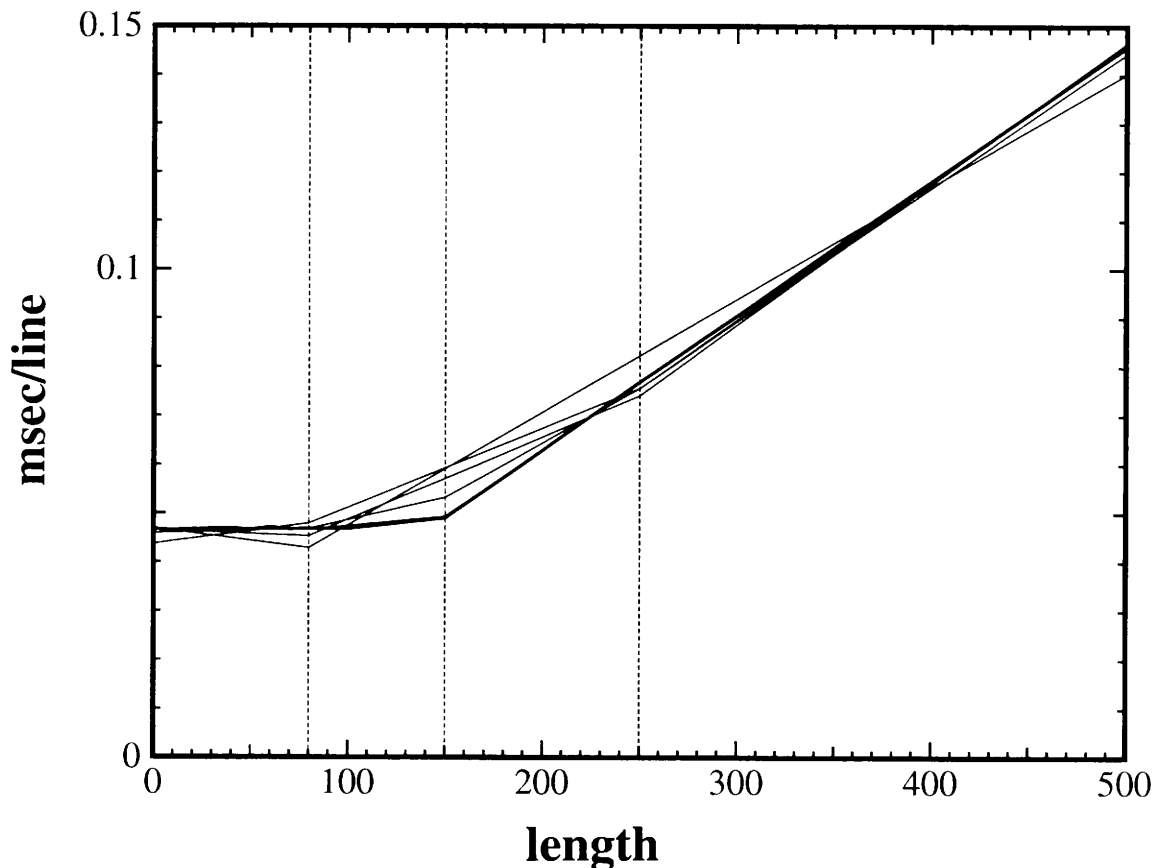


Figure 3: Linear Spline Approximation

- Starting with a length between 1 (server E) and 150 (server D) the time needed is linearly dependent on the length of the line-segment.

To describe such a behaviour one needs to measure domain, slope and intercept of two regression lines. Various methods exist to deal with such a problem, the variant used by *steppingstones*, as illustrated in Table 1, is essentially a split-and-merge algorithm: In a first step, the data is fitted by a first order spline approximation. Such a fit results in a set of breakpoints, called knots, describing the domain, and a set of spline-coefficients describing the polynomials [Lan86a]. The quality of such an approximation depends crucially on a smoothing factor, which governs the maximal deviation of the approximation from the data. A factor too large gives a very poor approximation (see Figure 3), a factor too small results in an interpolation instead of an approximation. A reasonable value can be found by decreasing the smoothing factor and stopping the approximation as soon as the sum of the squared deviation between approximation and data reaches a plateau [Ich76a]. This approach tends to generate more knots and therefore more piecewise polynomials than necessary to describe the data. Therefore, in a second step, the slopes of adjacent lines are compared and the segments merged whenever their slope is equal. The final approximation describes a server by two linear functions:

	Original Knots	Final Knots	Constant Part	Linear Part
A	5, 10, 20, 30, 40, 50, 60, 80 100, 150, 200, 250, 300, 350, 400	—	—	$0.0806 + 9.341e-2 n$
B	5, 10, 30, 60, 80, 100, 150, 250, 400, 450	—	—	$0.0599 + 8.719e-4 n$
C	30, 60, 80, 250	30	$0.0472 + 9.888e-5 n$	$0.0382 + 2.853e-4 n$
D	80, 150, 250	150	$0.0464 + 1.022e-5 n$	$0.0481 + 1.696e-4 n$
E	80, 150, 200, 250, 300, 350, 400	—	—	$0.0708 + 1.032e-3 n$
F	30, 60, 80, 250, 400	30	0.0518	$0.0454 + 2.325e-4 n$
G	10, 30, 80, 250, 350, 500	10	0.0198	$0.0155 + 2.339e-4 n$

Table 1: Server performance for line-segments. Random orientation, blocking factor 1000

- A constant part which gives the server performance for those lengths where performance is governed by Xprotocol overhead and
- A linear part which describes the domain in which the server is pixel-bound.

2.2. Performance Effects of the Segment Orientation

Figure 4 shows the results obtained by drawing a line segment of length 250 pixels at varying orientations. Despite algorithms for line drawing are such well-known, that they appear in virtually every introductory book on graphics programming, not all servers take advantage of the possible optimizations for horizontal and vertical lines and none seem to use an optimized method for lines at 45 degrees. Despite the effects on performance should be rather small, they will be noticeable in some applications relying heavily on horizontal and vertical lines, especially when using the server which does these lines almost thrice as fast.

2.3. Performance Effects of the Repeat Count

Polycalls like *XDrawSegments* send only one Xheader to transfer a list of coordinates describing multiple drawable objects. For line-segments *x11perf* uses a blocking-factor of 1000 segments per call. But quite a lot of application programs, especially those drawing grids, send each segment on it's own. In such cases one would expect a considerable slower line-performance due to the increased X overhead. As can be seen from Figure 5 this is indeed the case, but one of the tested servers offers an additional surprise: it shows marked oscillations for small blocking-factors. A 250 pixel line takes 0.22 msec using a blocking-factor 2, 0.317 msec using 3 and again 0.273 msec using a factor of 4. It should be noted that this behaviour first is reproducible and secondly not confined to lines: this server shows exactly the same pattern when rectangles are tested.

Line Performance vs Orientation

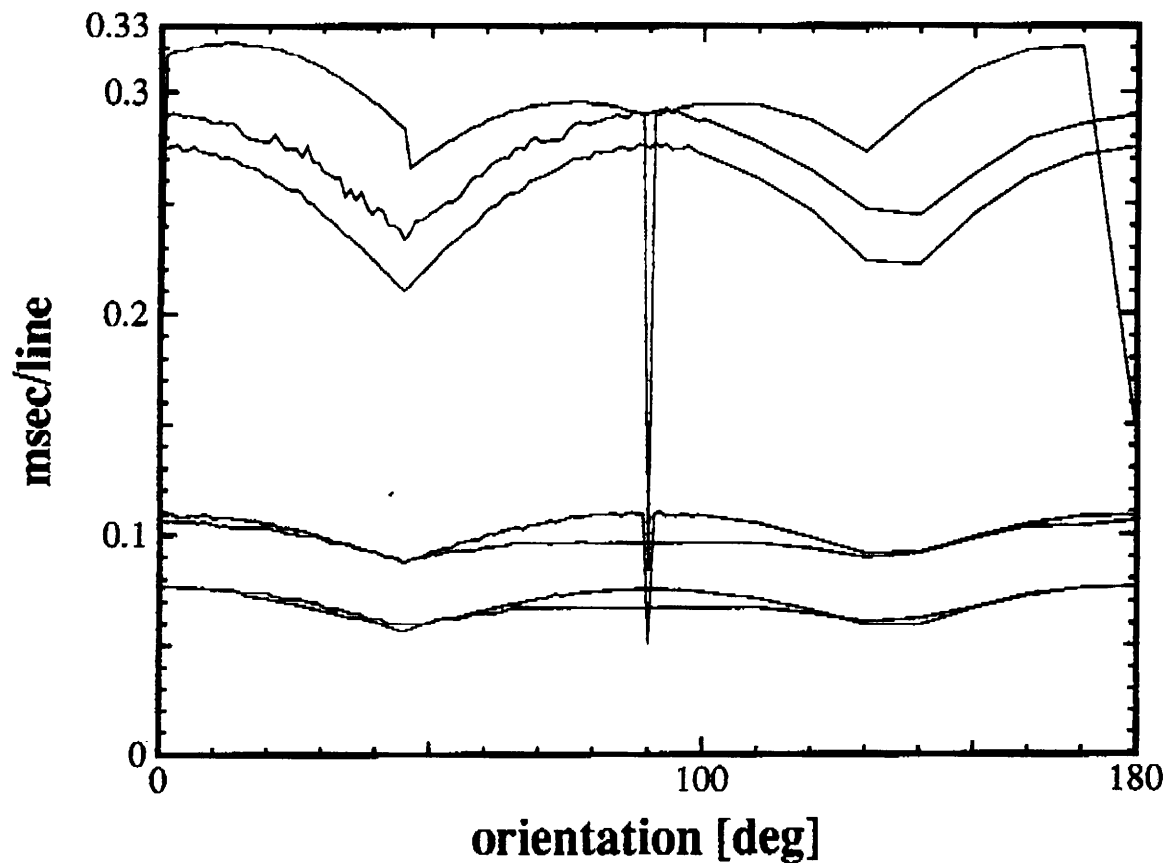


Figure 4: Line Performance vs. Orientation

2.4. Multi-User Performance

Even though the numbers presented above allow a rather complete characterization of the line-drawing performance of an Xserver, they have been measured in a so-called "controlled environment" i.e. without any other interfering users and only the absolutely essential system programs running on the machine. As can be seen from Figure 6 – the timing behaviour of multiple line-tests, each running in its own window on an otherwise unloaded machine – multiple sessions not only increase the time taken for an Xcall but in addition make its timing-behaviour less predictable. This is mainly due to a slight but steadily increasing desynchronization of the individual test-programs caused by the scheduling algorithm: especially in the last third of the tests some programs have already finished while others are still running. Of course this could be easily overcome by introducing synchronisation points after each subtest and discarding the last few runs of such a test. A more realistic approach to multi-user performance, at least in our opinion, is not to get rid of the desynchronization, but to control it by introducing a stochastic delay after each Xcall and comparing the distribution of Xserver-times such measured against the single user case. As soon as the mean of the probability distribution used to generate the delay is large enough, the mean processing time for an Xcall should be the same, but the variance should be still bigger than in the single user

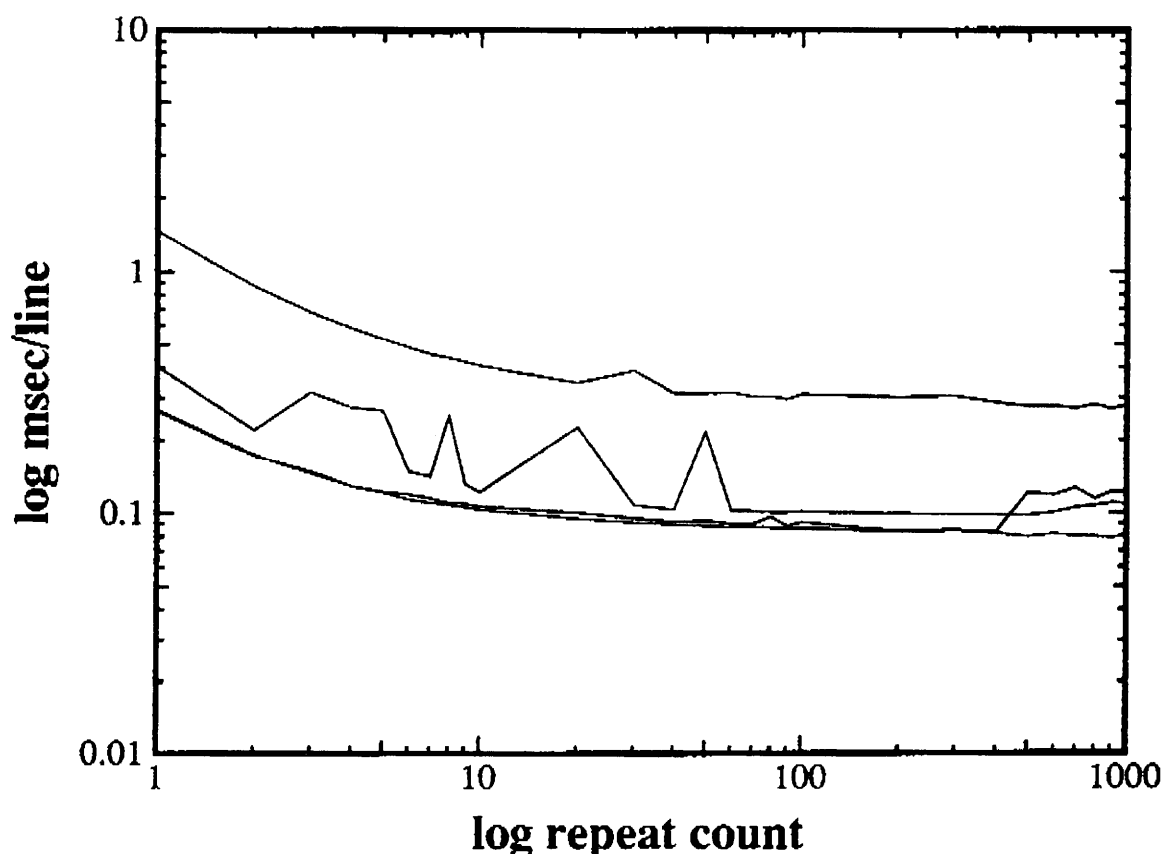


Figure 5: Line Performance vs. Repeat Count

case. Any interactive user or any program which produces such or bigger delays will – on average – not be slowed down by the other processes using the same Xserver. The variance of Xperformance measured at this break-even point can be used as an estimate of the worst-case timing-behaviour of the tested Xcall. The implementation of such a test is under way, but results are not yet available.

3. Getting Few from Many: Data Reduction Strategies

Despite data-reduction techniques like the spline-approximation described in Section 2.1 a potential user of *steppingstones* has to face an even larger amount of data than someone running *x11perf*. As yet, the design goal (3) has clearly not been met, and *steppingstones* has to be completed by an analysis tool reducing the measured data to a manageable set. Depending on the results of still unfinished tests, this will be achieved by principal component analysis and/or cluster analysis of the timing results like suggested by Wang et al [Wan90a].

Multi-user Line Performance (local)

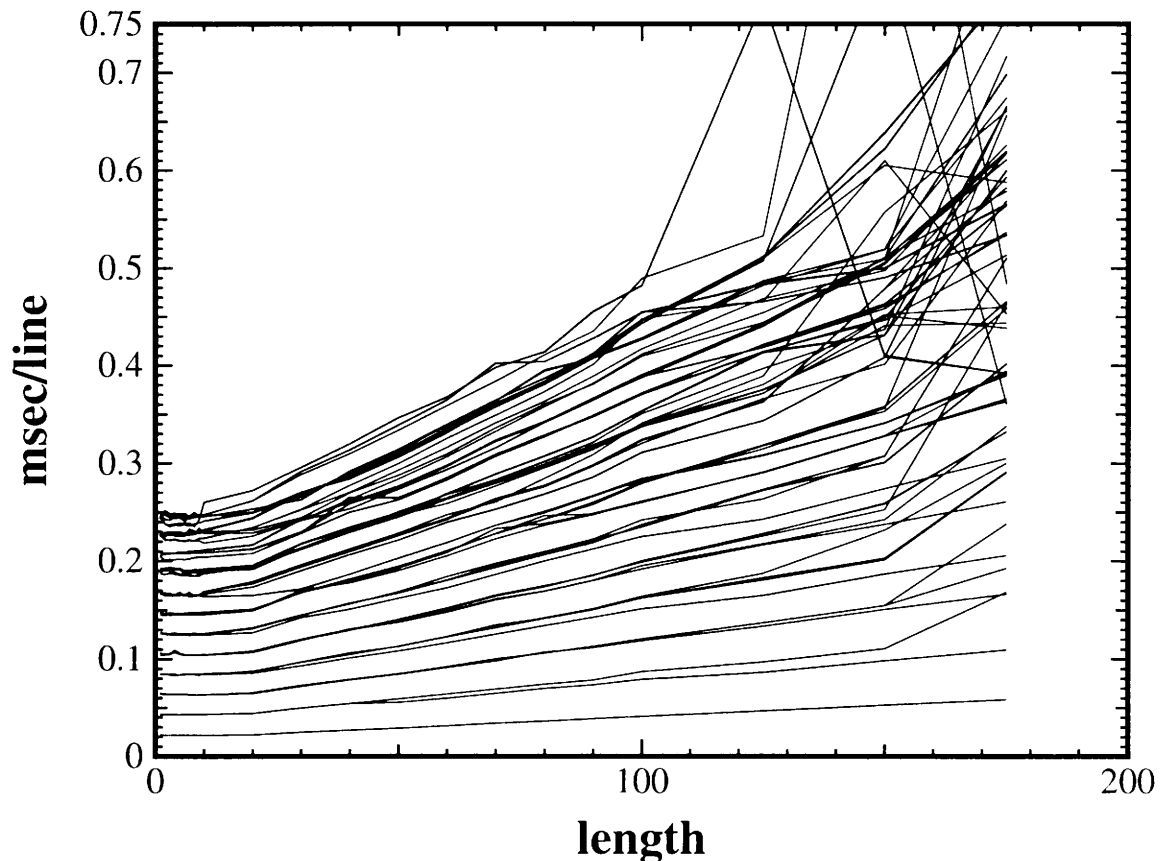


Figure 6: Multi-User Line Performance (local)

Acknowledgements

I would like to thank Harald König from TAT and Franz Pestenhofer from our group for their willingness to discuss even the most obscure ideas on Xperformance. In addition many thanks to all those from the Department of Theoretical Astro-Physics (TAT) at the University of Tübingen, those from the Department of Computer Science at the Polytechnic Institute of Regensburg, and last but not the least, all those from IBM EWD at Austin who made it possible to run some of the tests on their machines.

References

- [Ich76a] K. Ichida et al, "Curve fitting by a piecewise cubic polynomial," *Computing* **16**, pp. 329-338 (1976).
- [Lan86a] Peter Lancaster and Kęstutis Salkauskas, *Curve and Surface Fitting*, Academic Press, London (1986).
- [Wan90a] J. C. M. Wang and et al., "Technique to evaluate Benchmarks:," *The International Journal of Supercomputing Applications* **4**, pp. 40-55 (1990).

Security and Open Working in the Networked Academic Community

Denis Russell

*University of Newcastle upon Tyne
England*

Denis.Russell@Newcastle.ac.uk

Abstract

Even though academics thrive on the publication of their work, there is still a need for security services. This paper summarizes some work that has been done in assessing the perceived threats in the UK, and looks at possible defences against such threats. The emphasis in this paper is upon defence against attack rather than recovery from damage caused by a successful attack.

International standard solutions will not be available, for a long time and may well not properly recognize the multi-protocol nature of international academic networking, and Kerberos seems to be the best available system.

Even Kerberos has its problems and limitations, and this paper discusses some of them, including password guessing and non-Internet protocols such as X.29.

A system such as Kerberos is just a tool to aid in the implementation of a security policy. The role of a security policy is examined.

1. Background

The worldwide Academic Community community is experiencing several changes which are simultaneously causing it to consider security measures:

- The world of computers is subject to increasing instances of "hacking" and "viruses". Some, such as the "Internet Worm" even reach the popular media and anti-hacking legislation is being considered in many countries.
- The rapid standardization on a few operating systems has meant that any security flaws quickly become widely known.
- With the benefit of perfect hindsight it can be seen that many systems suffer from an over-relaxed attitude to security both by the designers, developers, and operators. Amazingly some manufacturers still ship their operating systems configured with ridiculously lax security.

Some specifically networking issues are important:

- International networking, particularly the explosive growth of the Internet, has enabled "hackers" to prowl the computer globe with unprecedented ease.
- There is an increasing desire to make administrative data selectively available across University LANs. Some of this kind of data is by its very nature, and because of privacy legislation, more sensitive than much of our traditional research and teaching work.
- There has always been some "sensitive" data within our systems, such as examination marks. However there are increasing categories of sensitive data. Medical patient-specific data of various sorts is one particular category.
- Much LAN technology, especially Ethernet, is particularly easy to monitor. Simple and generally available programs allow a modest PC to watch traffic between third parties. This allows the "capture" of data, including passwords. In addition to the ease of monitoring, much LAN technology allows LAN stations easily to masquerade using the network and MAC addresses of others.

It is perhaps the last point – the ease of monitoring Ethernet – that has most concentrated the minds of the community.

1.1. The Threat

In response to the issues listed above, the author conducted an investigation of network security for the Joint Network Team of the UK Academic Community [Rus91a]. The risk was assessed in several ways, including personal contacts, mailing lists, and presentations and discussion sessions at the annual UK Networkshop and at the first Joint RARE-EARN Networkshop [Rus90a]. Much of this paper is based on the findings and conclusions of that study.

The community at large seems to perceive three main kinds of threat. The most pressing is the compromising of passwords that are still used as almost the only form of user authentication. This threat, in its various forms, is perceived to be by far the most important at the present time. The second threat is that there are various forms of traffic on the network that contain confidential data. This may be administrative data, sensitive personal data, or even sensitive research results. Often ignored is that one kind of very sensitive data is passwords or encryption keys – not only as they are being used in the authentication phase, but also when they are being changed. The data in transit may be during any kind of network interaction, whether terminal access, FTP, disc backup, or whatever. Opinions as to the importance of this threat vary from not at all, to being vitally important.

However, it should be noted that FTP security is often achieved in practice by encrypting a file before it is transmitted, and decrypting it after it has been received. Typically, the "key management" is the same user supplying the key to the encryption and decryption operations via a remote login. It is not known how often this operation is done over an insecure channel (see last paragraph).

The third kind of threat is manifested as a general unease concerning the increasingly important applications of electronic mail. Mail is open to various abuses, including forging the origin, observing the contents, tampering, denying reception, and other kinds of attack. Some of the threats can be countered by the same kinds of defence that we shall dis-

cuss in the context of the first two threats, but others, such as forging the origin of messages, and denying their reception are more characteristic of mail.

The threat of password compromise is most widely feared and the consequences are too widely known to need reiteration here. Passwords are compromised in two main ways. The most obvious is by the network eavesdropper recording passwords that have been transmitted in the clear. Less obvious, but perhaps more important is password guessing. Though the number of possible passwords is usually so large that password guessing would appear difficult, users have a strong tendency to choose easily remembered passwords. The result is that short names or common words are used with dismaying frequency, and a large fraction of passwords can be guessed by trying the words a modest dictionary. Methods that improve the quality of passwords chosen, and force them to be changed periodically, can greatly improve system security. However the designers of such systems often forget human frailty and users of unmemorable passwords tend to write the passwords down or even attach them to their terminals.

Systems with strong authentication that allow users the freedom to share facilities without sharing computer identities (and thus passwords) improve overall system security. A sharp distinction must be made between doing this in conjunction with strong authentication and systems that dispense with weak authentication and rely on even weaker host authentication – resulting in a hacker's delight.

The eavesdropping threat may be countered by suitable strong authentication methods. Such schemes protect against eavesdropping by the use of cryptographic techniques. However, many such methods are still ultimately based upon passwords that users must remember, and there is still a large advantage in encouraging high quality passwords [Lom89a, Mer90a, Hig90a].

The second threat, that of disclosure of confidential data to eavesdroppers, is clearly important for some kinds of service. It has been suggested that all Ethernet data should be protected from disclosure by encryption. However, this is clearly impractical at present. Apart from the cost of such a crash program, the standards and technology are just not available. In addition the average editor of a FORTRAN program or a report such as this, or the recipient of a voluminous mailing list that is disseminated worldwide is unlikely to welcome the cost of such security services. It is clear that selective confidentiality services are required, but only as and when the need is perceived rather than all the time.

In addition to access for which security is currently inadequate, there is a large category of access, best characterized by anonymous ftp for which no access controls are needed. On the contrary, academic life thrives on the free and open exchange of information. In the electronic world, this kind of free exchange is achieved by several means, including electronic mailing lists and anonymous ftp. It is vital that any security measures do not impede these kinds of exchange.

1.2. The Current Situation

The distinction between "hosts" and "terminals" is now almost totally arbitrary. Not only are micro-computers often used as terminals to access mainframes, but they are also used in peer-to-peer situations, particularly when transferring files or participating in so-called network operating systems. The humblest PC can be directly connected to an

Ethernet, and there is virtually a continuous spectrum of capability from such a machine right up to the mainframe.

At the same time as this, the academic community also possesses a large number of simple (sometimes called "dumb") terminals in use on PAD lines, and there is little sign that such terminals have become a thing of the past.

The academic community contains a huge variety of systems – both operating systems and networks. Any proposal to improve security must take account of this. It is assumed that operating systems will range from those (especially UNIX) in which certain security measures will either be easy to introduce, or even to specify on procurements, to others which are totally fixed and nothing at all can be done inside the system itself. These latter systems include those used for finance and administration, and so are those for which the highest levels of security are desirable or essential. The kinds of network connection that we envisage with such a machine is at worst a "reverse-PAD" connection, or perhaps an X.25 connection. It is likely that the only way of improving the security of such systems is by interposing some kind of active security box between them and the part of the network where there is the biggest threat. However, it is likely that such security boxes could be used only to enhance the security of terminal access, and not of file services.

We assume in this study that minimal, preferably zero, hardware changes are desirable. Apart from the obvious desire to minimize cost, minimal hardware changes would also expedite the implementation of any security measures. For these and other reasons we have not seriously considered such radical solutions as insisting that all access should involve smart-cards and associated equipment. There are many more extreme solutions that were also ignored as being inappropriate to our community, including "scrap Ethernet and UNIX". We also need to remember that any academic community must remain open both nationally and internationally. Indeed this is one of the prime motivations for networking. Security measures must not isolate academics.

2. Standards

The most desirable solution to our problems would be to appeal to international security standards. If these could be specified in procurements, would counter all the threats we perceive, be affordable, allow open access of valid users with minimal disruption, and be available in short order, then this would obviously be the way to go.

Unfortunately, the likely timescale for security standards that satisfy the above criteria is very long. It seems inevitable that if security measures are to be taken for real users in the near future, then they cannot be international standards. (For a fuller discussion of this matter, the reader is referred to the author's earlier paper [Rus90a].)

In addition, the international academic networking scene is undergoing a rapid revolution. For some time there has been a division with North America using the TCP/IP suite of protocols, and the rest of the world, particularly Europe, using X.25-based protocol stacks. Local campus Ethernets in Europe have been multi-protocol – with TCP/IP and X.25 (and proprietary protocols) used side-by-side, but the wide area has been exclusively X.25. In parallel with this so-called "protocol wars" have raged. However, we are now seeing a very rapid adoption of TCP/IP for wide-area communication across Europe, and indeed across

the Worldwide academic community. It is a fact that we now live in a multi-protocol world. This fact must be recognized by any security measures. Just as the needs of users have swept away the artificial barriers erected by pedantic adherence to incompatible standards, so they will also force security measures to work across heterogeneous systems, networks, and protocol stacks.

3. Kerberos

The Kerberos system from MIT in the USA was produced as part of project Athena, and is freely available within the USA. There are two versions of Kerberos that are relevant to our discussion, versions 4 and 5. Version 4 is widely deployed in the USA. It provides strong authentication in an environment of networked UNIX systems. Kerberos is used to secure UNIX high-level networking protocols.

Kerberos uses a trusted key server, and uses a version of the Needham and Schroeder protocol [Nee78a], sitting on the Internet suite of protocols. The protocol is weakly dependent on IP in that the IP address is included in certain protocol exchanges in order to make the protocol more resistant to certain forms of attack. The encryption method used is DES. There are optional confidentiality and integrity services defined.

Version 5 is an updated version of the protocol. There are many detailed changes, but perhaps the most important are that the encryption method can be chosen from amongst many; the underlying network may be of any suitable kind, including ISO; inter-organizational authentication mechanisms are more extensively developed; and several of the limitations of version 4 have been eliminated.

As was explained at the last EurOpen meeting [Koh91a], a completely new beta-test version of Kerberos is now available (from mid June 1991) from MIT that supports both version 4 and 5. This is by far the most promising basis for future security work. However, the problems related to export of cryptographic software from the USA again impede progress in this area.

3.1. Limitations of Kerberos

Bellovin and Merrit [Mer90a] have pointed to several limitations of the Kerberos protocols. Many of these have been eliminated in version 5, but some remain. Perhaps the most practically important of these is the possibility of password guessing. It would seem that this weakness could be reduced by the use of techniques that encourage the choice of good passwords, together with user education concerning the ways of memorizing good passwords – see Section 8.

It is not clear just how weak the remaining holes make Kerberos. Some of the weaknesses are not easy to exploit. However, even with its acknowledged problems, Kerberos is much stronger than most other existing methods. To quote Bellovin and Merrit themselves:

“We wish to stress that we are not suggesting that Kerberos is useless. Quite the contrary – an attacker capable of carrying out any of the attacks listed here could penetrate a typical network of UNIX systems far more easily. Adding Kerberos to a network will, under virtually all circumstances, significantly increase its security; our

criticisms focus on the extent to which security is improved."

3.2. Application Programming Interface to Security Services

Recently two "Application Programming Interfaces" to (generic) security services have been defined. One has been defined by the OSF as part of its adoption of Kerberos, and the other is being developed independently by John Lynn at Digital. The idea is to isolate any users of the interface from the details of the underlying protocol. The MIT release of version 5 includes an implementation of the Lynn interface.

4. The Sphinx System

Kerberos is based on private-key or symmetric encryption technology. One of the most interesting developments in cryptology has been the invention of public-key or asymmetric technology ([Dif76a, Dif88a, Hem89a]). In April 1991, DEC made available an implementation of an authentication service using asymmetric encryption – the Sphinx or SPX system. This is similar in function, but exploits the unique characteristics of asymmetric encryption.

While this is an exciting development that should be watched carefully, it is not something that should be exploited for practical user security at the present moment because

- The system is an early beta-test version. It has not yet been open to public scrutiny for long enough and flaws may yet be discovered (this is one of the points of it being released).
- The system is subject to US export restrictions that are even more severe than those for Kerberos.
- Public Key technology rests on a single algorithm – RSA. Should this be broken, there is no alternative asymmetric algorithm.
- The RSA algorithm is subject to US patents from RSA Inc. While these do not apply outside the USA, their existence may well impede the export of such technology from the USA just as they are impeding the availability of public or shareware implementations within the USA. (N.B. No comment is being made here on the rights or wrongs of such patents.)

5. The Security System of Choice

From the survey of the situation in the UK, it is clear that authentication services plus at least occasional confidentiality services are urgently required by the UK Academic community. It seems unlikely that the situation in other European countries is significantly different. In the absence of standard solutions, Kerberos seems by far the best choice to fulfill this need. Much of the rest of this paper is concerned with what needs to be done to fill the gap between what Kerberos provides, and what the community needs.

There do remain problems in adopting Kerberos. Apart from the export problem and protocol weaknesses mentioned above, there are services in the community that may be difficult to cover with Kerberos. In particular, these are dumb terminals on PADs, non-UNIX services in general, and "turn-key" hosts in particular – especially financial and administrative systems. There is the question of retaining access from

"non-Kerberized" sources without compromising security. There are questions about the scalability of a Kerberos system should it be applied nationwide to the UK community, or even internationally.

In addition, there are various questions concerning the interaction of the interim mechanisms suggested in this section with current interim and long-term services. For example, if cryptographic tickets are to replace passwords in file protocols, then limitations of "password" length or content may become important. There are obvious implications concerning the different semantics of cryptographic tickets versus passwords. Some of these questions would be answered by the provision of a suitable library of calls within each system that allowed each service to be "ticketized" or "kerberized". The latest version of Kerberos goes some way towards providing that. However, there remain some significant areas where work needs to be done.

Again, with interactive access, such interim solutions involving the exchange of a PAC in place of a password have protocol implications. Neither X.29 nor VTP explicitly contain the notion of a password. In the case of X.29 it would be possible to handle the communication of a ticket by a negotiation mechanism at the start of the session (that is, by the representation of the ticket exchange protocol in terms of user-level messages that are unusual enough not to be confused with normal terminal traffic). We outline one such scheme later in this paper. However, it is not clear that the spirit of ISO VTP would allow such an interim solution, and thus it may well be that ISO VTP can only be secured by the use of the secure CASE when it eventually becomes available.

It is the task of the second stage of the study to pull these various strands together, resolve many of the questions, and produce a practical plan for standardization, development and implementation.

6. The Need For a Security Policy

Without a security policy, security devices are useless. It is no good purchasing a pile of padlocks to secure a house unless there is a suitable policy about fitting them to doors and windows, of locking them at appropriate times, and of allocating keys to keyholders appropriately. General guidelines as espoused by police crime prevention officers are very useful in improving even such an apparently simple thing as home security.

Similarly with computer security, a body of general advice and guidance needs to be developed. For example, how to make data available for anonymous ftp without compromising other data; how to secure a set of computers using Kerberos; how to configure systems (especially UNIX) and networks for secure operation; how to protect confidential data; how to evaluate and counter the threat posed by dial-in telephone lines and incoming X.29 and Telnet; how to develop a security policy that is suitable for a particular University and then to get everyone to implement the policy, etc.

At least as important as any technical decisions are administrative decisions concerning acceptable use policies within an institution and the consequences for people who do not comply with such policy. For example, at one university some years ago, the official position was that no disciplinary action could be taken against students who were compromising the service on the mainframe computer. As a result, the maintenance of the system security turned into an open contest between

hacking students and hard-pressed system staff. This both wasted staff time and significantly degraded the service for other users. In contrast, another site that uses technically advanced security measures also has a very firm and well-known policy on computer misuse. The administrators there know of no instance where the system security has been compromised. They attribute this at least as much to the firm policy as to the technical measures.

The author's University is starting to develop a security policy that will encompass both technical and administrative measures, and will involve university administration at a very high level.

In addition, various national data protection legislation must be taken into account.

Recently, the Security and User Services Areas of the Internet Engineering Task Force have produced a draft "Security Policy Handbook". This should soon appear as an Internet RFC. This is an extremely valuable document, and should be required reading for anyone developing a security policy.

7. Reaching Parts Kerberos Doesn't Yet Reach

The author is strongly of the opinion that Kerberos should be used on a wide scale to improve the network security both within campuses and between campuses. Many forms of network access have already been "kerberized". However, to fully achieve improved security, some significant extensions of the current Kerberos need to be made to cover some common forms of network use.

7.1. Kerberizing X.29 Terminal Access

X.29 terminal access is one of the most ubiquitous forms of simple terminal access. One of the biggest problems with securing X.29 in general is the remarkable number of ways in which the terminal end is realized [Rus91a]. The original model is of a very simple ASCII terminal attached via an RS232 serial line to a PAD. This is probably the exception now rather than the rule. The "terminal" is very likely to be a terminal emulation program running in a micro-computer. Indeed, often the PAD function is also located in the same PC. Perhaps the ultimate example of this is the Rainbow package from the University of Edinburgh that runs on a PC with an Ethernet interface and implements various stacks of protocols including an X.29 "terminal" on top of an X.25 module that operates directly across an Ethernet. The same package also includes a nice implementation of the UK NIFTIP file transfer protocol.

In [Rus91a] the present author suggests various ways in which the Rainbow services can be "kerberized". In general terms, these involve the Rainbow micro computer exploiting the fact that it can be a full-blown participant in network interactions. Thus it can make initial calls to a Kerberos authentication server to obtain tickets, and then use them in authentication exchanges with a host (which might be another PC). The initial ticket request will be over an X.25 call rather than using Internet protocols. This would be a problem with version 4 because of its weak dependence on IP. However, version 5 solves this problem with its more general protocol formats.

Rather more interesting is the way in which the Kerberos protocols might coexist with X.29 and the like. The Kerberos protocol involves a

set of protocol messages or data units (PDUs) that must be exchanged between the client and the server. With many existing protocols they are exchanged as the initial messages on a connection. However, with X.29, the possible format of the messages depends on just where the messages are inserted into the stream and removed from it. X.29 connections are notorious for not being fully transparent, and this is especially the case if any security features need to sit on top of an X.29 stream rather than being able to sit between the X.29 and X.25 layers. If the protocol could sit between the two layers, then transparency could be assured, but if it has to sit on top of X.29, then not only must many characters be avoided (use only seven bits and avoid all control characters), but there is no record structure available. Unfortunately, the boundary between the two layers is only easily accessible on some machines, and it would not be wise to require such access if it can be avoided.

Fortunately, these problems are well known, and many encoding schemes (e.g. Kermit or BinHex) that allow the passage of arbitrary data as is required in cryptographic authentication.

While it is easy for a fully functional Rainbow micro to interact directly with a network authentication server, this option is not possible for many X.29 configurations. Perhaps the micro is emulating just the terminal, and the PAD function is performed by a physically separate piece of equipment. Fortunately, there is an indirect way of contacting the authentication server. Since we are defining a way of transparently passing authentication messages on top of the X.29 connection to the host, there is no reason why the initial dialogue between the client to the authentication server should not also be encoded in a similar way and passed over the X.29 connection to the host. The host can then re-code the messages, and interact suitably with the authentication server on behalf of the client. The assumption here is that any host that is being accessed via X.29 will have sufficient communications capability to be able to contact the authentication server.

At first sight, this appears to be dispensing with one of the advantages of Kerberos – the possibility of authenticating the host to the user as well as the user to the host. This is because the host appears to be involved as a trusted party in this initial exchange with the authentication server. Even if this were the case, it would be no worse than the present situation in which the host is the only trusted party. In fact the host is not really being trusted at all in such a situation. The Kerberos protocol protects against the majority of network attacks, including the observation, forgery, and replay of messages. If the host is asked to forward the initial messages between the client and the authentication server, it is just acting as a kind of network device that gateways between the representation on top of the X.29 connection, and the “native” network representation. The messages themselves are cryptographically protected, including from the “friendly” host.

Given this mechanism seems to be necessary for the less flexible implementations of X.29, then we see it can also be used equally well for the fully functional Rainbow case without exploiting that extra functionality. The advantage is that the one host implementation will work with a variety of terminal configurations (providing of course that all of the terminal configurations implement the same protocol).

At the terminal end there are two possibilities for inserting such security facilities into the micro. One is to take the source of a terminal emulator and extend it to provide such facilities. A practical problem with this is that the source of the emulator must be available for such a

change to be made. Another is that only those emulators that have been modified will be secure. In the way of things, this is certain to exclude the one that is most desired for some other reason.

One way of avoiding these problems is to arrange that the security facilities are implemented in a program that intercepts the stream of data between the terminal and the (virtual) PAD. In this way, any favoured terminal emulator may be used unchanged. The virtue of this approach is that it is perhaps the most generally applicable form of security.

7.2. Dumb Terminal X.29

At the extreme end of the X.29 spectrum is the dumb physical terminal connected by a serial line to a PAD. In this case, there is no computing power available between the user and the network to use for cryptography. However, for secure authentication there is a possibility – the “hand held authenticator”.

The hand-held authenticator is traditionally something like a pocket-calculator. The mode of use is that instead of prompting for a password, the host presents a kind of challenge; the user reads the challenge off his or her terminal and enters it into the keyboard of the hand held authenticator; the response is read off the display of the hand held authenticator, typed on the terminal and sent to the host. The relationship of the challenge and the response enables the host to decide whether the user should be allowed onto the system.

To use this in conjunction with Kerberos, the host would have to obtain something corresponding to the encrypted ticket from Kerberos and send it to the user as the challenge. The hand-held authenticator would have to perform an operation on the “challenge” that corresponds to deriving the Kerberos authenticator from the ticket, and this is the response sent back to the host. Such an arrangement was discussed on the Kerberos mailing list early in 1990 and is on the list of possible future extensions to Kerberos [Koh91a].

This kind of authentication for “dumb terminals” could be implemented in a number of ways. It might be possible to get actual hand-held devices with suitable software to perform this function, or alternatively, the function could be programmed and made available for a range of portable or pocket devices. ISO Virtual Terminal

The extent to which Kerberos security services can be exploited for ISO Virtual Terminal (VT) is not at all clear. There are at least two problems.

The most serious problem is that VT contains an elaborate and tightly defined virtual terminal model. At least one VT expert is of the opinion that this rules out the pragmatic encoding of the Kerberos messages in the data stream that we suggest above for X.29. If this really is the case, then most of the kinds of solutions that we discussed in the section on X.29 will not be possible. However, it may well be that further study will revise this conclusion.

VT uses the presentation layer, and thus similar comments apply as with ISO FTAM (see below) – i.e. that it might be possible to protect the user data, but that protecting the structure of the data would need much more work. However, it may be that the structure of the data with VT is so fixed that its disclosure would not be an important breach of security.

If all else fails, then at least the hand-held authenticator together with a challenge/response dialogue would allow for secure authentication.

7.3. Kerberizing File Transfers

For protocols such as FTP, the password field is the obvious vehicle for security PDUs. The sequence would then be for the client to obtain a Kerberos ticket for the FTP server, derive the authenticator, and transmit it in the password field. The FTP server would then check the "password" by calling the Kerberos library. To achieve such simple one-way authentication, the process of Kerberising involves making suitable modifications in the initiating end to obtain the authenticator, the transmission of the authenticator in place of the password, and the calling of the Kerberos library in place of the previous password checking logic.

Of course, the password field must be suitable for such use – the authenticator is a general bit-pattern (because of the use of encryption), and is substantially longer than is usual for passwords. If mutual authentication is required, then the server must return the extra message to the client. If it is possible to send a password back from the server to the client to the server, then this can be used to convey the extra message. It is not clear whether file transfer protocols allow such mutual exchange of authentication data, and less clear whether implementations could, in practice, be modified to do such an exchange even if the protocol permits it. However, the protocol definition of FTAM has recently been changed to allow the password to carry cryptographic information precisely so that it can convey either a Kerberos authenticator or similar security information.

If the file transfer is to use the integrity or confidentiality services, then the communication must be by means of the appropriately constructed messages. For FTAM the choice is not particularly obvious because of the use of the presentation layer. If the contents of the file need protection, but the structure of the file does not need such protection, then the File Access Data Units could be protected by inclusion within the Kerberos integrity or confidentiality messages, and these in turn transmitted by the protocol. However, if the information on structure that is represented by the presentation layer needs protection, then the presentation layer in turn needs protecting by lower layer security services. This may require a more extensive development unless the Kerberos privacy and integrity message formats could be used by the presentation layer as its transfer syntax.

It has been pointed out that if the integrity of the PDUs that define the operations on a file cannot be guaranteed, then an attacker could, for example, turn a "read" operation into a "delete" operation. Thus, the security requirements of the control parts of a FTAM session are more critical than the data-transfer parts.

The standards community is working towards a "Security CASE" for general use by application protocols. While this is a desirable target, it is very much a long term process.

In summary, given a "Kerberos Infrastructure" making one-way authentication secure for FTAM seems straightforward. Mutual authentication depends on the availability of a reverse path from server to client for the appropriate information. For either integrity or confidentiality services to be used the communication must use the appropriate Kerberos messages. In the case of FTAM this is compli-



cated if the structure needs to be protected as well as the content of the file.

7.4. General Discussion

In this section we have discussed various ways in which, given a basic Kerberos infrastructure, the Kerberos security services – authentication, integrity, and confidentiality – could be more widely deployed.

For FTAM the provision of secure authentication would be straightforward given that the password field could be used to transmit the appropriate Kerberos messages. Mutual authentication requires the transmission of an extra Kerberos message. Whether this is possible within the protocols and then within the implementations needs to be answered by protocol experts, and by implementers.

The exploitation of the integrity and confidentiality services requires that the transfer itself use the appropriately formatted Kerberos messages.

Given the availability of the Kerberos Infrastructure, then converting FTP to use Kerberos authentication would be a straightforward matter of calling a small number of subroutines to generate and process the authentication messages and of acting on the positive or negative results of those calls. Using the integrity or confidentiality services would involve calling two further routines to encode data within the Kerberos protected format before transmission and extracting data from this format after reception.

For X.29 the process of adopting Kerberos is much more complex, mainly because of the many different ways in which the X.29 protocol is used in practice. However, the main work is in deciding exactly what way the Kerberos protocol is to be combined with the use of X.29. Once this has been decided, the process of implementing authentication is a matter of arranging for the authentication messages to be passed to the Kerberos library and the results of those calls to be acted upon. As with FTP, the use of integrity and confidentiality involves arranging for all I/O to be coded into the appropriate message before transmission and decoded from it after reception.

Note that with all of the proposed applications of Kerberos, the service provided is secure authentication of existing registered users of the system. Thus, there is no external “Access Control List” provided by such a system. This is particularly relevant with respect to file transfer, where it would be desirable to permit access to a file to a specific external user without needing first to register that user as a user of the host operating system. Kerberos does not provide such a mechanism, and users unknown to the host of a filestore would presumably have to be mapped onto “anonymous” or *somesuch*.

8. Password Guessing

The Kerberos system is subject to password guessing. If the attacker can eavesdrop, then certain initial message exchanges can be recorded and then (since, following Kerckhoffs, we assume the protocol is well known) offline trials against a dictionary can be made at leisure. Given sufficient eavesdropping, this would yield the equivalent of obtaining much of the UNIX */etc/passwd* file together with the well-known consequences. (However, note that in the more usual plaintext password situation, such an eavesdropping attack would already have

obtained the plaintext passwords with no further effort required.) If the attacker cannot eavesdrop, she can still obtain an initial ticket merely by asking for it. However, this method might call attention to an attack if used extensively.

Password guessing works because users tend to choose simple short passwords and are very unoriginal. Many systems make this situation worse by only accepting very short passwords, or even by shortening passwords.

One response to this is not to let users choose their own passwords, but to allocate them, perhaps generating them pseudo-randomly. Such passwords are difficult to guess, but they are also difficult to remember, and users tend to write them down and even to attach them to the terminal with sticky paper thus making the security even worse.

There are several fairly well-known ways of getting users to choose good passwords which are still easy to remember [Hig90a]. One way is to get the user to choose a phrase rather than a word, and then take the first letter from each word of the phrase. There are many variants on this theme. Perhaps more effective is to allow passwords of unlimited length and insist on fairly long passwords, or, preferably phrases. Long strings may be folded into fixed length keys (of, say, the 64 (or 56) bits required for the likes of a DES key) by folding and exclusive-ORing the segments of a long key. Since a typical phrase typically contains only a few bits of entropy per word, a substantial phrase (preferably not from Shakespeare!) should provide good security against password guessing and yet be easy both to remember and to type.

If users are educated to choose good passwords in this kind of way, and this education is reinforced by a suitable interactive password changing system that filters out easily guessed passwords and institutes a sensible password ageing scheme, then password security can be made very good and most password guessing attacks would be thwarted. Having passwords administered centrally by a system like Kerberos should make the implementation of such a scheme of choosing good passwords that much easier.

We believe that a system to improve the quality of user passwords in this kind of way is essential, independently of whether Kerberos or some similar system is adopted.

9. Implications for the Transition to ISO

One of the aims of the ISO transition was that complete ISO protocol stacks could be purchased from commercial companies and used, unmodified. If we start to require security services before they are commercially available, or services that are different from those that are commercially available, then there is a real conflict with this aim. On the other hand, the projected timescale for ISO standard security measures is clearly too long for the requirements of our users. There is obviously a dilemma here.

We believe that this dilemma is best resolved by taking interim measures, and trying to ensure, by tracking and participating in the standards process, that these measures converge with emerging standards. In addition the use of a standard API should ease such transition problems.

The alternative of no security measures is just not acceptable for most users.

10. Some Unanswered Questions

The problem with these solutions is not whether they are likely to work, but rather whether they will be useful enough to justify the effort needed to develop them. It is not at all clear just how important "dumb terminal" access is for confidential data. Would it be acceptable if dumb terminal access was authenticated (using a "hand-held authenticator"), but could not be made confidential? The input to this study indicates that this would be a very welcome move for many applications.

Again, would it be sufficient to produce just a hand-held authentication program to run in a range of common micros, or would actual hand-held devices be required? How important are dumb terminals anyway? The range of possible requirements is vast and the whole process could be delayed indefinitely while the possibilities were argued at length.

The answers to these detailed questions were not identified by the study. At all stages, the most diverse opinions have been expressed on this point, ranging from the belief in the imminent demise of X.29 in favour of X, to the extreme importance of X.29 access using dumb terminals and standard PADs. Indeed, it seems unlikely that this matter can be resolved until at least some experience has been gained in the practical deployment of a security service like Kerberos. The most common reaction at present is a blank and uncomprehending demand to secure everything against anything. With suitable experience in developing security policies and implementing them using a core set of security services on certain protocol stacks, the real need for further services could be identified.

In order to make progress, the author is experimenting with a Kerberos client that is combined with a terminal emulator as described in the section of X.29. It is hoped that experience with this, together with a host implementation for UNIX, will make it clearer whether such a security mechanism will be useful. At the same time, the deployment of Kerberos to protect other types of network service is being pursued.

Perhaps most importantly, this is being done in close conjunction with the development of a University security policy, and together with other methods of securing host systems.

11. Acknowledgements

Some of the work on which this paper is based was supported by the UK Joint Network Team under grant number J018/500/01.

References

- [Dif76a] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions in Information Theory* **6**, pp. 644-654 (Nov 1976).
- [Dif88a] W. Diffie, "The first Ten Years of Public-Key Cryptography," *Proc IEEE* **76**(5) (May 1988).
- [Hem89a] V. Hempel, *Final Report - Protection of Logistics Unclassified/Sensitive Systems (PLUS)*, Office of the Secretary of Defense Production of Logistics, Systems Department of Defense (1989).

- [Hig90a] H. J. Highland, "Random Bits and Bytes – Using Good Passwords," *Computers and Security* 9 (1990).
- [Koh91a] J. T. Kohl, "The Evolution of the Kerberos Authentication Service," in *Proceedings of the Spring 1991 EurOpen Conference, Tromsø, Norway* (Spring 1991).
- [Lom89a] T. M. A. Lomas, L. Gong, J. H. Saltzer, and R. M. Needham, "Reducing Risks from Poorly Chosen Keys," in *Proc 12th ACM symposium on Operating System Principles* (Dec 1989).
- [Mer90a] M. Merritt and S. Bellovin, "Limitations of the Kerberos Authentication System," *Computer Communications Review* 20(5), pp. 119-132 (Oct 1990).
- [Nee78a] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Comm ACM* 21(12) (Dec 1978).
- [Rus90a] D. M. Russell, "High-level Security Architectures and the Kerberos System," *Computer Networks and ISDN Systems* 19(3-5), pp. 201-214 (Nov 1990).
- [Rus91a] D. M. Russell, *A Study of Security for Networked Interactive and File Services – First report: Current Situation – Second Report: A plan of action*, available from the author (Feb 1991). Prepared for the UK Joint Network Team.

phLOGIN, Why, What and How

Alain Williams

Parliament Hill Computers Ltd, UK

addw@phcomp.co.uk

Abstract

The paper will discuss the history, aims and design philosophy for the program. An overview of the major program features and how they are to be used will be given.

Porting issues on a utility like this are unusual and will also be covered. I end with an indication of where further work is needed.

This paper was first delivered at the NLUUG conference of 8 May 1991.

1. What is it all about?

phLOGIN is a set of programs which replace the UNIX *login*, *getty*, *rlogind* and *telnetd* programs. phLOGIN is a product that has grown out of a special purpose utility to meet the specific needs of a customer to a tool of general applicability.

2. Main Aims

phLOGIN is designed to make great improvements over standard *login* in three areas:

Security

phLOGIN is the first point of contact that an intruder will have with a system, it is important that life is made difficult for him without hampering legitimate users.

It is also important that the System Administrator knows what is going on. Most versions of *login* only record successful logins, the failures are just as important. The ability to take immediate action on a failure is important.

phLOGIN addresses both of these issues.

End User Functionality

Standard *login* is very much a 1960's program, it makes no use of the abilities of "smart" terminals – not even the now common place cursor positioning sequences. It detracts from a nicely designed menu driven system that may be started from the current *login* mechanism – where it is not even obvious how to correct a mis-typed character.

phLOGIN lets the System Manager design a forms type login screen for his users. This may have the date, time, and Message Of The Day displayed, the arrow keys may be used and the function keys bound to actions such as help and information services.

System Administrator Control

There are a variety of ways in which the System Administrator's life can be made easier, these range from stopping logins during backups (and letting the user know why) to the ability to dedicate a terminal for a specific use.

phLOGIN has been designed to integrate with extra security devices that may exist on a particular system, e.g. it knows about `/etc/shadow` or various TCBs (Trusted Computing Bases) where they exist.

phLOGIN only deals with the login step, security at other times needs to be examined independently.

phLOGIN controls may be chosen according to the origin of the login attempt. In the case of a direct connected terminal this is the physical tty, and in a network login (rlogin/telnet) this is the remote machine name/address and possibly the remote username.

phLOGIN is simple to set up and get going, if something complicated is wanted a bit more work may be needed.

3. What does it do?

The features of phLOGIN will now be examined in more detail.[†]

3.1. Origin Specific Controls

This is a key concept with phLOGIN. The point is that the system administrator needs to be able to distinguish between different terminals: be they direct connected or networked.

The direct connected case is easy, one phLOGIN is run per tty from `/etc/inittab` and so different options are given:

```
tty9:1:respawn:/etc/phlogin -f /etc/phlogin.d/cf4 -T vt100 tty9
```

With the networked case a more subtle approach is needed. The options list is built up dynamically, a configuration file is scanned and controls taken or ignored depending on the conditions in selectors. The syntax will be familiar to AWK users with selectors in `//`. e.g.

```
# Some users on remote machines use strange keyboards
/ r:mordor u:gollum / {
    -S erase=~@
    -e TZ=VDT10VWT
}
```

The assertions are:

- `l:name`

Assert that the current localhost (machine being called and on which phrloginp is running) is *name*.

The judicious use of this allows the same control file to be used for several machines.

[†] Only an outline of features is given.

- **r:name**
Assert that the remote (calling entity) is *name*. *Name* may take the form *portnumbername*: or just *name*.
- **p:proto**
Asserts that the connection protocol is **proto**. This may be used, for instance, to distinguish between *telnet* and *rlogin* connections.
- **u:name**
Assert that the remote (calling) username is *name*.

In all of the above a list may be created by separating the *names* by commas. The names given in such a list are taken as alternatives, if any one matches the assertion succeeds.

The control file is compiled so that petty syntax errors do not make set up difficult. The compiled file is a machine independent binary; e.g. you may compile it on a MC68000 and use it on an i486, this means that it can be shared by a network of heterogeneous machines.

3.2. Security

Security and ease of use do not, generally, go together. To make a system more difficult to break into someone has to do some more work and the user interface may need to be changed.

In the first instance the system administrator is going to have to spend some time thinking about how the system is going to be used, and setting the phLOGIN options to enforce this.

There are several methods by which phLOGIN helps to tighten a system up:

- **Login Restrictions – 1**
Terminals are very often used by one individual or for one purpose, use is only expected at certain times. This is especially true in many commercial systems.

phLOGIN allows this to be enforced, thus the options:

```
-A g:accounts,t:09..1730,w:1..5
-A u:backup,t:13..14
```

will only Allow logins by members of the *accounts* group between 9am and 5:30pm, Monday to Friday. The user *backup* is also allowed to login at lunchtime, any day of the week.

This simple facility is remarkably good at enforcing a “who does what and where” policy, and forbidding (for example) *root* logins anywhere except at the terminal in the locked room.

If you don’t trust a remote machine you can lock it out completely:

```
/ r:rogue / { -A =0 }
```

If you **must** permit a *root* login over a network, then at least restrict it to a secure terminal. If you know the port number that the trusted terminal is connected to, you can use:

```
# Forbid all root logins
// { -A !u:root }

# Allow root logins on port 192 of network called web
/ r:web:192 / { -A u:root,w:0..6 }
```


- Login Restrictions – 2

The above feature is a good first attempt, but it is not powerful enough to satisfy everyone. phLOGIN can optionally run a program/shell-script before login is completed. This can do further checks on who is attempting to login.

Since this is supplied by the System Administrator it can check anything that he wishes, e.g. run the diary program to see if the user is on holiday.

- Backoff/Timeout

One way of getting into a system is to try lots of username/password combinations. This is made easier as usernames are often known, and people are very bad at choosing good passwords.

It is a simple matter to connect another computer to a login line (possibly via a network or modem) and sit back and wait.

phLOGIN provides protection against this. After an unsuccessful login attempt phLOGIN will pause for a few seconds, throw away pending input and allow another try. As more failures are made the pause gets progressively longer. It can also be arranged that the line/connection will be dropped after a specified time or number of failures.

This makes brute force methods very slow and also increases the likelihood of detection.

- Username Equivalences

Networking is well known to cause security problems, some of which can be avoided quite simply. When performing an *rlogin* there is the notion of equivalent machines (where a user on one may login to the same name on the other without quoting a password), and a similar idea in users' *.rhosts* files where password free logins are granted as the user sees fit.

The first of these is under the System Manager's control, but is quite crude in application (all users or none at all are equivalent). The second is under users' control, do *you* trust your users?

With phLOGIN in the networked environment the System Administrator can provide machine specific equivalence lists. The users' *.rhosts* files may be ignored or acted on as desired. The allows (-A above) let you specify equivalenced user names, and a password can be insisted on regardless of equivalences (Figure 1).

The net effect of this is to put equivalence control back into the System Administrator's hands. This is important, it was one of the main areas of attack of the internet worm.

- Audit Trails

Standard login only records successful logins. phLOGIN also records failures, this includes the reason why the login was refused.

It is very important that failures are recorded, unless the System Administrator can see that an attempt is being made to break into his system, he can do little to act against it.

The log file is a binary file, this makes it difficult to illegally remove entries.

```
# Ignore standard equivalences from the machine hacker
/ r:hacker / { -q =0 }

# Give equiv for some friends to the fileserver
/ r:friend l:fileserver / {
    # Take note of users .rhosts files.
    -q =1

    # Add some of our own
    -q bill,ben
    -q richard -> dick

    # Only let people login as their equiv users
    -A u:$e

    # But they always have to quote a password
    # regardless of equivalence.
    -p yes
}

# Allow a specific remote users
/ r:kremvax u:gorby,raisa / { -A u:$u }
```

Figure 1:

The audit trail is used on successful login by phLOGIN to tell the user when the last successful and unsuccessful logins were made on the username.

- Elimination of .profile

phLOGIN can be asked to do much of what is usually done in a .profile, e.g. setting environment variables, umask, terminal mode, change directory, ..., and finally running an application program. This means that a .profile is no longer needed, and that the login shell need not be /bin/sh.

This is good news as it is possible (with determination) to break out of a .profile and get a \$ prompt. Also as it is a file in a user writable directory, it can be easily changed. By running an application as the login shell, the user becomes logged out as soon as it is terminated.

- Immediate Action

There are several things which may be wanted to be done immediately a login event happens, these range from printing a line on a hard copy terminal to disabling a line on which there have been too many failures.

These can be arranged. For flexibility the mechanism is to make the log file a fifo.

In addition some systems have a database in which information such as the number of consecutive failures is recorded, phLOGIN knows about these and will maintain as appropriate. Administrative requests to lock a user or terminal out in certain conditions will be honoured.

- Login without Passwords

phLOGIN can be configured so that a user is logged in immediately – as soon as the system goes multi user or the previous login is ended. Optionally the user may be required to press the **return** key.

This is not the security loop hole that it seems. If it is also specified that the login program is, say, an accounts package the

system is quite safe. Most accounts packages insist on a password being entered before anything can be done.

This setup is likely where a terminal has a dedicated use, this is quite common in turn-key systems. It eliminates the need to "login twice", once in to UNIX and once in to the accounts system. It thus helps to hide more of UNIX from the end user – this must be a good thing.

3.3. End User Functionality

phLOGIN appears very different to the end user.

- Full Screen Login

To the End User the most obvious improvement that phLOGIN brings is that the whole screen is used, gone is the line at a time *login*, here is a smart, modern start screen.

Colour may be employed if the terminal supports it.

- Date, Time and MOTD

The date and time are probably displayed on the screen, the Message Of The Day is displayed (and updated as it changes as the clock ticks).

- Arrow Keys

The user may use the arrow keys to edit input. Backspace and Delete are regarded as synonymous.

Gone is the nightmare of guessing the erase key.

- Function Keys

The System Administrator may have bound functions to some of the function keys.

phLOGIN examines the file specified when the key is pressed, if it is executable it runs it, or if not it displayed it to the screen. This allows both a simple help facility, and the ability to run specified (and trusted) programs without logging in.

- Native Language

The wording on the screen and error messages may be displayed in the user's native language. Most computer users do **not** understand English.

- Correct Clocks on Remote Login

Those who remotely login to machines in different time zones have got used to having the time displayed wrongly (from their point of view). This can be quite easily cured:

```
# Get timezones right for the non UK countries.
# Set this up. Note how some should be put in quotes.
/R:paris,bonn,madrid/ { -e TZ='MET-1MET' }
/R:canberra/          { -e TZ='EST-10' }
/R:kremvax/            { -e TZ='MCT-3MST' } # Moscow site
/R:whsun/              { -e TZ=EST5EDT }   # White House
/R:cairo/              { -e TZ='ET-2' }
```

To eliminate confusion, the concept of a system administrator's time zone is supported, this is the time zone which is used in allows (-A).

Much of the description above uses "may", this is because the form and function of the login screen is completely definable by the System Administrator. An example screen file is given in Figure 2. The \$...\$ sequences are substituted, e.g. \$T\$ for the current time and \$M1\$ for

```
# Demo screen file, ADDW 20 May 1991.
#
# Allow the user to get a screen full of help on how to login.
##fl=/etc/phlogin.d/lhelp
#
                                     $T$          $>$
+-----+-----+
|                                     |
|               Welcome to $n$      |
|                                     |
|      Username: $U$                $>$ |
|      Password: $P$                $>$ |
|                                     |
+-----+-----+
|                                     |
| $M1$                               $>$ |
| $M2$                               $>$ |
| $M3$                               $>$ |
| $M4$                               $>$ |
| $M5$                               $>$ |
| $M6$                               $>$ |
|                                     |
|               Networked phLOGIN    |
|      Login from $run$              |
|                                     |
+-----+-----+
| $E$                               $>$ |
|                                     |
+-----+-----+
                                     $D$          $>$
# end
```

Figure 2: An example screen file

the first line of */etc/motd*. The username and password are prompted for at *\$U\$* and *\$P\$*. In all 42 different sequences are defined.

3.4. System Administrator Control

There is much that the System Administrator can do in controlling what is to happen in individual terminals. Much of this has been discussed above, new features are:

- Global allow/disallow of logins

If you want to do a backup it is easy to log people off the system. phLOGIN allows a specified file to be used as BSD *login* uses */etc/nologin*. phLOGIN however tracks the existence and contents of the file on to the screen so that the user does not need to keep on trying to login to find out if it has been removed.

- Maximum File Size

The *ulimit* on some systems is quite small. This frequently causes problems with database users. As the limit can only be increased by *root* I have occasionally found such applications running as *root*.

phLOGIN allows this to be set appropriately.

- Terminal Groups

Very often a whole group of terminals need to be treated in a similar way. phLOGIN allows options to be read from a file. This saves effort and errors.

3.5. Networking

This is a recently released module and involves replacement of the *rlogin* and *telnet* daemons. The whole basis of the extra protection offered is that the phLOGIN options list is built dynamically and what is included can depend on where the remote rlogin is coming from.

This allows the System Manager to provide access in a measure commensurate with the level of trust that he has in the remote system or its users. As mentioned above username equivalences can also be controlled.

4. Porting Issues

Most of phLOGIN can be ported to most systems with little, if any, work being needed. For hardware with an established ABI (Applications Binary Interface), such as the i386, programs compiled elsewhere will work properly. This is true for the small utilities that come with phLOGIN like the log print program.

phLOGIN itself, unfortunately, works in an area where many manufacturers have added "security specials" to their system. Many of these have been in an effort to attain a US DOD "Green Book" rating. These come in several flavours:

- Audit Trails

Login success and failure records may need to be written. These will have to be in a specific format, there may be library routines to help with this.

- TCBS

This is a Trusted Computing Base which is generally implemented as a Control Database which contains an entry for every user who is known to the system. In addition to duplicating some or all of the information in */etc/passwd* various other items such as recent login history and "what the user is allowed to do" are kept here.

As these are very manufacturer specific the format and access methods of the TCB vary widely.

- Kernel Authorisations

Before completing a login a system may demand that an authorisation vector is initialised. This will tell the kernel what system calls it is to honour, and perhaps add an extra dimension to file access control. The information for this will come from the TCB.

There are other arbitrary actions which may also need to be taken, e.g. the *setluid()* system call on a SCO V.3 machine.

The main problem with all of the above is the documentation. This is frequently sparse, occasionally inaccurate and sometimes unavailable. With HP, for instance, the format of audit trail information is not specified; I asked the European Support Centre for help, they did not know and the US office refused to help them.

The problems that we have had are because we are working in an area where the pressures of standardisation are not as great as those for adding special features which will make a particular system "better than all others". Standards have not yet caught up here – most applications don't need to know about them.

5. Future Work

A product cannot stand still, requirements change as do users' needs. Much of the list below has been asked for by users:

- **Inactivity Timeouts**

If a user walks away from a terminal that terminal should be secured against use by others until the user returns. Some systems offer facilities whereby a daemon examines terminals for inactivity (i.e. nothing being typed on it) and forces a logout.

This is not a good way of doing things, what if the user was doing a long and complicated database select? A logout is the wrong answer.

A better solution is to grab control of the terminal, clear the screen and insist that the login password be re-entered before the session can continue.

- **System Manager Setup Tools**

Currently phLOGIN setup is done by editing configuration files. This requires a greater level of knowledge and skill than we should assume. A menu driven setup tool is needed.

- **Modem Handling**

phLOGIN needs to be able to talk to modems so that it can understand when the baud rate is changed on call connection.

- **Smart Cards**

The use of Swipe Cards and Smart Cards to identify individuals is becoming increasingly popular. A hook already exists to allow a verification program to be run, we need to use this to access the cards.

- **Network Encryption**

Data transferred between machines is not encrypted under current TCP/IP protocols. It is not very difficult to plug into a network and read the messages as they go by.

Encrypting the network traffic would raise network security.

We have recently been approached by a company that wants this.

- **Expert System Audit Trail Analysis**

One of the problems with most security tools is that they generate large audit trails. This is the correct thing to do, the difficulty is in analysing them.

The hundreds or thousands of audit records that will be generated daily need inspection, but to expect a human to spot anything other than blatant attacks is not realistic. A human inspector (i.e. the System Administrator) also has many other jobs to do and only works at certain times of day, but the time when (s)he next looks might be too late.

It is easy to ring an alarm whenever there is a security violation, but many of these will be due to genuine user errors; the attacks need to be sorted out from the mistakes. An attack may be subtle, e.g.: one may decide that two successive violations on a terminal constitute an error not an attack, but a rush of "errors" on terminals all in the same room is likely to be an attack rather than real errors – except when people sign on in the morning.

The point is that if alarm bells are rung too often they are ignored. If they are not rung often enough security breaches are probable. We would like to build an expert system to help in this area.

MANIFOLD: A Language for Specification of Inter-Process Communication

Farhad Arbab and Ivan Herman

*Centre for Mathematics and
Computer Science (CWI), The Netherlands*
farhad@cwi.nl ivan@cwi.nl

Abstract

Management of the communications among a set of concurrent processes arises in many applications and is a central concern in parallel computing. In this paper, we introduce a language whose sole purpose is to describe and manage complex interconnections among independent, concurrent processes. In the underlying paradigm of this language, MANIFOLD, the primary concern is not with *what* functionality the individual processes in a parallel system provide. Instead, the emphasis is on *how* these processes are inter-connected and how their interaction patterns change during the execution life of the system.

It is interesting that the conceptual model behind the MANIFOLD language immediately leads to a very simple, but non-conventional model of computation. Contrary to most other models, *computation* in MANIFOLD is built out of *communications*. As such it advocates a view point reminiscent of the connectionist view: that all (conventional) computation can be expressed as interactions.

1. Introduction

Specification and management of the communications among a set of concurrent processes is at the core of many problems of interest to a number of contemporary research trends. Although communications issues come up in virtually every type of computing, and have influenced the design (or at least, a few constructs) of most programming languages, not much effort has been spent on conceptual models and languages whose sole prime focus of attention is on process interaction. Notable exceptions include the theory of neural networks, and to some extent, the concept of dataflow programming and the theory of Communicating Sequential Processes.

In this paper, we introduce MANIFOLD: a language whose sole purpose is to describe and manage complex interconnections among independent, concurrent processes. A detailed description of the MANIFOLD model and the syntax and semantics of the MANIFOLD language is of course beyond the scope of this paper. The specification of the MANIFOLD model and

system is given elsewhere [Arb91a]. We summarize only enough of the description of the MANIFOLD model to give an impression of its potentials. To give a flavor of the MANIFOLD language and show how it is used in parallel computing, in this paper we explain the implementation of a parallel bucket sort algorithm in MANIFOLD. More examples of the use of the MANIFOLD language are given elsewhere [Arb90a] and a larger example, related to computer graphics and interaction, has been published in [Soe91a]. Only enough of the syntax and semantics of the language is discussed here to make the critical parts of the bucket sort example program understandable.

It is interesting that the conceptual model behind the MANIFOLD language immediately leads to a very simple, but non-conventional model of computation. The MANIFOLD model is conceptually as powerful as conventional models, e.g., the Turing Machine. However, contrary to most other models, *computation* in MANIFOLD is built out of *communications*.

The rest of this paper is organized as follows. Section 2 presents some of the motivation behind the design of MANIFOLD. In Section 3, we inspire an intuitive feeling for what MANIFOLD programming is like by comparing and contrasting it with a number of different styles of programming. Section 4 contains a summary of the key concepts in the MANIFOLD model. Section 5 explains the critical part of a complete MANIFOLD program which implements a parallel bucket sort algorithm. The complete MANIFOLD program itself appears in Appendix A. A flavor of the syntax and the semantics of the MANIFOLD language can be skimmed from the explanation of the piece of code presented in Section 5. Section 6 mentions some other application areas where the MANIFOLD style of programming seems to be a promising approach. Section 7 contrasts MANIFOLD with a few other systems with similar features or concerns. Finally, Section 8 contains a few concluding remarks about MANIFOLD.

2. Motivation

One of the fundamental problems in parallel programming is coordination and control of the communications among the sequential fragments that comprise a parallel program. Programming of parallel systems is often considerably more difficult than (what intuitively seems to be) necessary. It is widely acknowledged that a major obstacle to a more widespread use of massive parallelism is the lack of a coherent model of how parallel systems must be organized and programmed. To complicate the situation, there is an important pragmatic concern with significant theoretical consequences on models of computation for parallel systems. Many user communities are unwilling and/or cannot afford to ignore their previous investment in existing algorithms and "off-the-shelf" software and migrate to a new and bare environment. This implies that a suitable model for parallel systems must be *open* in the sense that it can accommodate components that have been developed with little or no regards for their inclusion in an environment where they must interact and cooperate with other modules.

Many approaches to parallel programming are based on the same computation models as sequential programming, with added on features to deal with communications and control. There is an inherent contradiction in such approaches which shows up in the form of complex semantics for these added on features. The fundamental assumption in sequential programming is that there is only one active entity, *the* processor, and the executing program is in control of this entity, and thus in charge of the application environment. In parallel programming,

there are many active entities and a sequential fragment in a parallel application cannot, in general, make the convenient assumption that it can rely on its incrementally updated model of its environment.

To reconcile the "disorderly" dynamism of its environment with the orderly progression of a sequential fragment "quite a lot of things" need to happen at the explicit points in a sequential fragment when it uses one of the constructs to interact with its environment. Hiding all that needs to happen at such points in a few communication constructs within an essentially sequential language, makes their semantics complex. Inter-mixing the neat consecutive progression of a sequential fragment, focused on a specific function, with updating of its model of its environment and explicit communications with other such fragments, makes the dynamic behavior of the components of a parallel application program written in such languages difficult to understand. This may be tolerable in applications that involve only small scale parallelism, but becomes an extremely difficult problem with massive parallelism.

Separating the communication issues from the functionality of the component modules in a parallel system makes them more independent of their context, and thus more reusable. It also makes it possible to delay decisions about the interconnection patterns of these modules, which may be changed subject to a different set of concerns.

There are even stronger reasons in distributed programming for delaying the decision about the interconnections and the communication patterns of modules. Some of the basic problems with the parallelism in parallel computing become more acute in distributed computing, due to the distribution of the application modules over loosely coupled processors, perhaps running under quite different environments in geographically different locations. The implied communications delays and heterogeneity of the computational environment encompassing an application, become more significant concerns than in other types of parallel programming. This mandates, among other things, more flexibility, reusability, and robustness of modules with fewer hard-wired assumptions about their environment.

The tangible payoffs reaped from separating the communications aspect of a multi process application from the functionality of its individual processes include clarity, efficiency, and reusability of modules and the communications specifications. This separation makes the communications control of the cooperating processes in an application more explicit, clear, and understandable at a higher level of abstraction. It also encourages individual processes to make less severe assumptions about their environment. The same communications control component can be used with various processes that perform functions *similar* to each other from a very high level of abstraction. Likewise, the same processes can be used with quite different communications control components. This helps modularity, efficiency, and reusability.

3. What is it Like?

The Webster's dictionary defines the term *manifold* as an adjective to mean:

1. having many forms, parts, etc.
2. of many sorts
3. being such in many ways
4. operating several parts of one kind.

It also defines *manifold* as a noun to mean:

a pipe with several outlets, as for conducting cylinder exhaust from an engine.

MANIFOLD can be viewed from several different perspectives, each revealing similarities with the features and concerns of a different set of models and systems. A comparison of MANIFOLD and some such models and systems is made in Section 7. However, it is useful to establish a few approximate reference points to inspire an intuitive feeling for what MANIFOLD is all about before encountering the details. To that end, we mention dataflow programming, shell scripts, and event driven programming in this section.

To the extent that the primary focus in MANIFOLD is the connections among processes, not the processes themselves, it is a *conductor* that orchestrates the interactions among a set of cooperating concurrent processes, without interfering with their internal operations. As such, MANIFOLD programming is vaguely reminiscent of writing shell scripts in a system like UNIX. Similar to a shell script, the concurrency and interconnection issues are completely outside of the processes. However, the possibilities for defining and dynamically changing the interconnections among processes in MANIFOLD go much beyond what is offered in such simple shell scripts.

Orchestration of the interactions among a set of processes in MANIFOLD is done in an entity with multiple inlets and outlets, called a *manifold*. As the conductor of such interactions, a manifold has a number of *states*, each specifying a specific connection pattern. Connection patterns define links between the input and output ports of various processes, called *streams*, through which the information produced by one process is made available for consumption to another.

A manifold goes through state transitions as a result of observing in its environment the occurrences of *events* in which it is interested. State transitions cause dismantling of the interconnections set up in pre-transition states, and establish the ones defined in the post-transition states. As such, events are the principal control mechanism in MANIFOLD, which makes it an event driven programming system.

The streams among processes in MANIFOLD form a network of links for the flow of information that is reminiscent of dataflow networks. However, there are several major differences between MANIFOLD and dataflow programming. In MANIFOLD the connection patterns among processes change dynamically. Furthermore, processes are created and deleted dynamically as well. This by itself makes the connections graph of a MANIFOLD program, which is the combined effect of all its manifolds, very dynamic. However, there is more. The manifestation of a single manifold is of course a single (dynamically changing) process inter-connection graph. Since manifolds too are processes, the combined graph of a MANIFOLD program is indeed not a simple graph, but a hyper-graph, where each node in itself is a dynamically changing graph of connections among processes.

Although conceptually, the dominant control mechanism in MANIFOLD is event driven, the dataflow type, data driven style of control through streams is at least equally as important. A manifold can internally raise an event for itself, causing a state transition. This can be, for instance, due to the arrival of a unit of information in the pre-transition state through a certain stream, and may also depend on the contents of this information. Thus, there is a smooth transition between the two

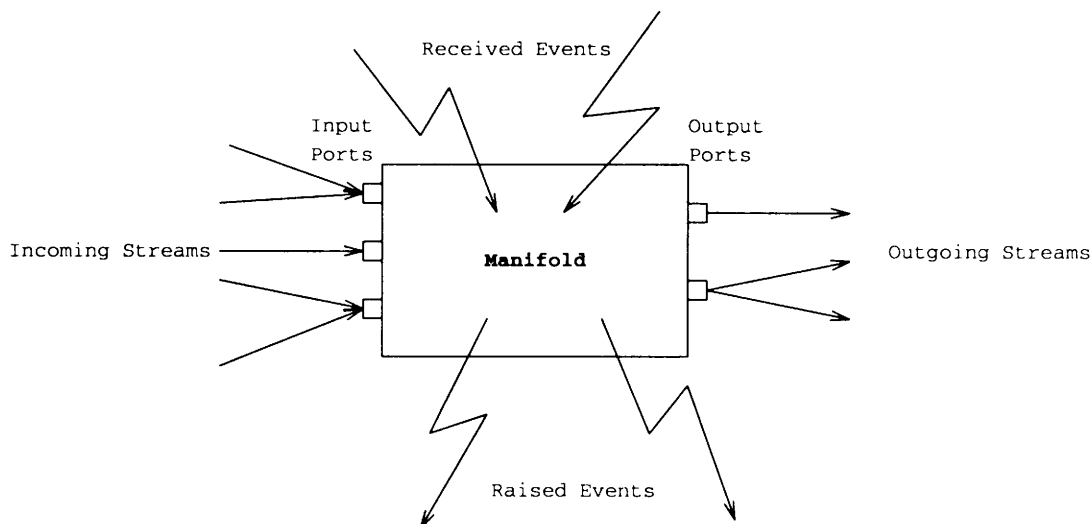


Figure 1: The model of a process in MANIFOLD

mechanisms of control in MANIFOLD. The coexistence of event driven and data driven control gives MANIFOLD a unique flavor.

4. The MANIFOLD Model of Computation

The basic components in the MANIFOLD model of computation are processes, events, ports, and streams. A process is a *black box* with well defined ports of connection through which it exchanges *units* of information with the other processes in its environment. The internal operation of some of these black boxes are indeed written in the MANIFOLD language, which makes it possible to open them up, and describe their internal behavior using the MANIFOLD model. These processes are called manifolds. In general, a process in MANIFOLD does not, and need not, know the identity of the processes with which it exchanges information. Figure 1 shows an abstract representation of a MANIFOLD process.

The interconnections between the ports of processes are made with streams. A stream represents a flow of a sequence of units between two ports. Streams are constructed and removed dynamically between ports of the processes that are to exchange some information. The constructor of a stream need not be the sender or the receiver of the information to be exchanged: any third party manifold process can define a connection between the ports of a producer process and a consumer process. Furthermore, stream definitions in MANIFOLD are generally additive. Thus a port can simultaneously be connected to many different ports through different streams. The flows of units of information in streams are automatically replicated and merged at outgoing and incoming port junctions, as necessary. The units of information exchanged through ports and streams, are *passive* pieces of information that are synchronously produced and synchronously consumed at the two ends of a stream, with their relative order preserved.

Orthogonal to the stream mechanism, there is an event mechanism for information exchange in MANIFOLD. Contrary to units in streams, events are *active* pieces of information that are broadcast by their sources in the environment. In principle, *any* process in the environment can pick up such a broadcast event. In practice, usually only a few processes

pick up occurrences of each event, because only they are *tuned in* to their sources. Occurrences of the same event from the same source can override each other from the point of view of some observer processes, depending on the difference between their *sampling rate* and the occurrence rate of the event.

Events are generally raised synchronously by their sources and dissipate through the environment. They are active pieces of information in the sense that in general, they are observed asynchronously and once picked up, they preemptively cause a change of state in the observer. Events are the primary control mechanism in MANIFOLD. Each state in a manifold defines a pattern of connections among the ports of some processes. The corresponding streams implementing these connections are created as soon as a manifold makes a state transition (caused by an event) to a new state, and are deleted as soon as it makes a transition from this state to another one. In general, the set of sources whose events are honored by an observer manifold, as well as the set of specific events which are honored, are both state dependent.

The remainder of this section contains more detailed definitions of the basic concepts of the MANIFOLD model.

4.1. Processes

A *process* is an independent, autonomous, active entity that executes a procedure. A process has its own private processor and memory. Independence means that a process is not necessarily aware of the number and the nature of other processes that are simultaneously active in its environment. The environment of a process contains the set of other processes that directly or indirectly influence the behavior of the process or its performance.

Autonomous means that conceptually, no process exerts direct control on any other process. The only way to influence a process is through its input and output streams and the events to which it is sensitive. For example, once a process is activated, it cannot be "forced" to terminate by other processes, including its activator. However, it can be "asked" to terminate, by placing appropriate symbols in its input streams, or by raising an appropriate event. Similarly, there is no guarantee that a process will indeed read from its input streams, write to its output streams, react to some arbitrary event, or stay alive for any length of time.

The above model of communication is powerful enough to support all forms of interprocess communication. Therefore, in principle, there is no need for other forms of communication among processes. In practice, however, it may be desirable to allow other forms of inter-process communication, e.g., for convenience. For example, processes may need to communicate and influence each other through other means for purposes such as resource management, job control, side effects (e.g., files), interaction with the real world, etc., and may use mechanisms such as message passing, shared memory, etc. While the MANIFOLD model does not preclude such communications, *it assumes that all communication of interest with a process takes place through its input and output streams and via events.*

There are two kinds of processes: *atomic processes* and *manifolds*. An atomic process is similar to a black box whose internal structure and behavior are unknown. The set of atomic processes is application dependent, and thus, is neither predefined nor fixed. Examples of atomic processes include processes written in some programming

language other than MANIFOLD, a hardware device, and a person interacting with a program.

A manifold is a process whose behavior and structure are described in the MANIFOLD language by a *manifold definition*. Manifolds “orchestrate” the communication and interaction among processes (atomic processes and other manifolds alike), and provide a dynamic means of control over a multiprocessing environment. The processor that runs a manifold is called the *manifold processor*.

4.2. Streams

A *stream* is a sequence of bits, grouped into (variable length) *units*. A stream represents a reliable, directed flow of information in time. Reliable means that the bits placed into a stream are guaranteed to flow through without loss, error, or duplication, with their order preserved. It does not, however, imply timing constraints. Directed means that there are always two identifiable ends in a stream, a *source* and a *sink*.

The size and the contents of the units that flow through streams are defined by their sources. Although units are meaningful inside streams, they imply no corresponding boundaries, types, tags, or interpretation on their contents at their sinks. Unit boundaries are used in streams to preserve the integrity of their information contents, and for synchronization purposes.

Conceptually, a stream in MANIFOLD has an unspecified capacity that is used as a FIFO queue, enabling asynchronous production and consumption of units by its source and sink. Streams in MANIFOLD are dynamically constructed and dismantled.

4.3. Ports

The connection between streams and processes is through ports. A *port* is a regulated opening at the boundary of a process, through which the information produced (consumed) by the process is placed into (picked up from) a stream. Regulated means that the information can flow in only one direction through a port: it either flows into or out of the process.

While streams are independent entities outside of processes, ports are properties of processes and are defined and owned by them. Information placed into one of its output ports by a process, flows out of the port only when it is connected to a stream. This ensures that no information is lost if a process writes to one of its output ports while it is not connected to any stream.

4.4. Events

An event is an asynchronous, non-decomposable message, broadcast by a process to its environment. Broadcasting such a message is called *raising the event*. Events are identified by their names, and can also be distinguished based on their sources (except, perhaps, when they are raised by atomic processes).

Although conceptually, an event is broadcast when it is raised, only a subset of the processes in its environment can pick up the broadcast and react to it. A process that picks up an occurrence of an event is called an *observer* (of the event and of its source). To pick up a broadcast event, a manifold must be in a state wherein the source of the event is *visible* to the manifold. In general, a manifold reacts only to a

subset of the events it observes. These are the ones for which it has an event handler. The other observed events are ignored. Reacting to an event always causes a change of state in a (receiver) manifold.

Different occurrences of the same event from the same source may override each other before some of their observers get a chance to observe them. The overridden event occurrences are thus lost to those observers. This means that some event occurrences may be lost to some observers, but not to others, depending on the speed with which they *sample* their environment. Occurrences of events from different sources do *not* override each other. Occurrences of different events from the same source do not override each other, either.

An observed event may cause a change of state in a manifold, or it may decide to ignore the event. The change of state in a manifold may affect its sensitivity and reaction to future events. In each new state, a manifold begins to *react* to the observed event that caused the change of state. An observed event may *preempt* a manifold's attempt to react to a previously observed event (from the same or a different source).

5. A Parallel Bucket Sort Example

In this section we introduce some of the key concepts of the MANIFOLD system by presenting a MANIFOLD program that implements a parallel bucket sort algorithm. The complete MANIFOLD program appears in Appendix A. However, only the critical parts of the program are explained in this section.

Our parallel sort algorithm is similar to the one presented by Suhler et al [Suh90a], for a dynamic dataflow environment. The two algorithms, however, are not identical. The essence of the algorithm is as follows. There exists an atomic process (perhaps a piece of hardware) that performs an efficient sorting of a number of input units, provided that this number is below a fixed threshold, b . For example, if b is 2, all that this atomic process has to do is a simple compare to decide the proper order of its two input units. The aim is therefore to start off as many instances of this atomic process as possible, passing up to b units of the incoming stream to each, and then merge the sorted output streams of the parallel sort processes into the final sorted output stream.

The core of the solution is a manifold called `Sort_def`. This manifold receives all the units on its input and produces the sorted units on its output. It counts the number of incoming units and forwards the first bucket of units to an instance of the atomic sorting process. The size of a bucket, b , is the value of the variable `limit`. In case the original input contains more than one bucket-full of units, `Sort_def` directs the output of the atomic sorter to a so called `Merger` manifold. The `Sort_def` manifold then activates a new instance of itself and directs the rest of its incoming units to this new instance. The output of the new instance of `Sort_def` is directed to the same `Merger`. Finally, the output of the `Merger` is connected to the output of the former instance of `Sort_def`.

The `Merger` manifold merges its two incoming streams of ordered units into a single output stream of ordered units. We do not discuss the details of the `Merger` manifold here, because explaining more details of the syntax of the language is beyond the scope of this paper. (The *manners* that appear in the `Merger` manifold, for example, are dynamically nested subroutine calls.)

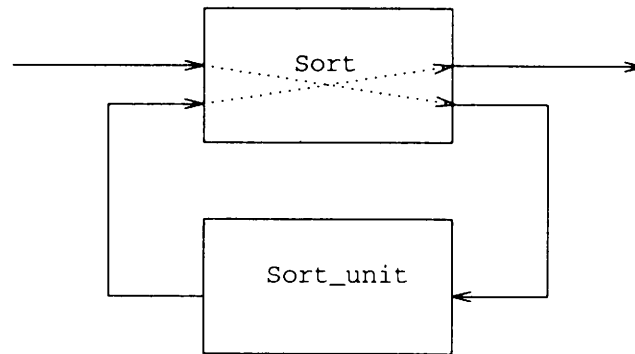


Figure 2: Terminal Case for the Recursion

Observe that the behavior of the `Sort_def` manifold is recursive. The terminal case for this recursion is when the number of incoming units is less than the bucket size. In this case the corresponding instance of `Sort_def` simply connects the output of its atomic sorter to its own output (see Figure 2). In other cases, it splits the incoming units between an instance of the atomic sorter and another instance of itself (see Figure 3).

We use the core of the program, the `Sort_def` manifold, as a reference to explain how a manifold works, and clarify its syntax. This portion of the program appears below. A `//` symbol marks the start of a comment that extends to the end of the line.

The header of this manifold defines its name, `Sort_def`, and its parameter, `limit`. In its *public* declarations section following its header, the input and output ports of the manifold are defined. The declarations for `input` and `output` are indeed redundant, because they are the same as the default definitions for all manifolds. In addition to the ports, `sort_full` is also declared here as an event exchanged between this manifold and its environment.

The body of the manifold consists of the lines enclosed between the symbols `{` and `}`. In this case, it starts with some *private* declarations, all of which happen to define instances of various processes. For example, the line `process Sort is Sort_def.` defines `Sort` as an instance of the manifold `Sort_def`, and `process Count is count1.` defines `Count` to be an instance of the library manifold `count1`.

The bulk of the body of a manifold consists of a number of blocks, each labeled with a list of events. In this case it so happens that each block has only one label. The event `start` is raised automatically when an instance of a manifold is activated. The block labeled `start` is thus the first block that is entered in every manifold instance. In the case, the `Sort_def` manifold activates an instance of an atomic sorter process and an instance of a special manifold which is used to count the number of incoming units. It then sets up a pipeline connecting its own standard input to the standard input of its atomic sorter, with the counter in between. Thus, all incoming units will be directed to the atomic sorter. The manifold processor of this instance of the `Sort_def` manifold is now waiting for the expiration of the pipeline it just set up.

The `count1` manifold basically passes all of its input units on to its standard output, up to the point in time when it has passed `limit` number of units. It then halts. This event is observed by the instance

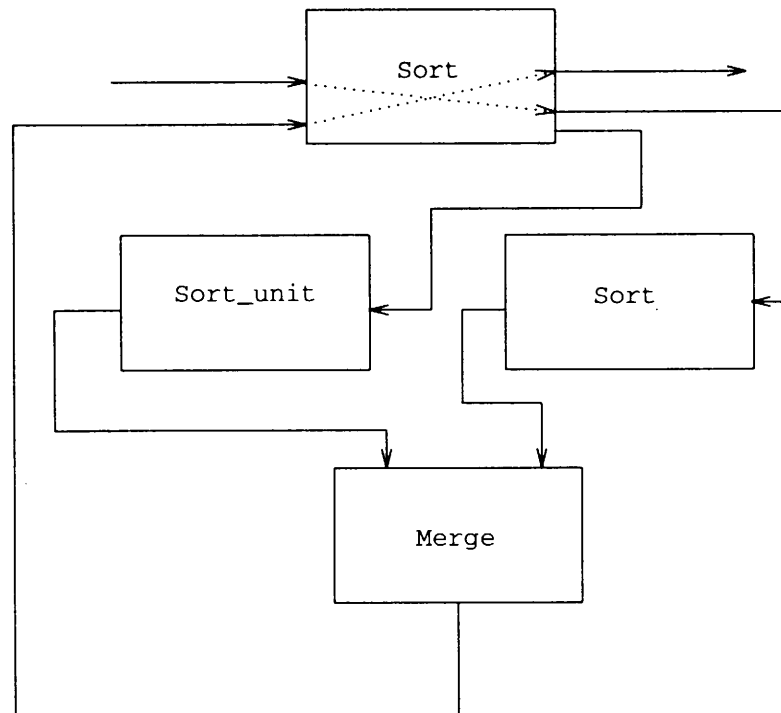


Figure 3: *Recursive branch*

of the `Sort_def` manifold that activated this counter instance, and it causes a state transition in `death.Count`. Reacting to this event, the processor of the `Sort_def` instance leaves the `start` block, dismantling the pipeline set up there, and finds the proper handler block for `death.Count`. This happens to be the last block in the `Sort_def` manifold.

Since the pipeline set up in `start` is now broken, no more units will flow to `Sort_units`. Instead, a `Merge` and a new instance of `Sort_def` are activated and, then, a number of parallel pipelines are set up in the block labeled `death.Count`. This block consists of the two activate actions mentioned above, followed by a construct called a *group*. A group is a comma-separated list of pipelines enclosed in a pair of parentheses, and represents parallel operation of its component pipelines. This is the situation depicted in Figure 3. From now on, all incoming units flow to the recursively activated instance of `Sort_def`.

Note that a slight modification of `Sort_def` in this block can improve the performance of the sort algorithm by simplifying the function of the `Merge`. `Sort_def` can use the first incoming unit as a “pivot” and send the first limit number of units that are smaller than this pivot to the `Sort`, and the rest to its recursive incarnation.

Note also that the part of the program which is really dependent on the MANIFOLD language gives just a skeleton for solving the whole sorting problem; how the effective sorting is done is hidden in the two atomic processes `Sort_units_def` and `Compare_units_def`. This also means, however, that using the very same “skeleton” but using two different atomic processes it is possible to realize a parallel program for very different purposes (for example a fast filtering program). This shows the power offered by the modular structure of the MANIFOLD language.

```
//
// Effective Sorter
//   I/O ports:
//   input:           units to sort
//   output:          sorted units
//   Caught events:
//   sort_full:       the number of incoming events have reached "limit"
//
// Makes a recursive call to itself if the number of the incoming units
// is more than the "limit"
// To count the incoming event, the manifold "count1" is used.
// Halts when all units are sorted and sent
//
Sort_def( limit )
port in input.
port out      output.
{
    process Sort          is      Sort_def.
    process Sort_units    is      Sort_units_def.
    process Merge         is      Merge_def.
    process Count         is      count1.

    start:
        activate Sort_units;
        activate Count( limit );
        input → Count → Sort_units.

//-----//

    disconnected.input: // There are no more units than limit!
        deactivate Count;
        Sort_units → output.

    death.Sort_units:
        halt.

//-----//

    death.Count:
        activate Merge;
        activate Sort(limit);
        ( Sort_units → Merge.b,
          Sort       → Merge.a,
          input      → Sort,
          Merge      → output ).
}
```

Program 1:

6. Other Applications

The possible application areas for MANIFOLD are numerous. It is an effective tool for describing interactions of autonomous active agents that communicate in an environment through message passing and global broadcast of events. For example, elaborate user interface design means planning the cooperation of different entities (the human operator being one of them) where the event driven paradigm seems particularly useful. In our view, the central issue in a user interface is the design and implementation of the communication patterns among a set of modules. Some of these modules are generic (application independent) programs for acquisition and presentation of information expressed in forms appealing to humans. Others are, ideally, acquisition/presentation-independent modules that implement various functional components of a specific application. Previous experience with systems like DICE (see [Lie87a] or [Sch88a]) has shown that con-

currency, event driven control mechanisms, and general interconnection networks[†] are all necessary for effective graphics user interface systems. MANIFOLD supports all of that and in addition, provides a level of dynamism that goes beyond many other user interface design tools.

Separating the specification of the dynamically changing communication patterns among the modules from the modules themselves seems to lead to better user interface architectures. A similar approach can also be useful in applications of real time computing where dynamic change of interconnection patterns (e.g., between measurement and monitoring devices and actuators) is crucial. Complex process control systems, must orchestrate the cooperation of various programs, digital and/or analogue hardware, electronic sensors, human operators etc. Such interactions may be more easily expressed and managed in MANIFOLD.

Coordination of the interactions among a set of cooperating autonomous intelligent experts is also relevant in Distributed Artificial Intelligence applications, *open* systems, such as Computer Integrated Manufacturing applications, and the complex control components of systems such as Intelligent Computer Aided Design.

Recently, scientific visualization has raised similar issues as well. The problems here typically involve a combination of massive numerical calculations (sometimes performed on supercomputers) and very advanced graphics. Such functionality can best be achieved through a distributed approach, using segregated software and hardware tools. Tool sets like the Utah Raster Toolkit [Pet86a] are already a first step in this direction, although in case of this toolkit the individual processes can be connected in a pipeline fashion only. More recently, software systems like the apE system of the Ohio Supercomputer Center [Dye90a] work on the basis of inter-connecting a whole set of different software/hardware components in a more sophisticated communication network. An "orchestrator" like MANIFOLD can prove to be quite valuable in such applications.

Advances in neuroscience have shown that to properly model the nervous system requires massively parallel systems where, in contrast to conventional neural networks, each node in the system has the computational complexity of a microcomputer (see e.g. [Mat88a] or [Tho91a]). MANIFOLD may offer an appropriate paradigm for expressing the dynamic behavior of such complex inter-connection networks.

7. Related Work

The general concerns which led to the design of MANIFOLD are not new. The CODE system (see [Bro89a] and also [Bro90a]) provides a means to define dependency graphs on sequential programs. The programs can be written in a general purpose programming language like Fortran or Ada. The translator of the CODE system translates dependency graph specifications into the underlying parallel computation structures. In case of Ada, for example, these are the language constructs for rendezvous. In case of languages like Fortran or C, some suitable language extensions are necessary. Just as in traditional dataflow models, the dependency graph in the CODE system is static.

The MANIFOLD streams that interconnect individual processes into a network of cooperating concurrent active agents are somewhat similar to links in dataflow networks. However, there are several important

[†] In case of DICE, this is actually a strict hierarchy, and has turned out to be one of its shortcomings in practice.

differences between MANIFOLD and dataflow systems. First, dataflow systems are usually fine-grained (see for example Veen [Vee86a] or Herath et al [Her88a] for an overview of the traditional dataflow models). The MANIFOLD model, on the other hand, is essentially oblivious to the granularity level of the parallelism, although the MANIFOLD system is mainly intended for coarser-grained parallelism than in the case of traditional dataflow. Thus, in contrast to most dataflow systems where each node in the network performs roughly the equivalent of an assembly level instruction, the computational power of a node in a MANIFOLD network is much higher: it is the equivalent of an arbitrary process. In this respect, there is a stronger resemblance between MANIFOLD and such more advanced dataflow environments like the so called Task Level Dataflow Language of Suhler et al [Suh90a].

Second, the dataflow like control through the flow of information in the network of streams is not the only control mechanism in MANIFOLD. Orthogonal to the mechanism of streams, MANIFOLD is an event driven paradigm. State transitions caused by a manifold's observing occurrences of events in its environment, dynamically change the network of a running program. This seems to provide a very useful complement to the dataflow like control mechanism inherent in MANIFOLD streams.

Third, dataflow programs usually have no means of reorganizing their network at run time. Conceptually, the abstract dataflow machine is fed with a given network once at the initialization time, prior to the program execution. This network must then represent the connections graph of the program throughout its execution life. This lack of dynamism together with the fine granularity of the parallelism cause serious problems when dataflow is used in realistic applications. As an example, one of the authors of this paper participated in one of the very rare practical projects where dataflow programming was used in a computer graphics application [Hag90a]. This experience shows that the time required for effective programming of the dataflow hardware (almost 1 year in this case) was not commensurate with the rather simple functionality of the implemented graphics algorithms.

The previously mentioned TDFL model [Suh90a] changes the traditional dataflow model by adding the possibility to use high level sequential programs as computational nodes, and also a means for dynamic modification of the connections graph of a running program. However, the equivalent of the event driven control mechanism of MANIFOLD does not exist in TDFL. Furthermore, the programming language available for defining individual manifolds seems to be incomparably richer than the possibilities offered in TDFL.

Following a very different mental path, the authors of LINDA [Car89a] were also clearly concerned with the reusability of existing software. LINDA uses a so called generative communication model, based on a *tuple space*. The tuple space of LINDA is a centrally managed space which contains all pieces of information that processes want to communicate. A process in LINDA is a black box. The tuple space exists outside of these black boxes which, effectively, do the real computing. LINDA processes can be written in any language. The semantics of the tuples is independent of the underlying programming language used. As such, LINDA supports reusability of existing software as components in a parallel system, much like MANIFOLD.

Instead of designing a separate language for defining processes, the authors of LINDA have chosen to provide language extensions for a number of different existing programming languages. This is neces-

sary in LINDA because seemingly, its model of communication (i.e., its tuple space and the operations defined for it) is not sufficient by itself to express computation of a general nature. The LINDA language extensions on one hand place certain communication concerns inside of the "black box" processes. On the other hand, there is no way for a process in LINDA to influence other processes in its environment directly. Communication is restricted to the information contained in the tuples, synchronously and voluntarily placed in and picked from the tuple space. We believe a mechanism for direct influence (but not necessarily direct control), such as the event driven control in MANIFOLD, is desirable in parallel programming.

One of the best known paradigms for organizing a set of sequential processes into a parallel system is the Communicating Sequential Processes model formalized by Hoare [Hoa85a]. CSP is a very general model which has been used as the foundation of many parallel systems. Sequential processes in CSP are abstract entities that can communicate with each other via pipes and events as well. CSP is a powerful model for describing concurrent systems. However, there is no way in CSP to dynamically change the communications patterns of a running parallel system, unless such changes are hard coded inside the communicating processes. In contrast, MANIFOLD clearly separates the functionality of a process from the concerns about its communication with its environment, and places the latter entirely outside of the process. It then completely takes over the responsibility for establishing and managing the interactions among processes in a parallel system.

8. Conclusion

The unique blend of event driven and data driven styles of programming, together with the dynamic connection hyper-graph of MANIFOLD seems to provide a promising paradigm for parallel programming. The emphasis of MANIFOLD is on orchestration of the interactions among a set of autonomous *expert* agents, each providing a well-defined segregated piece of functionality, into an integrated parallel system for accomplishing a larger task.

In the MANIFOLD model, each process is responsible to *protect* itself from its environment, if necessary. This shift of responsibility from the producer side to the consumer seems to be a crucial necessity in open systems, and contributes to reusability of modules in general. This model imposes only a "loose" connection between an individual process and its environment: the producer of a piece of information is not concerned with who its consumer is. In contrast to systems wherein most, if not all, information exchange takes place through targeted send operations within the producer processes, processes in MANIFOLD are not "hard-wired" to other processes in their environment. The lack of such strong assumptions about their operating environment makes MANIFOLD processes more reusable.

The MANIFOLD model of communication is conceptually powerful enough to express general purpose computing. Therefore, although the primary purpose of MANIFOLD is to manage communications, the same language also expresses computation in terms of communication. Thus, it is theoretically possible to replace *every* process in a MANIFOLD program by a manifold that expresses the same computation in terms of interactions among a set of finer-grained processes. This refinement can recursively be carried out all the way down to the level where each process expresses the functionality contained in a piece of hardware.

9. Acknowledgment

We wish to thank our colleagues at the Interactive Systems Department for their direct and indirect contributions to the work reported in this paper. In particular, Paul ten Hagen inspired the original concerns and the motivation for MANIFOLD by his earlier work on the Dialogue Cells, and through our numerous ongoing discussions. Kees Blom helped to refine the formal syntax for the MANIFOLD language and is presently working on the MANIFOLD compiler. Per Spilling and Dirk Soede's exercises in MANIFOLD and their ongoing contributions to the project are also acknowledged and much appreciated.

References

- [Arb90a] F. Arbab and I. Herman, "Examples in Manifold," Technical Report, Centre for Mathematics and Computer Science (CWI), No. CS-R9066, Amsterdam (1990).
- [Arb91a] F. Arbab, "Specification of Manifold," Technical Report (in preparation), Centre for Mathematics and Computer Science (CWI), Amsterdam (1991).
- [Bro89a] J. C. Browne, M. Azam, and S. Sobek, "CODE: A Unified Approach to Parallel Programming," *IEEE Software*, pp. 10-18 (July 1989).
- [Bro90a] J. C. Browne, T. Lee, and J. Werth, "Experimental Evaluation of a Reusability - Oriented Parallel Programming Environment," *IEEE Transactions on Software Engineering* **16**, pp. 111-120 (1990).
- [Car89a] N. Carriero and D. Gelernter, "Linda in Context," *Communication of the ACM* **32**, pp. 444-458 (1989).
- [Dye90a] S. Dyer, "A Dataflow Toolkit for Visualization," *IEEE Computer Graphics & Application*, pp. 60-69 (July 1990).
- [Hag90a] P. J. W. ten Hagen, I. Herman, and J. R. G. de Vries, "A Dataflow Graphics Workstation," *Computers and Graphics* **14**, pp. 83-93 (1990).
- [Her88a] J. Herath, Y. Yamaguchi, N. Saito, and T. Yuba, "Dataflow Computing Models, Languages, and Machines for Intelligence Computations," *IEEE Transactions on Software Engineering* **14**, pp. 1805-1828 (1988).
- [Hoa85a] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, New Jersey (1985).
- [Lie87a] R. van Liere and P. J. W. ten Hagen, "Introduction to Dialogue Cells," Technical Report, Centre for Mathematics and Computer Science (CWI), No. CS-R8703, Amsterdam (1987).
- [Mat88a] G. Matsumoto, "Neurons as Microcomputers," *Future Generations Computer Systems* **4**, pp. 39-51 (1988).
- [Pet86a] J. W. Peterson, R. G. Bogart, and S. W. Thomas, "The Utah Raster Toolkit," in *Proceedings of the Usenix Workshop on Graphics*, Monterey, California (November 1986).

- [Sch88a] H. J. Schouten and P. J. W. ten Hagen, "Dialogue Cell Resource Model and Basic Dialogue Cells," *Computer Graphics Forum* 7, pp. 311-322 (1988).
- [Soe91a] D. Soede, F. Arbab, I. Herman, and P. J. W. ten Hagen, "The GKS Input Model in Manifold," *Computer Graphics Forum* 10 (1991).
- [Suh90a] P. A. Suhler, J. Bitwas, K. M. Korner, and J. C. Browne, "TDFL: A Task-Level Dataflow Language," *Journal of Parallel and Distributed Computing* 9, pp. 103-115 (1990).
- [Tho91a] S. J. Thorpe, "Image Processing by the Human Visual System," in *Advances in Computer Graphics VI*, ed. G. Garcia and I. Herman, Eurographic Seminar Series, Springer Verlag, Berlin - Heidelberg - New York - Tokyo (1991).
- [Vee86a] A. H. Veen, "Dataflow Machine Architecture," *ACM Computing Surveys* 18, pp. 365-396 (1986).

Appendix A

```
//
// Compare_units_def process:
//   I/O ports:
//     a:      first unit to compare
//     b:      second unit to compare
//     output: boolean result, true iff a <= b
Compare_units_def()
    port in    a.
    port in    b.
    port out    output.
atomic.
pragma Compare_units_def internal "compare"

//
// Sort_units_def process:
//   I/O ports:
//     input:  units to sort (up to "end of file", i.e. broken port)
//     output: sorted units
Sort_units_def()
    port in    input.
    port out    output.
atomic.
pragma Sort_units_def internal "sort"

//-----//
manner next_element( smaller,smaller_data,larger,larger_data,
                    dest_smaller,dest_larger,other_port )
port    in smaller.
port    in larger.
{
    event go_on.
    start:
        do go_on.
    go_on:
        smaller_data → pass1() → output;
        getunit( smaller ) → (→ dest_smaller, → smaller_data );
        larger_data → pass1() → dest_larger;
        if( getunit(result), do go_on, other_port ).
}
```

```

        disconnected.smaller:
            larger_data → pass1() → output;
            larger → output.

        disconnected.larger:
            do finish.
    }

//
// Merge manifold:
//   I/O ports:
//     a:      first list of units
//     b:      second list of units
//     output: sorted & merged units
//     result: result of comparison
//
// uses a process "Compare" (of type "Compare_units_def")
// to compare two units; the latter returns a boolean unit
// on input port "result"
//
Merge_def()
port    in      a.
port    in      b.
port    in      result.
port    out     output.
{
    process      store_a      is      variable.
    process      store_b      is      variable.
    event        a_st_b.
    event        b_st_a.
    event        finish.
    process      Compare      is      Compare_units_def.
    permanent    Compare → result.

    start: // activate registers and reads in the two first values
            activate Compare;
            ( getunit(a) → (→ Compare.a, → store_a ),
              getunit(b) → (→ Compare.b, → store_b ) );
            if( getunit(result), do a_st_b, do b_st_a ).

//-----//

    a_st_b: // a <= b
            next_element( a,store_a,b,store_b,Compare.a,Compare.b,do b_st_a ).

    b_st_a: // a > b
            next_element( b,store_b,a,store_a,Compare.b,Compare.a,do a_st_b ).

    finish:
            deactivate Compare.
}
//
// Effective Sorter
//   I/O ports:
//     input:      units to sort
//     output:     sorted units
//   Caught events:
//     sort_full:  the number of incoming events have reached "limit"
//
// Makes a recursive call to itself if the number of the incoming units
// is more than the "limit"
// To count the incoming event, the manifold "count1" is used.
// Halts when all units are sorted and sent
//

```



```

Sort_def( limit )
port  in    input.
port  out    output.
{
    process      Sort      is      Sort_def.
    process      Sort_units is      Sort_units_def.
    process      Merge     is      Merge_def.
    process      Count     is      count1.

    start:
        activate Sort_units;
        activate Count( limit );
        input → Count → Sort_units.

//-----//

    disconnected.input: // There are no more units than limit!
        deactivate Count;
        Sort_units → output.

    death.Sort_units:
        halt.

//-----//

    death.Count:
        activate Merge;
        activate Sort(limit);
        ( Sort_units → Merge.b,
          Sort      → Merge.a,
          input     → Sort,
          Merge     → output ).

```

A Distributed Concurrent Implementation of Standard ML

David C. J. Matthews

University of Edinburgh

dcjm@lfcs.ed.ac.uk

Abstract

Standard ML is a functional programming language used extensively in universities and increasingly in industry. This paper discusses a concurrency mechanism which has been implemented in the Poly/ML implementation of Standard ML and has been used on uniprocessors and shared memory multiprocessors. It is now being implemented on a distributed network of UNIX workstations. Each of these implementations is described.

The aim of this work is to produce a distributed system that will allow large ML programs to be run on a network of processors. Although eventually such a network might be a closely-coupled network of transputers, the initial design is intended for the sort of system that many organisations have, namely personal workstations on a local network. Making use of these out of office hours will provide a substantial improvement in the computing power available.

1. Standard ML

The Standard ML language [Mil90a] was developed over the last few years as an attempt to bring together a number of strands of work on functional languages. It culminated in a language definition in which the semantics of the language are defined formally, probably the first formal definition of a full-scale language. Having a formal definition gives both implementors and users a degree of certainty that is lacking in other languages.

Standard ML has a static polymorphic type system based on type inference by unification. What this means in practice is that there is no need to put explicit type information into the program but that the compiler will automatically compute the most general type for each declaration. If it cannot find a suitable type the declaration must contain type errors and is rejected. This makes it particularly suitable for interactive program development. For programming in the large there is a module system that allows programs to be split up and developed and tested separately.

Standard ML is a functional language in the sense that functions, or more accurately closures because the environment of the function must be treated as part of it, are first-class objects. They can be passed as

arguments to other functions or returned as results. The language includes updatable variables or *references*, which can contain functions as well as simpler values such as integers.

Standard ML is being increasingly used both in the academic community and now for industrial purposes. There are a number of high quality implementations, including the Poly/ML implementation used for the work described in the rest of this paper. It is used as the initial teaching language at a number of universities and is used as the main language of implementation at LFCS in Edinburgh.

2. Poly/ML

Poly/ML [Mat89a] is an implementation of Standard ML which is currently commercially supplied and supported by Abstract Hardware Ltd. It implements the Standard ML language as closely as possible but also includes a number of extra features.

The Poly/ML implementation itself is beyond the scope of this paper, but several features of it are relevant to the provision of concurrency. As well as the concurrency primitives described in this paper and an X-windows package built using them, there is a persistent store system. The persistent store gives the ML program access to large amounts of data which are brought into store as required.

Efficient management of store is important for good performance and the design of the garbage collector can have a critical effect. All ML objects, including stacks and executable code segments, are held in the heap. The normal UNIX stack is used only by the run-time system, which is written in C. This contains the garbage collector, persistent store handler and the interface to the operating system. In all there is about 9000 lines of C in the system.

3. Concurrency

The Standard ML language includes all the features required for sequential programming, but one area that was not addressed in the language definition was the provision of any mechanism for concurrency, sometimes called parallelism.

Concurrency is needed for two quite different purposes. On the one hand there are some applications for which the non-determinacy implicit in concurrency is an integral part. A window system must be able to respond to a user pointing anywhere on the screen and clicking the mouse button, and one of the easiest ways of programming this is to have separate processes manage events associated with each window. On the other hand there has been the increasing use of multiprocessors to speed up applications but to use this a mechanism must be provided to enable programs to be split into parts that can be run in parallel.

So far work on concurrency in ML has focussed on the first of these requirements. The Dialog interface to the Lambda theorem prover, for example, has been implemented using the Poly/ML concurrency primitives. This is a graphical front end which must be able to respond to user actions while the theorem prover is still computing. Without concurrency it would be necessary for the theorem prover to periodically check to see if the user had typed or clicked. With concurrency the theorem prover can be written without having to be concerned with

events, which are only of relevance to the front-end, and yet they can all run as part of the same ML session.

The second reason for having concurrency, the speed-up to be achieved with multiprocessors, has not had so much attention, yet this area may in the long term prove to be the more profitable. This is the area currently being addressed.

4. The Poly/ML Concurrency Primitives

Before describing the various implementations it is useful to summarise the primitives themselves, and some of the reasoning behind the choice. The choice of primitives was governed by a number of factors. First of all was the need to avoid any change which would render existing Standard ML programs unusable. This precludes the addition of new syntax since existing programs which might have already used the reserved words as identifiers would no longer compile. It also requires that the new operations not allow the type system to be broken.

The obvious way of defining the primitives was as built-in functions. Their types have to be carefully chosen but, by using functions they are not being given the sort of special status that, for example, Ada tasking primitive [Bur87a] have. It is perfectly possible to experiment with alternative sets of primitives written in terms of those provided, and these will have the same status within the language as the built-in primitives. Should a derived form prove to be generally useful it could be built-in for efficiency. An example of this in the language already are the *array* operations. Arrays are not actually defined in Standard ML but can be declared using lists. This would be hopelessly inefficient for many purposes so most implementations of ML contain arrays operations as built-in functions. They are however equivalent in behaviour, though not in speed, to a reference implementation in terms of lists.

The choice was made largely on the basis of an expectation of what would be relatively simple to provide semantics for, rather than on the grounds of what might be efficiently implemented. Providing a semantics for a language without concurrency is a major piece of work. Adding the non-determinacy inherent in concurrency makes the task extremely difficult. Nevertheless a semantics has been produced [Ber91a] for a simple language using a variant of the primitives present in Poly/ML.

Processes are created using the *fork* function, which takes as its argument a function and runs it as a separate process. When the function returns the process dies. There is a related function, *choice* which provides mutual exclusion between a pair of processes. Of the two processes it creates, only one is allowed to do a communication and the other one will be killed.

The communications system chosen was essentially that of CSP [Hoa85a]. A process sends a value on a channel to another process which receives it. Both processes are blocked until the value is passed which happens atomically. The processes are then allowed to proceed. A non-blocking version of *send* could be implemented using the blocking primitives provided, by associating a buffer with a channel. In a distributed system there might well be efficiency gains if this was built in.

Channels are typed, that is the type system ensures that the values sent and received on a particular channel have the same type. When a reference or a channel are passed through a channel they are shared between

the sender and the receiver. This allows, for example, a server process to receive a request from a client which includes a channel on which to reply. The primitive functions and their types are shown in Figure 1.

```

val fork: (unit -> unit) -> unit
val choice: (unit -> unit) * (unit -> unit) -> unit
type 'a channel
val channel: unit -> 'a channel
val send: 'a * 'a channel -> unit
val receive: 'a channel -> 'a

```

Figure 1: *The Poly/ML Concurrency Primitives*

5. The Uniprocessor Implementation

The first implementation of the primitives was on a Sun3 to support multiple threaded windows. On a uniprocessor there is no advantage in speed of using concurrency, although for illustration a few example programs were written. The concurrency primitives in ML were mapped directly onto calls into the run-time system, written in C.

5.1. Process Creation

Each process runs on a separate stack, with each stack an individual heap object. In keeping with the idea of processes being light weight, the stacks are initially small, but when a process overflows the end of a stack all the stack values are copied into a new larger stack segment. Other schemes, such as allocating individual stack frames on the heap, were considered but put too high an overhead on the garbage collector. Whenever a process makes a call into the run-time system, whether concerned directly with the process mechanism or not, all the registers are saved in the stack segment. The ML processes can be interrupted resulting into traps into the run-time system, but the run-time system itself cannot be interrupted.

As well as a stack segment, each process has a process base which points to the stack segment, and is used for synchronisation. The process bases for runnable processes are linked together in a chain, and time slicing involves simply moving a pointer round the chain to select the next process to run. A periodic interrupt is used to provide the time-slicing. There is no priority scheduling, although it would be possible to use some algorithm based on, say, whether a process had used all of its time-slice or whether it had become blocked for a communication on a channel or for external input or output.

Fork creates a new stack segment and process base for the function. The stack is initialised to start executing the function and to call a kill-process function in the run-time system when it returns. The kill-process function is also called by an exception handler if an uncaught exception is raised in the process. A newly created process base is added on to the run queue and so will be run when its turn comes.

Choice is implemented using similar code to *fork* for each of two processes, but a state variable is shared between the two processes created. The state variable is pointed at by the process bases of the two processes, and examined when one of the processes attempts to do a communication. If the state variable is already set the communication does not happen and the process is killed, but if the variable was clear it is now set and the communication proceeds.

5.2. Communication

A channel is a two word updatable object and the *channel* function merely allocates store for it. Each word is used as the head of a list of processes blocked on the channel waiting to do a send or a receive. Normally only one of the lists will be non-empty, but it is possible for processes to be waiting to both receive and send on a channel if they are alternative choices. When a process attempts to send or receive on a channel which cannot immediately satisfy the request, its process base is removed from the run queue and linked onto the appropriate chain. Blocked processes use no resources other than the store required for their process bases and stacks, and this store will be garbage collected if the channel is not reachable from a runnable process and so can never be woken up.

Transferring the value is the easiest part of the communication. Since both processes are running within the same memory only one word needs to be copied, and this is held in the sender's process base until the communication is complete.

5.3. Input/Output

External input and output, such as to files or to the console, are not dealt with in the same way as communications on channels. The main concern is to ensure that a *read* or *write* does not block since that would cause the UNIX process to be suspended and so prevent any of the ML processes from running. A process that attempts to do an operation that will block is left on the run queue but set up so that it retries the operation when its time slice comes round again. The alternative, of setting the stream to generate an interrupt had unfortunate side-effects, and this scheme works satisfactorily provided the time-slices are not so long that the external events are unacceptably delayed or so short that the processor spends too much time polling devices which are not ready.

6. A Shared Memory Multiprocessor Implementation

The process primitives and the uniprocessor implementation were designed to support the window system so that there would be a satisfactory interaction with the user. Some examples of other programs were written to experiment with a process-based style of programming and also to see how well the implementation dealt with large numbers of processes, but the overhead of process creation meant that all these examples ran considerably slower than the equivalent programs without processes. On a multiprocessor it is possible to use the process mechanism to split a computation so that it will run faster, but the requirements on a process scheme designed for efficient execution on a multiprocessor might well be different to that required for the window system. The aim of the Firefly implementation was to investigate this.

The Firefly multiprocessor [Tha87a] is an experimental system developed at DEC-SRC. Each Firefly consists of 4 Microvax processors together with an IO processor connected to a shared memory. The hardware ensures that the processor caches are consistent. The Fireflies run a kernel called Topaz which provides lightweight processes called "threads" and multiplexes these onto the available processors. Getting true parallelism in ML was a matter of using these threads to run ML processes. Provided threads are not used the Firefly



can be run exactly like a uniprocessor Vax with Topaz providing UNIX emulation.

Poly/ML had originally been written for the Vax processor and so porting the sequential part of the system was simple, requiring only a few hours work. The major problem was converting the run-time system to run in a parallel environment since it was not re-entrant. In the end the solution adopted was to treat the whole run-time system as a monitor and use a single semaphore for the whole run-time system. A more careful design with monitors on smaller sections of code would almost certainly have improved performance but required considerably more drastic changes.

The run-time system can be entered either by an explicit call, for example to read a character from a file, or by means of a trap. Traps occur as a result of using an object in the persistent store, resulting in an illegal address trap, or as a result of a trap instruction being executed, indicating that a heap or stack segment is exhausted.

6.1. Threads and Processes

The initial design used one thread for each ML process, but although threads are intended to be lightweight the overheads of thread creation and manipulation are still considerable. Instead the design was changed to use only four worker threads, i.e. one per processor, each forked at the start and merely stopped if there is nothing for it to do. Each thread takes a process off the run-queue and runs it until either the process blocks or its time-slice is exhausted. This is in many ways a simple extension of the uniprocessor implementation. It is obviously necessary to reduce the number of worker threads if there are not enough processes to run, though this is complicated if processes needing to do external input or output are left on the run queue, as in the uniprocessor implementation.

As well as the worker threads there are two threads concerned with trap handling. One deals with synchronous traps such as heap segments becoming exhausted or persistent store faults, the other deals with asynchronous traps such as console interrupts and time slicing. These are needed because, although Topaz emulates the basic UNIX system, once multiple threads are used there are problems with signals.

6.2. Garbage Collection

Garbage collection was implemented as a synchronous scheme in which all the processes were stopped, the whole store was garbage collected, and the processes were then allowed to continue. It is probable that asynchronous garbage collection would have considerably improved the performance of the system, but it would have required a drastic redesign of the garbage collector. In any case there has been a considerable amount of work in this area, including work on the Fireflies themselves [EI188a].

Each worker thread has its own segment of heap in which to allocate store. This allows the threads to run independently and avoids the need for frequent calls to a central heap allocator. When a thread exhausts its segment it traps into the run-time system and is allocated a new segment. Only when the allocator is unable to satisfy the request must the store be garbage collected.

Garbage collection requires all the worker threads to stop so that there is no activity which would affect the store. When a process traps in to

the run-time system it has to acquire the run-time monitor lock, and this is used as a way of stopping activity. Each worker thread is persuaded to make a run-time system call as soon as it is in a safe state, and then it becomes blocked on the run-time system semaphore, since the semaphore is already held by the thread performing the garbage collection. As soon as all the threads have become blocked the store can be garbage collected. When garbage collection is completed the thread allocates itself the store it needs and returns from the run-time system, releasing the semaphore. This now allows one of the other threads to do its run-time system call, which then gradually releases the other threads.

7. The Development of a Distributed Implementation

Although the underlying implementation of processes on the Fireflies is substantially different from the uniprocessor version, the semantics is the same. It is important that any distributed implementation should have the same semantics so that users can move their programs from a uniprocessor to a distributed multiprocessor without seeing any difference in behaviour, except one hopes, a substantial speed-up.

The Fireflies have one serious disadvantage, namely that the individual processors were very slow, and so there is little incentive for further work on them when a four processor Firefly can be outperformed by a single uniprocessor. To continue to develop multiprocessor applications it was necessary to look for a distributed implementation.

One approach would be to design for a network of transputers, but there are a large number of personal workstations in the Laboratory which are left idle out of office hours, and it seemed sensible to design a system which would make use of them.

7.1. Data Transfer

The underlying problem with any multiprocessor system is data distribution and consistency. If programs can be split into processes that are truly independent then they can easily be run on separate processors. If they use large amounts of shared data then we have the problem of moving that data between machines. Equally if one process changes a piece of data on one machine that change must propagate to every process that has a copy. Even on the Firefly with a shared main memory this problem arises since each processor has its own cache memory and the caches have to be kept consistent by special hardware. To solve the problem on a network of UNIX workstations we have to devise a scheme for data transfer.

Before describing the system in detail it is first important to distinguish two kinds of data in ML programs. The ML type system only allows values of types *ref* or *array* to be changed, objects of all other types are given their values when they are created and cannot afterwards be changed. The Poly/ML run-time system distinguishes these values at run-time. Together with the objects representing bit-maps, channels, process bases and stacks, these are collectively known as mutable objects. All other objects, including pieces of executable code, are immutable.

As well as the difference with assignment, mutability also affects the definition of equality. When two mutable objects are compared they are considered the same if and only if they are the same object, in other



words they have the same address. In contrast immutable objects are equal if their values are equal. It is not possible to find out if two immutable objects are actually at the same address. An implementation is therefore at liberty to merge immutable objects with the same contents to reduce the storage requirements, should that be required, or to make multiple copies of immutable objects; the programmer will not be able to tell the difference.

Immutable objects therefore do not require any mechanism to ensure consistency, but that still leaves mutable objects. Fortunately, mutable objects are actually a very small proportion of the total number of objects used by ML programs. In C, by contrast, all objects can be changed. The data transfer mechanism for ML must be able to handle mutable objects but must be designed to be efficient for immutable objects.

The semantics of immutable objects allows the implementation to make a copy of an immutable object to send to another machine, without affecting the meaning of a program. There is however the problem of the size of a data structure. Immutable data structures could be transferred by walking over data structures, packaging them up, and sending them in single packages to the destination. This is the method used in RPC (remote procedure call) systems and is usually known as "marshalling". There it is assumed that the data structure will be small and there is usually some implementation-imposed limit on the overall size of the package. In a language like C there is a fairly clear correspondence between data at the user-level and machine words, but in ML that is by no means true, and a user cannot be expected to know in advance whether a particular closure, say, will fit in a given packet size. A mechanism for ML must be able to deal with arbitrarily sized data structures.

7.2. The Network Address Space

To handle this we have to introduce a network-wide address space. The address space is partitioned so that each machine has part of the space. Suppose a process on one machine wants to send a value through a channel to a process on another machine. If the value is a local address it converts the address into the network form and sends that across. The network address identifies both the machine and the object on that machine. The other machine may pass the address on or may at some point send back to the originator to get a copy of the object it refers to. The originator returns a copy of the object, possibly with more network addresses in it. If we are only concerned with immutable objects, this copy is indistinguishable from the original. For efficiency a number of objects are packaged together so as to minimise the number of calls back, but this scheme allows for arbitrary amounts of data to be sent.

This mechanism also provides several alternative ways of dealing with mutable objects. A simple solution is to keep a reference on the machine on which it originates and always send a network address. The code for assignment and dereference on the receiving machine make the appropriate calls back to do the assignment or dereference on the originator. Alternative schemes could involve making copies of references but remembering where they were so that any changes could be broadcast to them [Li90a]. Given that assignments are very infrequent compared with other languages, the apparent inefficiencies are not necessarily a problem. The trade-offs here depend very much on the way in which references are used.

Immutable objects include segments of machine code so it is perfectly possible for a process to send a function from one machine to another, and indeed one would expect this in a functional language. This would not make sense in a heterogeneous network, but if we confine ourselves to a homogeneous network it is perfectly satisfactory.

7.3. Implementation of the Network Address Space

Perhaps the simplest way to implement the network address space would be to include it within the virtual memory system, but one of the overriding principles was not to make changes to the UNIX kernel. However, there was already the basis of a suitable mechanism in Poly/ML, the persistent store. The persistent store provides access to workspaces for users and automatically pages data in from disc. This is very similar to a network where objects have to be copied from one machine to another. Persistent addresses occupy a part of the address space which is illegal for user programs so that any attempt by the program to use a persistent address to access an object results in a signal, either a bus error or a segmentation violation. The signal is handled by a routine which brings the object in to memory, changes the persistent address into the real address of the object, and retries the trapping instruction.

This scheme can be used for network addresses. Part of the persistent address space becomes the network address space, with the network space divided between the processors. When a process on one machine tries to use a network address to access an object, it traps; but this time the trap handler makes a call to the machine encoded in the address and asks for the object, retrying the object as before.

7.4. Process Creation and Communication

The eventual aim is to implement exactly the same primitives as used on the uniprocessors and the Fireflies. However as a step in the development a separate *rfork* call has been introduced which forks a process on another processor, but is otherwise the same as *fork*. In the present implementation new processes are randomly assigned to the available processors. This apparently inefficient scheme appears to work reasonably well, provided the number of processes is large, but process allocation or even process migration depending on load and/or the demands of processes on shared data structures would certainly be worth exploring.

Process communication by channels has been implemented for the distributed system. Although channels are the means by which values are transferred between processes it is their synchronisation function which is by far the most important. Indeed the transfer of data is almost incidental. They should not be thought of as "pipes". Channel objects are never moved from the machine on which they are created, instead any communications through a channel are synchronised by the machine on which the channel object resides. As in the uniprocessor scheme a blocked process is put on a chain associated with the channel if the process and the channel are on the same machine. If they are on different machines a process wishing to do a communication sends a message to the channel and receives a reply allowing it to continue or blocking it. If it is blocked it will be sent a message in the future waking it up.

8. Development and Future Work

The basic design is intended to provide a working system on which programmers can begin to develop distributed applications. It is not particularly efficient but provides a starting point for an exploration of some of the trade-offs. Inevitably the trade-offs depend on the sorts of programs written, so it is important to encourage the use of the system in order to have realistic benchmarks. There is some degree of circularity here in that programmers will tend to use features that are already efficiently implemented. This is one reason for providing as primitives those functions which have well-defined semantics rather than those which can be efficiently implemented.

There are many issues to be explored in this area. How are network addresses garbage collected? What is the best way of dealing with references? How much data should be sent when a large data structure is sent to another process? When should processes be migrated and how should the load be balanced between the machines? While these issues have been and are being explored for other languages the special characteristics of ML make this research a particularly interesting topic.

9. Related Work

The relationship of functional languages and concurrency has been explored from a number of different perspectives and is an active area of research. PFL [Hol83a] is an extension of an earlier version of ML with concurrency primitives and has been implemented on a uniprocessor. Reppy [Rep88a] has provided an alternative set of concurrency primitives for Standard ML. Facile [Gia89a] is a small language similar to ML but with concurrency built in. This work is particularly concerned with the semantics of concurrency and functional languages but has been implemented in Standard ML.

References

- [Ber91a] David Berry, Robin Milner, and David N. Turner, "A Semantics for ML Concurrency Primitives," In preparation (1991).
- [Bur87a] Alan Burns, Andrew M. Lister, and Andrew J. Wellings, *A Review of Ada Tasking*, Springer-Verlag (1987).
- [Ell88a] John R. Ellis, Kai Li, and Andrew W. Appel, "Real-time Concurrent Collection on Stock Multiprocessors," Technical Report 25, Dec Systems Research Center (1988).
- [Gia89a] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad, "Facile: A Symmetric Integration of Concurrent and Functional Programming," *International Journal of Parallel Programming*, pp. 121-160 (1989).
- [Hoa85a] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall (1985).
- [Hol83a] Sören Holmström, "PFL: A Functional Language for Parallel Programming and Its Implementation," *Chalmers/SERC Workshop on Declarative Programming, University College* (1983).

- [Li90a] Kai Li, *Private communication*, 1990.
- [Mat89a] David C.J. Matthews, "Papers on Poly/ML," Technical Report 161, Computer Laboratory, University of Cambridge (1989).
- [Mil90a] Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML*, MIT Press (1990).
- [Rep88a] John Reppy, "Synchronous Operations as First-class Values," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (1988).
- [Tha87a] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr., "Firefly: A Multiprocessor Workstation," Technical Report 23, Dec Systems Research Center (1987).

Load Balancing Survey

Dejan S. Milojevic

Milan Pjevac

Institute "Mihajlo Pupin", Beograd

eimp002@yubgss21.bitnet

Dusan Velasevic

Faculty of Electrical Engineering, Beograd

velasevic%buef78@yubgef51.bitnet

Abstract

The field of load balancing has been the target of many interesting research efforts. Quite a few important theoretical conclusions and significant implementations have been achieved. However, no widespread used or commercially available load balancing has evolved. In order to investigate this anomaly, the authors have surveyed four most promising and complete load balancing implementations. Critical hints for further research in contemporary environments are discussed. The paper also discusses standard UNIX and its impact on future load balancing implementations. A potential support of the state-of-the-art distributed operating systems is presented.

1. Introduction

The paper presents a survey of load balancing implementations on UNIX or UNIX-like platforms. The prime goal is to find out why load balancing has never achieved widespread usage. Very interesting results have been achieved, even particular implementations resulted. However, neither commercial, nor wide academic acceptance resulted. In computer industry there is a need, almost a must, for standard, open solutions. One of the examples towards this goal are three OSF requests for technology: OSF-1 operating system kernel, based on Mach technology, Distributed Computer Environment (DCE) and Distributed Management Environment (DME). Similar efforts, at more or less commercial or academical level, are being done by other developers of DOS, e.g. Chorus [Roz90a], Amoeba, [Mul86a], RHODOS [Ger90a] and others.

The presented work is part of the PhD thesis, started by the first author and guided by the third one, at the Faculty of Electrical Engineering, Belgrade, Yugoslavia. The survey should be followed by practical LB implementation on top of the Mach, to be performed at the University of Kaiserslautern, BRD, guided by Prof Nehmer, and supported by DAAD grant in 1991/92.

The authors suspect that new, state-of-the-art technology, computer architectures and system software, could be a more natural environment for load balancing. In order to help these efforts, most promising and complete load balancing implementations in the past of UNIX, or UNIX-like world, have been surveyed. The missing elements have been pointed out, and possible solutions on the state-of-the-art kernels suggested. The paper presents a survey of the Nest, Ferrari and Zhou, MOS(IX) and RHODOS implementations. Similar investigations have been performed by other authors as well [Wan85a, Cas88a, Jac89a]. Presented work resembles the EUUG conference paper [Jac89a]. There are, though, some essential differences. We eliminated all the work that did not carry explicit load balancing character, i.e. we did not consider work on Sprite [Ous88a], Charlotte [Art89a], V Kernel [Che86a] and Amoeba [Tan90a]. These are very successful projects, but mostly the aspects of task migration have been treated in them. On the contrary, authors considered the work that treats load balancing as a whole. That is, LB information management, scheduling and some kind of data transfer (remote execution or task migration). Something was essentially missing in the so far implemented load balancing projects, since no one was widely accepted. Therefore, authors compared the solutions with possible solutions on the state-of-the-art kernels, trying to find the most natural synthesis.

The second section presents a short survey of compared LB implementations. Only characteristics relevant for this survey have been mentioned. The third section presents a survey of particular criteria classified in four categories. UNIX and its impact on load balancing have been surveyed in the fourth section. Presented are characteristics that will characterise UNIX as an application on top of the DOS kernels. The fifth section presents state-of-the-art distributed kernels and their inherent characteristics that could be used for better LB implementation.

2. Description of the Selected Implementations

This section gives a short overview of the four LB implementations, stating their contributions, peculiarities, etc. A broader description is given in references and surveys. The paper considers only characteristics relevant to presented survey. **Nest** [Ezz86a, Ezz85a, Ezz86b, Agr85a, Agr87a] is a closed project that was conducted at the AT&T Bell Laboratories at New Jersey. Principal investigators were Rakesh Agrawal and Ahmed Ezzat. The environment was the network of AT&T 3B2 computers running UNIX System V. Nest LB model consisted of two modules: information policy and control policy. Three fundamental components were LB mechanism, policy and cost formulation. Nest was based on: imperfect knowledge; no apriori knowledge of the new process; fully distributed LB; adaptive and stable algorithms; remote execution; balancing over entire network.

Information was averaged and threshold based. Parameters like sampling and transmission periods were dependent upon the communication speed and the application nature. Logical pools of processors were introduced, compared to physical pools in Plan 9 [Pik90a] and Amoeba [Tan90a]. Authors implemented a flexible scheme of adding and withdrawing from the pool of LB servers. SWITCH capability directed using either local or global files systems for performance optimization. This is a nice example of LB relation to other resources – files in this case.

Ferrari and Zhou [Zho88a, Zho87a, Fer86a] is a closed research project that was conducted at the University of California, Berkeley. Principal investigators were Songnian Zhou and Prof Domenico Ferrari. The underlying environment was the network of VAXes running BSD UNIX. Load balancing problem in FZ was studied using simulation models driven by job traces collected from a production system. The authors believe that this approach may show more reliable results than analytical models or simulations driven by probability distributions. The drawback may arise in the case of different computing environment, since this approach is biased towards measured one. The authors investigated: the performances of load dependent against load independent (e.g. random) decision making and centralised against distributed and global; effects of system scale; effects of LB on individual hosts; parameter selection and adaptive LB; avoidance of instability and immobile jobs.

They have drawn the following conclusions. Overall improvements has been achieved for each of algorithms. Centralised algorithms may show up to be more appropriate than distributed ones, though costs of up to 35% of its CPU time may be paid for LB functions. There is no need for more information than it is considered. For example, if every 10sec there is a need for LB, and every 1sec information is spread, then 90% of information is wasted. This agrees with Barak's algorithms. Algorithms with little or no exchanged information showed very good performances, significantly better than those with accurate information exchange and, in particular, distributed ones. Central algorithm with information exchange showed the best performance. Algorithms with little or no information exchanged showed excellent scalability. Better predictability could also be observed. For a moderate load, the mean response time was cut by a factor of 1.5-2, and deviation by 2-4.

Following configurable parameters have been discussed: local load threshold, job threshold, the exchange period (periodical information exchange) and probe limit (non-periodic policies). It was shown that exchange period of approximately 1sec was optimal. A shorter period introduced high overhead and lower period was outdated. There was interdependency between various factors and the load: the higher the load – the higher the job threshold and the longer the exchange period. Adaptability for underlying environment did not show high benefits of switching between most promising algorithms. Non-periodic information collection policies were less susceptible to host overloading. Host overloading depended directly on system load and load information exchange period.

MOS(IX) [Bar85a, Bar86a, Bar86b, Bar89a, Alo87a, Bar85b, Bar89b] is an active projects, being taken at Hebrew University of Jerusalem, led by Professor Amnon Barak. It is one of the longest living load balancing project, started on PDP computers and UNIX version 7, to be continued on VAXes and National multicomputers. The project was based on a simple but effective and feasible scheme. It has been improved many times since. Being one of the few fully implemented LB projects and evolving over the years and computer architectures, MOS(IX) is one of the key references in the field of load balancing. In order to overcome the lack of existing distributed systems, authors introduced some concepts that have only recently been widely introduced in the state-of-the-art DOS, e.g., MOS(IX) has been split into the high and low level parts; even the same name was coined for Distributed Computing Environment as is for OSF DCE etc.

MOS(IX) LB consists of three algorithms: local load algorithm, exchange algorithm and process migration algorithm. The policy is distributed, dynamic, stable, without using apriori knowledge. It is adaptable to changes in: the overall load of the nodes; the run-time characteristics of processes and the number of nodes available in the network. The algorithm for load exchange is based on a simplified exchange of a subset of all loads in the system. The size of the kept load vector, l , is determined to be optimal for a given exchange time period, T , and the alpha, the probability that the processor X does not receive at least $\log 2l$ load vectors. A measure of the local load is the number of processes that are ready to run and waiting for CPU. This number is correlated with the instantaneous processor utilisation. In order to avoid fluctuations in load, after being measured, load is averaged over the period of time t , which is at least of the order of time required to migrate a process of an average size. Selected values are $t=1s$ (load distribution period) and $q=20ms$ (local measurements period). MOS(IX) supports dynamic process migration, including a policy for load balancing and a mechanism for process migration. All LB related functions are performed by a dedicated process that considers other processes for process migration in a round robin fashion. A process is considered for migration only after it executes for a minimal time on a local processor. This prevents short-lived processes from migration.

The following considerations are made in regards to process migration: processors' load considerations (self explanatory); communication overhead: each process tracks its communication, migration is done onto the machine where most of the communication has been performed to, in the case of local communication, an estimate of potential overhead, due to the communication is taken into account; processes that execute fork are considered more favourably since they are potential load; in the case that system table is full, local processes are supposed to migrate and local load is increased in order to prevent migrations to this machine. Process migration is based on the following actions: locally available loads are considered first; then they are modified due to communications needs and weighted; a stability value is added, proportional to the process size.

RHODOS [Gos90a, Zhu90a, Zhu90b, Zhu90c, Gos90b, Ger90a] is a recently started project at the University College, The University of New South Wales, Australia. The whole project is dedicated to the Distributed Operating Systems research. LB part of RHODOS project is led by W. Zhu and Prof Andrzej Goscinski. The underlying environment is a network of workstations running RHODOS.

The following issues are targeted for research: load estimation techniques – efficiency and feasibility; information exchange – the amount of information, the timeliness of the information and the pattern of information exchange; decision basis for migrating processes; the amount of knowledge about a migrating process; stability;

The following goals have been set: to evaluate a wide range of algorithms, provide for easy replacement of algorithms, easy tuning of system parameters (time interval for parameter collection, interval for information exchange, thresholds as the number of processes for lightly and overloaded computers etc.), provide for statistics gathering.

The authors propose two levels of resource allocation (including LB), distributed for local distributed systems and centralised for higher level of DS models (e.g. inter clusters of distributively organised systems). Load balancing policy answers the following questions: when to move

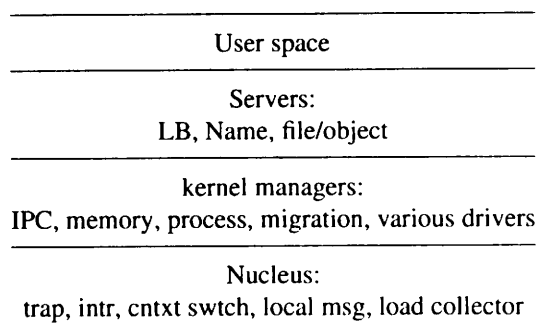


Figure 1: *Simplified RHODOS Architecture*

which process where. Parameters representing workload are divided into four groups: process- (number of ready and blocked processes), processor- (type of processor, memory size and devices), environment- (available memory, average CPU and network load etc.), and per-process-oriented (CPU time of process, waiting time, number of exchanged messages, I/O number, resource demand etc.).

LB implementation makes use of underlying RHODOS architecture. It is implemented through LB server – residing in server (user) space, migration manager – residing in kernel space and information collector – residing in nucleus space. Figure 1 presents a simplified RHODOS architecture.

3. Summarised Characteristics of Four LB Implementations

This section presents a comparison between four different LB implementations. The comparison is summarised in tables. Each table is preceded by criteria explanation and followed by a summary. Criteria have been classified in the following categories: The General Category, Load Balancing Information Management (LBIM), Load Balancing Scheduling (LBS) and Load Balancing Data Transfer (LBDT). The general category reflects the global characteristics of load balancing, common for each category. LBIM consists of information collection, advertising and negotiation components. Having more or less information on load, LBS module can make particular decisions. Once a decision is made, LBDT performs actual transfer – balancing of the load. Each category is described with particular characteristics, further presented in the following subsections. The authors assume this classification natural, since it reflects the functional characteristics of load balancing.

3.1. General Characteristics Category

This section summarises characteristics that are common to all parts of LB implementation. The following characteristics have been selected:

Distributed vs centralised authority has been used so far in either-or sense. [Gos90a] suggests the concept of simultaneous support of both authorities: distributed – better suited for small scale distributed systems and centralised – better suited to higher levels, large distributed systems. Each type of the authority has particular advantages and drawbacks. However, if applied at an appropriate level, it is possible to take advantage of both authorities, while avoiding their drawbacks. This is in particular important for new distributed system models, like Plan 9 [Pik90a], where centralised authority may be more appropriate.

General Category	Nest	Ferrari/Zhou	MOS(IX)	RHODOS
Distributed vs centralised authority	distributed	distributed	distributed & centralised	distributed & centralised
Degree of usage transparency	modified cmd syntax	transparent & user directed	transparent & user directed	transparent & user directed
Operat System modification	kernel & command	little to kernel and shell	embedded in MOS(IX)	embedded in RHODOS
Fault tolerance	considered	no	no	no
LB level	task	job	task	task

Table 1: *General Characteristics*

Usage transparency characterises particular syntax for using LB facilities. Transparency is appreciated, but allowing user to help making decisions if he wishes so, may be of particular concern.

Operating system modification is an undesirable, but usually unavoidable issue. It contradicts portability and transportability issues and should be kept to the lowest possible level. The usual modifications are regarding LB information collection, which is performed in the kernel, and particular commands that request LB algorithms execution.

Support for fault tolerance is an inherent characteristic of LB implementation. Systems support it at different levels. It is question per se, but considering it in early design phases and treating it appropriately could improve fault tolerant characteristics significantly.

Load Balancing Level is the choice of the granularity of the operating system paradigms that are being balanced. It could be job, task or thread level.

Summary. Distributed and centralised authority should be supported. Depending on the particular level of the authority and desired type of LB, centralised or distributed could be dynamically configured. If these two modules could be used transparently, it would ease the overall LB implementation and allow unique solution. Any user hints regarding LB should be accepted. New DOS paradigms support should be treated properly and incorporated into LB modules. There is much less need to modify or extend DOS in order to have necessary functionality supported. Standard microkernels support all basic paradigms, and other modules are implemented in the user space, in a much easier way. There is need for fault tolerance incorporation into the LB, aside of its inherent virtue. Appropriate levels of LB should be used for various application granularities.

3.2. Load Balancing Information Management Characteristics (LBIM)

LBIM characteristics describe the issues involved in the first phase of LB. They define the kind of LB information that is collected, how, in what time frames, the amount, the way and how much it is distributed etc. LBIM characteristics are further classified into the following elements:

Local LBIM parameters. In order to be able to do load balancing, it is necessary to find out the actual load. Therefore, some metrics need to be introduced. There is a variety of parameters that have been used

for load measurement so far. Most of them are regarding the number of processes, represented through process queues, process execution thresholds, times consumed for particular processes; static and dynamical characteristics of the computer, like total and available memory, processor speed; i/o characteristics of both machine and of particular tasks that may be candidates for balancing etc. [Zhu90c, Fer86a] present a good example for surveyed load balancing parameters.

Passed LBIM parameters. Collecting all information does not mean that everything is passed to other computers in the system. Most of the parameters regarding task characteristics are used for local scheduling and are not passed further, except, maybe, during negotiation phase. Parameters that can be passed are mostly those parameters that characterise a particular computer as a whole: number of processes and available resources. Static parameters need not be retransmitted.

Negotiation LBIM parameters. Once that a machine is selected as a target and a task as moving candidate, it may be appropriate to contact the machine and negotiate about the actual load balancing. Parameters that are considered in negotiation phase describe the particular load that is distributed – command type for remote execution and task for task migration.

Degree of information distribution. Information need not be distributed to all computers in the DOS. It was shown through few practical implementations [Bar85a, Bry81a] that even a small number of informed machines could lead to a drastical improvement. Of course, globally informed computers in network could cause no harm and are appreciated, however, a large amount of information spreading causes a bottleneck. Algorithms that use excessive information could easily become intractable.

Amount of kept information. Not all information should be kept, particularly in the large distributed systems, it is impossible to keep track of all the information, therefore, only its subset is kept.

Local information collection. Local information could be collected periodically or event driven, after the creation or deletion of the task. If collected periodically, it has a higher frequency than the advertisement, for the sake of stability.

Information advertisement. It was shown [Eag86a] that even infrequent information advertisement could yield a dramatical improvement. Therefore, it is not necessary promptly to distribute information. It is also possible to transmit only when there is a significant change in information.

Communication paths. In order to provide information to other participants in the network one may use one-to-one or one-to-many communication, depending on the desired sophistication level and actual algorithm. There is a variety of possibilities logically to connect machines in order to provide information. It may be either through simple messages, RPCs, through IPC capabilities, through a sophisticated mechanism supported by NCS, or languages providing similar capabilities (MatchMaker).

Relationship to other resources. Load balancing is a particular case of a broader problem of the resource allocation, where CPU power is a particular resource. However, even in the case of load balancing, some clients may request other resources, apart from the CPU. Also, algorithms may perform much better if they pay more attention to other

LBIM	Nest	Ferrari/Zhou	MOS(IX)	RHODOS
Local parameters	UNIX accounting	linear combination of queue lengths, ..	avrg number of ready proces	proc.,per proc. & environment
Passed parameters	averaged load	same	same	process,processor&env.
Negotiation parameters	avrg sys load and local load	no negotiation	rec. proc may refuse incom. proc	per proc
Information distribution degree	partial (to client subset)	partial or full	partial, to subset	algorithm dependent
Amount of kept Information	partial,server subset	all (distrib.) no (centr.)	part., random subset	algorithm dependent
Param. collection periodic, frequency Event driv, event	periodical 1s	periodical	periodical	periodical, event driven or on request
Advertising periodic, frequency Event driv, event	periodical 3s	periodical not if across threshold	1-60s "worm" like	periodical low and high thresholds
Communication	one to many	one to one or one to many	one to many	algorithm dependent
Relation to other resources	files	no	no	files etc.
Receiver knowledge of incoming task	no	no	no	considered

Table 2: LBIM Summarised Characteristics

process requirements, e.g. files, FPU etc., and optimise decisions according to these requirements.

Receiver knowledge of incoming task. If the receiving machine "knows" characteristics of the incoming task, load balancing may be more up-to-date and algorithm could perform better.

Summary. The following issues are still open question in LDIM part of LB. Tradeoff between costs and simplicity. The amount of information that is collected, passed and negotiated. The way LBIM parameters are advertised: periodically, event driven etc. The choice of communication paradigms. The relation to other resources, in particular to the support of other system modules (servers) that also take care of this goal (DCE, DCM etc). Using more knowledge when and if appropriate, without paying high penalties for communication and computing costs.

3.3. Load Balancing Scheduling (LBS)

LBS characterises actual algorithms that are used. This subsection will present more precisely algorithm class, type, various tradeoffs that had to be resolved in the algorithm design and particular implementation.

Algorithm class. Algorithms could be (non)cooperative, approximate vs. heuristic and (non)adaptive. Cited characteristics are due to Casavant-Kuhl taxonomy [Cas88a]. The principal reason why the

LBS	Nest	Ferrari/Zhou	MOS(IX)	RHODOS
Algorithm class	adapt. coop. approximate	cooperative approximate	cooperative approximate	various
Considered info	all	algorithm depend.	all available	algorithm depend.
Considered costs	no	CPU overhead	CPU & comm	CPU & communication
Source of LB request	threshold cross or usr directed	algorithm dependent	threshold cross + difference	various
A priori knowledge	defined servers and clients	stat. defined elig. commands	random choice of servers&tasks	considered
Overcoming uncertainty	moving window size	lowering stand deviation	aging ld vector, task residency	through negotiation
Stability	biasing	non periodic information policies and high threshold	weighted by the value proportional to process size	considered in design of TM, servers

Table 3: LBS Summarised Characteristics

authors did not pay more attention to this taxonomy is the fact that most contemporary algorithms must show a degree of cooperative, adaptive and heuristic or approximate behaviour. Still, for historical reasons, this classification of algorithms is preserved.

Amount of information that is considered in scheduling. A lot of information could be collected during the LBIM phase of LB. Not all of it is necessary for scheduling algorithm. It is the open question, how much information algorithm should be based on.

Considered costs. LB costs stem from CPU and communication, while performing each of the steps in LB (LBIM, LBS and LBDT). It is also important to note side effects of the later IPC which could overweight all the benefits of LB. Therefore, various costs should be considered while making LB decisions.

Source of LB request. Scheduling has to be triggered to make load balancing decisions. Triggering could be either event driven or periodic. Events could be, for example, arrival of new task on a client or departure of a task on a server.

Existence of a priori knowledge. Having some amount of knowledge about either the system or future load behaviour could significantly help in making decisions. User directed LB is an example of this benefice.

Overcoming uncertainty and unknown future behaviour. The best results could be achieved if the behaviour of all processes is predicted. However, this is almost impossible to achieve. Therefore, it is important to allow for possible corrections and biasing decisions. Algorithms must react appropriately in order to adapt to possible future anomalies. They must dump all unpredictable disturbances.

The way stability is achieved. One of the characteristics that LB implementation should satisfy is response to unpredictable input, in particular to the bursts of incoming load. The algorithm should react appropriately under most conditions, i.e. lead to the evening of the load across the machines. This goal is hard to achieve under stringent conditions [Cas88b].

Summary. Only Nest has adaptive algorithm. Other LB implementations consider adaptive characteristics in its research, but neither had actually implemented it. New DOS architectures allow for easier design of adaptive algorithms. There should be distinction between desired goals, and according to it particular action supported, e.g. either better mean response time, standard deviation or throughput. For example, various servers that support different LBIM or LBS could be dynamically activated. There has been very little apriori knowledge. Stability has been considered and investigated only after the design and implementation. A lot of research has been conducted in the area of LB scheduling [Sta84a, Chu80a, Lo84a, Hac89a], etc. In the practice, however, only the simplest models have been used. New architectures should offer more opportunities for exploring appropriateness of particular algorithms. For example, in overcoming uncertainty, applying AI techniques or in achieving better stability, applying control theory etc.

3.4. Load Balancing Data Transfer (LBDT)

This category summarises tradeoffs and issues in transferring the actual load from one machine to another. LBDT consists of the transfer of: command (remote execution) or task (task migration); messages, after the LBDT has been performed; results, back to the originating machine. This category could be further partitioned into the following issues:

Remote execution vs task migration is the major classification in LBDT. Both cases could be very useful in particular applications. There is no need to perform migration if a command exists on both client and server machines. However, not only commands are distributed in DS, but also user tasks. This raises the question of the task migration. Both facilities should be supported and exploited appropriately.

Transparency, in relation to the LBDT, assumes location and name transparency. Transparency is supported in most LB implementations, at more or less elegant and flexible level. It impacts message transfer, residual effects etc.

Preemptiveness is an open question of the appropriateness of active tasks migration [Eag88a, Cha86a]. In some cases it is unavoidable. This capability raises a lot of questions and tradeoffs.

Residual dependency is one of the consequences of the task migration. When a task leaves a machine, new arriving messages should be handled appropriately. If there is leftover information about the migrated task in the machine that the task migrated from, there should be some mechanisms for message forwarding, failures should be accounted for etc.

Operating system support could largely improve performance and functionality. It is much easier to rely on the basic support than artificially to implement modules in higher levels. Most state-of-the-art operating system kernels inherently support task migration through their basic primitives. LBDT is related to operating system paradigms, e.g., in Mach, task migration is related to VMM and IPC [You87a].

Summary. Task migration has been the focus of many interesting researches lately. However, no LB work has been related to these researches. Therefore, it would be necessary to reevaluate TM efforts from the LB point of view. Extensive surveys of the field of task migration could be found in [Zhu90b] and [Smi87a].

LBDT	Nest	Ferrari/Zhou	MOS(IX)	RHODOS
Remote execution vs task migration	Remote execution	Remote execution	task migration	task migration
Transparency	location & name	yes	yes	location & name
Preemptiveness	no	no	yes	yes
Residual dependency	no	no	home dependent	home dependent
Operating system support	no, extension	no, extension	yes	yes

Table 4: *LBDT Summarised Characteristics*

4. UNIX and its Impact on Load Balancing

UNIX has been a dominating operating system for many years. However, technology is going ahead. New distributed operating systems that have been recently designed, match new architectures more naturally. UNIX was originally designed for a minicomputer. There are multiprocessor implementations, and UNIX was expanded across the network. However, in its essence, UNIX still lacks support for parallel and distributed architectures. Therefore, UNIX is retained as a genius developing environment, and the never ending race with technology should be yielded to the state-of-the-art operating systems. UNIX environment will continue to dominate as an application on the state-of-the-art distributed operating systems. UNIX BSD 4.3 has been ported as a server on top of the microkernels like Mach and Chorus. Other kernels will also have ports, or a similar way of UNIX compatibility.

One of the examples of UNIX inappropriateness for distributed implementations is load balancing. Despite the huge efforts to introduce LB to computer arena, it never really left research laboratories and universities. One of the reasons is the lack of widespread distributed operating systems as well, but this also accounts to UNIX inertia for distributed solutions. UNIX has been a base for most research in the field of computer science. UNIX will remain in computer science world as an excellent environment, and will become oftenly used application. A lot of issues, like user interface, transparency, task characteristics, will remain valid. Therefore a lot of achieved results for UNIX environment like [Cab87a] will continue to characterise UNIX applications for load balancing.

5. State-of-the-art Distributed Operating Systems and their Support for LB

The field of distributed systems is growing each day. New experiences in this field help in better understanding of the needs and characteristics of load balancing. While first LB implementations have been done on the raw or modified UNIX, recent LB research is implemented on the state-of-the-art distributed kernels. The following paragraphs summarise some of the characteristics of contemporary distributed operating systems that could contribute to LB design and implementation.

State-of-the-art distributed operating systems incorporate standard features like distributed file systems, resource allocation etc, as natural

features. This obviates LB treatment of these features separately, LB mechanisms can rather rely on existing support, and make better use of it.

Contemporary DOS are built as microkernels, performing a lot of features as standard servers in the user space. This leads to better modularity allowing clean and natural design of particular components, representing functional units. Similar is valid for LB design. Implementing servers in the user space makes LB design significantly simpler and easier for development and use. It is possible to dynamically use various servers for collecting and advertising load indices.

Exchangeable scheduling policies [Bla90a, Bla88a] provide for adaptability by exchanging various scheduling algorithms, a feature that was not provided in old operating systems, where scheduler policies were embedded in the kernel.

New distributed system architectures, like Plan 9 and Amoeba, introduce new DS models, based on CPU servers. This model favours centralised LB design.

State-of-the-art microkernels support some LB functions, e.g. task migration through VMM and IPC paradigms. While for previous LB implementations it was necessary to design task migration, today it is the task to incorporate existing TM mechanisms with overall LB scheme.

There are few levels of possible balancing – job, task and thread, providing for various levels of balancing granularity: thread balancing on parallel architectures and task and job on distributed. Growing experience in parallel and distributed computing opens new approaches for advances in LB. There is also RPC level of possible load balancing.

Communication paradigms and techniques have been developed that ease LBIM: MatchMaker [Jon86a], NCS [Kon90a] contemporary versions of RPC etc.

Fault tolerance can be treated on various system levels, leading to cleaner design.

Type of LBDT is related to different models of DS. There are clustered and loosely coupled distributed systems. This raises the issue of heterogeneity. Task migration is temporarily related only to clustered DS, and remote execution to loosely coupled ones

Examples of the contemporary DOS kernels are Mach [Ras88a], Chorus [Arm90a, Roz90a], Sprite [Ous88a], V Kernel, [Che86a], RHODOS [Ger90a], etc. An example of possible LB implementation on Mach kernel (OSF/1) has been discussed in [Mil90a].

Emerging industry standards are OSF/1, Distributed Computing Environment (DCE) and Distributed Management Environment (DME). The process of implementing LB under these three software packages is underway. OSF/1, Mach based microkernel provides basic DOS support. DCE provides higher level LB implementation, and DME provides management part of it. LB implementation in these products has only been started, but a systematic approach and correct distribution of functionality and design among vertically and horizontally integrated OSF/1, DCE and DME promises a lot. Figure 2 presents possible LB design: Similar action is undertaken by another industry consortium – UI, regarding UNIX V 4.

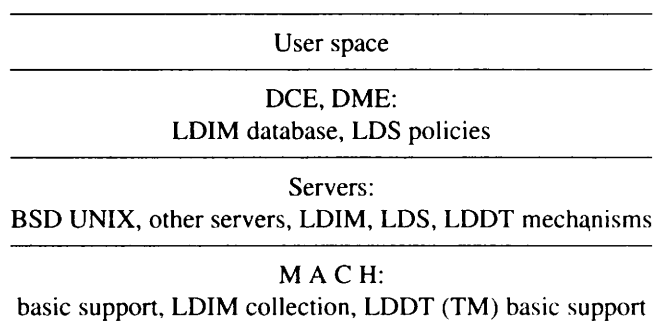


Figure 2: Possible LB design on top of the Mach

6. Conclusion

Distributed systems are continuously advancing and emerging as standards. In order to avoid later modifications and adaptation, it is necessary to predict load balancing requirements and consider them. Therefore, authors have chosen a few of the most important and complete implementations of load balancing and did a survey of the particular characteristics that could be important for the further research. Four implementations have been analysed according to selected criteria and summaries have been made. Characteristics of UNIX and its impact on future LB implementations have been discussed, as well as the opportunities that contemporary kernels provide. State-of-the-art distributed operating systems could not overcome all the drawbacks that LB has. There are some intrinsic characteristics that have prevented LB from becoming a standard and everyday tool in distributed environments. However, contemporary DOS could help in implementing LB that is modular, much easier to develop, use and administrate. Therefore, it should indirectly help to get a broader insight into the field of LB and reevaluate its tradeoffs and unknowns. It could help in applying techniques from other fields, e.g. AI, or control theory, it could contribute to improving fault tolerance, more up-to-date and broader information spreading etc.

Acknowledgements are due to the following scientists, in alphabetic order: Prof Hassan AlKhatib, Prof Amnon Barak, Prof Thomas Casavant, Ahmed Ezzat, Prof Domenico Ferrari, Prof Andzej Goscinski, C. Jacmot and Prof John Stankovic. Through the various stages of the work, they have given us advices, sent their own references, encouraged us for further work. Their help will be even more appreciated in the following stages of research, in never ending struggle with mysterious load balancing. Always attractive, so much explored and yet – so little used.

References

- [Agr85a] R. Agrawal and A. Ezzat, "Processor Sharing in NEST: A Network of Computer Workstations," *1st Intl Conf on Computer Workstations* (Nov 1985).
- [Agr87a] R. Agrawal and A. Ezzat, "Location Independent Remote Execution in NEST," *Transaction on Software Eng* **13**(8), IEEE (Aug 1987).
- [Alo87a] N. Alon, A. Barak, and U. Manber, "On Disseminating Information Reliably without Broadcasting," *Proc. 7th*

- Intl. Conf Dist Comp. Systems*, pp. 74-81, IEEE (Sep. 1987).
- [Arm90a] F. Armand, F. Herrmann, and J. Lipkis, "Multi-threaded Process in Chorus/MIX," *Proceedings of the EUUG Spring Conference*, Munich, Germany, pp. 1-13 (23-27 April 1990).
 - [Art89a] Y. Artsy and R. Finkel, "Designing a Process Migration Facility The Charlotte Experience," *Computer*, pp. 47-56, IEEE (Sep 1989).
 - [Bar85b] Amnon Barak and A. Litman, "MOS: A Multicomputer Distributed Operating System," *Software-Practice and Experience* **15**(8), pp. 725-737 (Aug 1985).
 - [Bar89a] Amnon Barak, A. Shiloh, and R. Wheeler, "Flood Prevention in the MOSIX Load Balancing-Scheme," *TCOS Newsletter* **3**(1), pp. 24-27 (1989).
 - [Bar89b] Amnon Barak, "The Evolution of the MOSIX Multicomputer UNIX System," 89-17 (Sep 1989).
 - [Bar85a] A. Barak and A. Shiloh, "A Distributed Load-Balancing Policy for a Multicomputer," *Software-Practice and Experience* **15**(9), pp. 901-913 (Sep 1985).
 - [Bar86a] A. Barak and O. Paradise, "MOS - A Load-Balancing UNIX," *Proc of the EUUG Autumn Conf*, Manchester, pp. 273-280 (Sep 1986).
 - [Bar86b] A. Barak and O. Paradise, "MOS - Scaling Up UNIX," *Proc of the USENIX Summer Conference*, pp. 414-418 (1986).
 - [Bla88a] D. Black, "Mach Processor Allocation Interface Draft of 13," CMU (Aug 1988).
 - [Bla90a] D. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *Computer* **23**, pp. 35-43, IEEE (May 1990).
 - [Bry81a] R. M. Bryant and R. A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proc Second Intl Conf on Distributed Computing Systems*, pp. 314-323, IEEE (April 1981).
 - [Cab87a] L. Cabrera, "The Influence of Workload on Load Balancing Strategies," *Winter USENIX Conf* (1987).
 - [Cas88b] Thomas Casavant and J. Kuhl, "Effects of Response and Stability on Scheduling in Distributed Computing Systems," *Trans on Soft Eng* **SE-14**(11), pp. 1578-1588, IEEE (Nov 1988).
 - [Cas88a] Thomas Casavant and J. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *Trans on Soft Eng* **SE-14**(2), pp. 141-152, IEEE (Feb 1988).
 - [Cha86a] H. Chang and M. Livny, "Distributed Scheduling under Deadline Constraints: a Comparison of Sender-Initiated and Receiver-Initiated Approaches," *Communications of the ACM* **31**(3), ACM (Mar 1986).
 - [Che86a] D. R. Cheriton, "The V Distributed System," *Communications of the ACM* **31**(3), ACM (March 1986).
 - [Chu80a] W. Chu et al, "Task Allocation in Distributed Data Processing," *Computer* **13**(11), pp. 57-69, IEEE (Nov 1980).

- [Eag86a] D. Eager, E. Lazowska, and J. Zahorjan, "Dynamic Load Sharing in Homogeneous Distributed Systems," *Trans on Soft Eng SE-12*(5), pp. 662-675, IEEE (May 1986).
- [Eag88a] D. Eager, E. Lazowska, and J. Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing," *Performance Evaluation* 6(1), pp. 63-72 (1988).
- [Ezz85a] A. Ezzat and R. Agrawal, "Making Oneself known in a Distributed World," *6th Intl Conf on Parallel Processing*, IEEE (Aug 1985).
- [Ezz86a] A. Ezzat, "Load Balancing in NEST: A Network of Workstations," *Proc of the ACM IFOMART Dallas TX*, pp. 1138-1149, ACM (Nov 1986).
- [Ezz86b] A. Ezzat, D. Bergeron, and J. Pokoski, "Task Allocation Heuristics for Distributed Computing Systems," *6th Intl Conf on Distributed Computing Systems*, IEEE (May 1986).
- [Fer86a] D. Ferrari, "A Study of Load Indices for Load Balancing Schemes," UCB/CSD 86/262 (1986).
- [Ger90a] G. W. Gerrity et al, "The RHODOS Distributed Operating System," CS90/4, University College, The University of New South Wales (6/2/90).
- [Gos90a] A. Goscinski and M. Bearman, "Resource Management in Large Distributed Systems," *Operating Systems Review* 24(4), pp. 7-25, ACM (Oct 1990).
- [Gos90b] A. Goscinski, "Resource Export and Allocation in Distributed Operating Systems," CS90/31, University College, The University of New South Wales (July 1990).
- [Hac89a] A. Hac, "A Distributed Algorithm for Performance Improvement Through File Replication File Migration and Process Migration," *Trans on Soft Eng SE-15*(11), pp. 1459-1470, IEEE (Nov 1989).
- [Jac89a] C. Jacqmot, E. Milgrom, W. Joosen, and Y. Berbers, "UNIX and Load Balancing: a Survey," *Proc EUUG Spring 1989 Conf*, pp. 1-14 (Apr 1989).
- [Jon86a] M. B. Jones and R. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," CMU-CS-97-150 (September 1986).
- [Kon90a] M. Kong et al, *Network Computing System Reference Manual*, Prentice-Hall, Englewood Cliffs, New Jersey (1987, 1990).
- [Lo84a] V. Lo, "Heuristic Algorithms for Task Assignments in Distributed Systems," *Proc 4th Intl Conf Dist Comp Systems*, pp. 30-39, IEEE (May 1984).
- [Mil90a] Dejan S. Milojicic and Dusan Velasevic, "Load Distribution - Application on top of the Mach Microkernel," *OSF Workshop: Applications on top of the Mach Microkernel* (November 1990).
- [Mul86a] S. J. Mullender and A. S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," *The Computer Journal* 29(4), pp. 289-299 (November 1986).

- [Ous88a] J. K. Ousterhout, A. R. Cherenson, F. Dougliis, M. Nelson, and B. Welch, "The Sprite Network Operating System," *Computer Magazine*, IEEE (Feb 1988).
- [Pik90a] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," *Proc UKUUG Summer 1990 Conf*, pp. 1-9 (July 1990).
- [Ras88a] R. Rashid et al, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *Tran on Comp C-37*(8), pp. 896-908, IEEE (Aug 1988).
- [Roz90a] M. Rozier et al, "Overview of the Chorus Distributed Operating System," CS/TR-90-25 (April 1990).
- [Smi87a] J. Smith, "A Survey of Process Migration Mechanisms," *Proc. 11th ACM Symposium on OS Principles*, pp. 28-40, ACM (Nov 1987).
- [Sta84a] J. Stankovic, "Simulation of the three Adaptive Decentralized Controlled Job Scheduling Algorithms," *Computer Networks*, pp. 199-217 (1984).
- [Tan90a] A. Tanenbaum, "Beyond UNIX - A True Distributed System for the 1990s," *Proc UKUUG Summer 1990 Conf*, pp. 251-259 (July 1990).
- [Wan85a] Y Wang and R. Morris, "Load Sharing in Distributed Systems," *Trans on Computers c-34*(3), pp. 204-217, IEEE (Mar 1985).
- [You87a] M. Young et al, "The Duality of Memory and Communication in the Implementation of Multiprocessor Operating System," *Proc of Symposium on Operating System Principles* (Nov 1987).
- [Zho87a] S. Zhou and D. Ferrari, "An Experimental Study of Load Balancing Performance," Report UCB/CSD 87/336 (1987).
- [Zho88a] S. Zhou and D. Ferrari, "A Trace-Driven Simulation Study of Dynamic Load Balancing," *Trans on Software Eng* 14(9), pp. 1327-1341, IEEE (Sep 1988).
- [Zhu90a] W. Zhu and A. Goscinski, "Load Balancing in RHODOS," CS90/8, University College, The University of New South Wales Technical Report (March 1990).
- [Zhu90b] W. Zhu, A. Goscinski, and G. W. Gerrity, "Process Migration in RHODOS," CS90/9, University College, The University of New South Wales (March 1990).
- [Zhu90c] W. Zhu and A. Goscinski, "The Development of the Load Balancing Server and Process Migration Manager for RHODOS," CS 90/47, University College, The University of New South Wales (May 1990).

A Public Access Interface to the OSI Directory

Paul Barker

Department of Computer Science

University College London

P.Barker@cs.ucl.ac.uk

Abstract

This paper describes a user interface to the OSI Directory. Although there are a considerable number of user interfaces already available, system administrators have complained that none of these interfaces is specifically intended for use as a "public access" interface. Such an interface must be very simple to use. This requirement leads to a set of key design goals: when there is a conflict of aims, simplicity must be favoured over functionality; ergonomic issues are of vital importance; the on-line help system must be simple but comprehensive.

The search strategy employed by the interface is also discussed in some detail. The way that the strings provided by the user are mapped onto sets of X.500 operations and matched with Directory entries is described.

The development of this interface has been funded by the PARADISE project, which in turn is funded by COSINE. PARADISE has a number of goals, including the coordination of directory service pilots. In addition, PARADISE is providing a number of central services, one of which is a public access interface to the Directory.

1. Introduction

A considerable range of OSI Directory [ISO88a] user interfaces have been produced over the last two years during the life of the directory pilot. It seems reasonable to ask the question: do we really need yet another directory user interface? This paper attempts to convince the reader that there is a requirement for a user interface designed specifically for use as a public access interface. It is suggested that such an interface should possess a certain set of characteristics not found in many of the existing interfaces. A low level of users' computer literacy must be assumed, as must total unfamiliarity with the Directory. No assumptions should be made about the sophistication of the user's terminal, or indeed about the user's ability to be able to make use of its facilities. The interface should be geared to answering common queries, such as finding someone's telephone number or email address, rather than satisfying arcane requests.

This paper describes an interface which is intended to fill this niche. The interface will henceforth be referred to as *de*, which stands for Directory Enquiries. First, the main goals of the interface's design are established and then discussed in detail. Particular emphasis is given to the key area of ergonomic issues, where the design is a combination of the author's prejudices, improvements suggested by an HCI expert, and response to the advice and suggestions given by a number of *guinea-pigs* during the course of testing.

Another area that is also considered in some detail is the mapping of the strings that the user types onto X.500 operations. An approach where a single query can lead to a number of successive searches of increasing "fuzziness" is described at some length.

The development of the interface described herein has been under the aegis of the PARADISE project [Goo91a] funded by the EEC's COSINE program. This project has a commitment to provide a mode of access to the OSI Directory to users in the research community who do not otherwise have directory software available.

2. Do we need yet Another Interface?

At the time of writing, there are at least 15 directory user interfaces (to the Quipu system alone) known to the author, and possibly more than double that in existence. Is it really possible to justify yet another interface? Some have argued that the plethora of interfaces presents an unconvincing picture to those new to directory services, indicating that the service providers don't seem sure themselves of the best tools for accessing the directory. However, there are a number of arguments in favour of having a variety of interfaces available.

First, there is the perennial trade-off between simplicity and complexity. Interfaces which offer complex facilities and a large measure of control over directory operations will inevitably be more demanding to use than interfaces which offer less control.

Second, the Directory is intended to support a wide range of queries. Following the principle of locality, a large number of queries will be for information about the local environment, which may be the organisation where the user works, or even the department in which the user works. Given that this type of query will predominate, it seems logical to offer users, *inter alia*, an interface which is tailored for these local queries.

Third, the computing environments in which people work vary greatly. Some directory users will have work-stations with bit-mapped screens, and run a windowing system with graphical user interfaces. These users may have a Directory System Agent (DSA) on their local area network. Other users may have personal computers, offering a sophisticated range of packages, but having rather restricted access to networks. A large group of users still only use character-mode terminals, and will have access to network services by use of a PAD or modem connection.

It is worth briefly considering some of the interfaces that are currently available. These interfaces are all available with the Quipu software [Kil88a], which is distributed as part of the ISODE package [Ros90a]. Almost all current directory piloting is based on the Quipu software.

dish This interface offers access to all aspects of the Directory Access Protocol. The interface is immensely versatile and can, in the style of the MH mail user interface, be used as a set

- of individual programs. This allows dish to be used to build other user interfaces using UNIX shell scripts. The interface supports modification of the DIT as well as querying. This is not an interface for the novice user!
- sd This interface is designed for character addressable terminals. The interface is essentially navigational, and a user will probably use the interface more successfully if they grasp the concept that the data is hierarchically structured. This interface has been used with some success as a public access interface, but is not simple enough for the ultra-naive user.
- pod This interface [Fin90a] makes use of the X window system to handle the display. The interface is also navigational. Pod supports modification of entries. The use of X restricts this interface to workstation users, and those who run X on other personal computers
- ufn Ufn [Kil91a] stands for user friendly naming, and is a style of directory querying and query resolution, rather than an interface *per se*. Several interfaces have this style of querying built-in. This style of querying is strongly favoured by many of those well-acquainted with the Directory. It requires knowledge of an input syntax, although this is very simple. This style of querying may well prevail as more and more users become aware of the Directory.
- fred This interface is intended to be similar in style to the *whois* program, familiar to users of the Internet. This is an advantage and a disadvantage, depending on whether one has used the *whois* program before! Fred supports user-friendly naming.
- osiw... Osiwotsits are a collection of very simple programs which are intended primarily for use for lookups within an organisation. The amount of typing required is minimal for simple, local queries, but the interface is more cumbersome for remote queries.

While some of the above interfaces could be used by novice users, none of them are specifically geared to the less sophisticated user of the Directory. This user may not be cognisant of the hierarchical nature of the Directory Information Tree (DIT); (s)he may be accessing the Directory remotely and not have access to on-line manual pages or other help with using the Directory; the user will probably be using a character mode terminal.

De has been designed principally as a public access user interface, to be used by those otherwise lacking a local access point to the directory. The interface could be configured for use in other circumstances, as it may be that some users prefer the style of querying and copious on-line help. At the time of writing it is too early to report on the success authoritatively as to the success of the design, and its suitability to the range of operational environments, although the initial feedback has been encouraging.

It is freely acknowledged that de has its limitations: it cannot, for example, be used for entry modification. However, it is hoped that some of de's strength is derived directly from its focussed approach. The intention, nevertheless, is that de will suffice for the vast majority of directory enquiries about people and the organisations they work for. If it transpires that de fails in this regard, it will be modified!

The author also regards de as being an "interface for the moment", in that it is aimed at solving today's problems. Following from this, de may well have a limited life. There are at least three reasons for this. First, de is seen in part as an interface suitable for introducing new users to the OSI Directory. As users become familiar with the services provided and the potential of the Directory, it seems likely that some users will migrate to more powerful user interfaces which allow more control over access to the Directory. Second, as more and more computer users have bit-mapped screens on their desks, the arguments for avoiding designs which rely on these capabilities will diminish. Third, de makes some assumptions about the hierarchy of the DIT. As the Directory grows, the DIT will almost certainly deepen, although to what extent it is not possible to predict at this moment.

To conclude this justification for another interface, it is worth setting out the main design goals of de, and attributing emphasis between goals as and where they conflict. The goals are as follows:

- *Ease of use.* This is best summarised by what amounts to a definition: the interface must be sufficiently easy to use for the first time user, that they are not deterred from trying to use the Directory again. This is the principal goal.
- *Useful for most queries.* It is assumed here that the predominant query will be for communications information – telephone and facsimile numbers, electronic and paper mail addresses – for people working in organisations. The interface will offer some tailoring, but will be fundamentally geared towards queries about people.
- *Terminal independence.* The interface should work with full functionality from the lowest common denominator (virtual) terminal. It should also work with barely reduced functionality even when the user is unable to provide a terminal type, or the user's terminal type is not recognised.
- *Good performance.* The interface must be able to deliver results reasonably quickly. Design decisions which prove to compromise responsiveness will be re-considered.

3. Design Characteristics of the Interface

This section considers the design of the interface. There is an underlying theme of ergonomics running through most of this section: the interface has to be simple to use. There is also a substantial discussion on de's *query engine*, or how de makes use of X.500 services to provide the results the user requires.

3.1. Screen Mode

A fundamental goal of this design is that de should be runnable from any type of terminal: it should even be runnable from a teletype! This clearly precludes the use of windowing interfaces. Full-screen designs are also considered inappropriate – experience has shown that a considerable number of users of an earlier public access service run at University College London (UCL) either found that their terminal type was not supported, or that the terminal emulation did not work correctly. One service provider told the author that many users are unfamiliar with the notion of a terminal type and, if asked to provide one, often type the name of the manufacturer of their monitor or key-

board. De tries to make it clear to such users that the service is perfectly usable even if this terminal information cannot be supplied. The more sophisticated network user, on the other hand, is often aware that they can use a variety of flavours of terminal emulation, but cannot find a type that the system they are connected to knows about. For such users, de provides the facility to list the supported types. Since so many terminal types are often supported (almost 400 under SUNOS, for example), a few well known ones are brought to the head of the list – many of the most familiar ones, such as vt100, come near the end of an alphabetically sorted list!

Given the requirement that de should function adequately without terminal type information, the design is for a scrolling interface, for both input and output. The basic style of querying is that the user is prompted for input by being asked a short series of questions. The prompts are verbose, so that it should be possible for the first timer user to query the Directory successfully. Results of queries are presented to the user a screen at a time. In the absence of any information about a user's terminal type, a terminal size of 24 rows of 80 columns is assumed. A special purpose pager has been written which supports a minimal set of commands. The pager's prompt includes information about how to get the next screen of information and how to exit the pager.

Apart from knowledge of screen dimensions, the only use which is currently made of the terminal type information is to allow the use of inverse video for prompts. It is noted that this feature has its supporters and its detractors: accordingly the use of inverse video is tailorable.

While the initial version of the interface is purely a scrolling, line-mode interface, it would be possible to produce a full-screen, character-mode interface or even a command line interface with many of de's characteristics. This is not regarded as a priority at the time of writing, but if a significant number of people request either of these styles, versions in these styles could be produced moderately quickly.

3.2. Specifying a Query

A user will specify a query by answering a short series of questions in response to some rather verbose prompts. As stated earlier the focus of the interface is on finding information about people. The user is asked to supply the following details:

1. The name of the person sought
2. Department name
3. Organisation name
4. Country name

The questions are asked in paper envelope order as this seems most natural to the author. Considerable use is made of default values: the default value is displayed as part of the prompt and accepted by entering <CR> at the prompt. Defaults help in two ways.

First, many queries made to an organisation's public access interface will be about people in the same organisation. It will thus be prudent for the system administrator to configure local values as defaults for the organisation and country names.

Second, while it is imagined that typical use of the directory will be to look up information about a single person or organisation, some support for extended querying is provided by retaining defaults from one query to the next. It should be noted that this feature is not always

helpful. It is of benefit if a user wishes to make repeated queries within various departments of a single organisation. However, if people wish to query somewhere entirely different, the defaults can get in the way. This happens in two ways. It has been observed that users tend to get "default-happy", and press <CR> to accept defaults even when they don't wish to retain existing values! Another problem is that if a subsequent query requires a null entry for a particular field (we will see cases like this shortly), where a previous query had a value entered, syntax is required to allow the elimination of the previous value. de uses the '-' character to achieve this, and informs the user within the prompt of how to clear the field. However it is arguable that <CR> may be a more natural way of clearing a field. One possible solution which has been considered is to give the user the ability to destroy all the defaults with a simple instruction, but this requires more knowledge of syntax. Feedback from users is eagerly awaited on this issue of defaulting!

A couple of examples should demonstrate the style of the interface. The reader should note the different use of defaults in the two examples. The following input should suffice for a query about the author, if using de as configured at the author's organisation:

```

Person's name, q to quit, * to list people, ? for help
:- barker
Dept name, * to list depts, <CR> to search all depts, ? for help
:- cs
Organisation name, <CR> to search 'ucl', * to list orgs, ? for help
:- <CR>
Country name, <CR> to search 'gb', * to list countries, ? for help
:- <CR>

```

The principal author of the Quipu software may be found by:

```

Person's name, q to quit, * to list people, ? for help
:- robbins
Dept name, * to list depts, <CR> to search all depts, ? for help
:- <CR>
Organisation name, <CR> to search 'ucl', * to list orgs, ? for help
:- xtel
Country name, <CR> to search 'gb', * to list countries, ? for help
:- <CR>

```

The prompting for input is seen as central to the design. The prompts are verbose to the point that users should be in little doubt about what sort of information they should be entering: this comment is subject to the proviso that the prompt is actually readable. The prompt includes as much guidance as possible about how to use the interface. The prompts may appear somewhat cluttered for the experienced user but, as de is not aimed at such users, this is a criticism which the author is prepared to live with. Apart from information on how to search for or list information in the Directory, two other key pieces of information are provided: how to quit the interface, and how to get help. De's help system is discussed in more detail later: for the moment it should be noted that the on-line help system should be sufficient to obviate manual pages and make the interface readily usable by first time queriers.

It is also possible to search for information about organisations or their departments with the same set of questions. This is achieved by the (what it is hoped is intuitive) device of the user omitting to provide input for the questions for which (s)he is not expecting an answer. An example should show that this is not as complicated as it sounds! The following query will return information about the author's department at UCL.

```

Person's name, q to quit, * to list people, ? for help
:- <CR>
Dept name, * to list depts, ? for help
:- computer science
Organisation name, <CR> to search 'ucl', * to list orgs, ? for help
:- <CR>
Country name, <CR> to search 'gb', * to list countries, ? for help
:- <CR>

```

Note above that the first <CR> is interpreted as null input, there being no default offered, whereas the latter two <CR>s accept the defaults indicated.

3.3. Search Strategy

This section considers the search strategy adopted by de to resolve a query. We will see that even simple queries may often map onto a complex set of X.500 search operations. This should not be surprising. The user will typically not provide strings which are exactly equivalent to the names in the Directory, but will often provide enough information that a unique match may be found given an intelligent search strategy. This strategy is now described.

A number of cases have to be considered: when there is a single match; when there are multiple matches; when there are no matches. For the moment we need not concern ourselves with what constitutes a match: this is discussed in the next section.

When searching for a person, the search algorithm is broadly as below:

```

for (list of countries matching $co)
  for (list of organisations matching $org)
    for (list of departments matching $ou)
      search $name

```

Given the hierarchical nature of the DIT, the searching is of necessity for countries first, then for organisations within countries, and so on. If there is a single country which matches the entered country input, a single organisation matching the country input, etc, then the behaviour of the interface requires no explanation.

If there is more than a single match for any field other than the person's name, the user is presented with a list of matches and is invited to select one from the list. The matches are numbered so the user can select an entry with very little effort. The search then continues below the selected entry. The procedure is repeated if more multiple matches are discovered, except that the user is not forced to select a single person's entry and may be showed the details for a number of entries (up to a configurable limit). There are two key features of this approach. First, a query is progressed until no match can be found for a particular category of input. The user is then asked to re-enter a name only where no match was found: on input of a new name, the query continues. Second, the user is shown the progress of the search as the query is resolved. This feedback both provides some assurance that the system is working, and also provides a user with the opportunity to abandon an operation, if it is clear that an unanticipated and unintended subtree is being searched. The control that the user can exercise to interrupt operations is discussed later.

If no matches are found for a particular type of input, the user is informed that no matches can be found, and the user is prompted for further input. The user's failure to find an entry may mean one of three things. First, the user has simply mistyped the name and retyping will result in the appropriate entry being found. Second, the user may be

specifying a name for an entry which exists in the DIT, but which does not have a name which can be matched with the name typed by the user. Third, the DIT is only sparsely populated at present, and it is quite likely that users will try to search for entries which are not yet in the Directory. For the last two reasons, it has been decided to give the user a "listing" option, to allow him/her to browse through some of the entries in the Directory. This option is invoked by typing a '*' character: it is hoped this choice is intuitive to most users, as '*' is widely used in command line shells to mean "all". It should be noted that a general listing capability will only be practicable while the DIT remains quite small. In particular, the ability to list all organisations in each country is not likely to be feasible, or even possible, for much longer. However it is felt that some sort of "listing" function will continue to be useful to allow users to examine the nature of the information in the DIT: this in itself may help users to specify their queries successfully. A few points should be noted:

- There is a fundamental assumption about the shape of the DIT. This assumption is in keeping with what currently prevails in the pilot. If practice changes, the design will need some enhancement. In particular, it is likely that an extra question asking for locality information will have to be asked before long.
- All references to *listing* above are in fact implemented by *search* operations for entries of the appropriate object class.
- Searching for a country will employ a single level search at the root of the DIT. Localities such as Europe and North America may also be searched for under the root. Searching for organisations will be by a single level search under country or locality entries. Searching for departments is initially by single level search underneath organisation entries. If single level searches fail, subtree searches are then attempted.[†] Searching for people uses subtree searches.

Given the design, there is no *prima facie* reason why the user should be restricted to searching within a single department within a single organisation within a single country. Indeed an early version of the interface offered this facility. However it has been excluded for two reasons. The first reason is technical. A search of this generality may well consume a vast amount of resources, and some sorts of limits will inevitably have to be imposed.

The second reason is political. The anxiety that many organisations and users have about making their data available will be heightened by interfaces which are able to trawl the global Directory. Since this interface is offered as the public face of the Directory, it has been decided initially at least to limit the scope of searching.

If users clamour for more powerful searching facilities, these can readily be provided by de.

3.4. Name Matching

The X.500 search operation allows complex searches to be specified by combining filters using exact, substring, approximate and other types of matching with Boolean *ands*, *ors* and *nots*. The following is typical of the sort of filter that has been employed by user interfaces to find entries in the Directory. The filter is expressed in the syntax used by the Quipu system's dish interface.

[†] The ability to specify an *n-level* search would help greatly here, and its omission from the X.500 standard is regrettable.

```
objectclass=person & (surname=$name | cn=*$name* | cn~=$name | userid=$name)
```

This can be expressed in english as "find all the people where one of the following conditions is met: the named entered exactly matches the surname, is a substring of a common name, approximately equals a common name, or exactly equals the login name attribute of a Directory entry".

However, the use of very complicated filters is not always helpful. In the above example search filter, exact matches will be returned mixed in with any substring and approximate matches. The failure to give preference to "good" matches over "looser" matches is counter-intuitive to most users. One user, mystified by the behaviour of an early interface, asked:

when searching for muller, why do I get 16 millers first?

and

typing "search tuck" produced 12 names in list before his. Not very precise. None of them were tucks!

There are several aspects to name matching which must be considered to make it more intuitive to the user. People will not usually type in strings of characters which exactly match names in the Directory. The interface must offer some sort of name matching support, and this must be available as a default. Most users will not be computer experts, deeply versed in the arcana of regular expressions, although many will know something of the use of wild-cards.

De offers both implicit and explicit wild-card support, although it is anticipated that explicit wild-carding will only be used by more expert users, and then only very occasionally.

Implicit wild-carding is provided using the following method. Rather than concocting complex search filters including various degrees of matching, de tries a sequence of searches with simpler filters. The initial searches specify exact, or very close, matching on the string entered, while subsequent searches specify increasingly looser matching. The looser matches are only tried if the closer matches fail to deliver any results. The proposed approach can entail up to four searches, although well specified searches will usually require less. This method of matching seems to provide the results expected, and reasonably quickly. An initial fear that this style of searching might be too slow has not been borne out by experience: if an initial search fails, subsequent searches of a DSA tend to be fairly fast as the DSA process and its database are paged into primary memory – this is particularly a feature of Quipu DSAs but other DSAs will probably share this characteristic to some extent.

One potential problem with this approach should be noted, although again experience has not as yet suggested that it is a practical problem. As a search is curtailed as soon as any matches are found, an unwanted exact match inhibits the search for less good matches, which might result in the required entry being found. This design decision will be reviewed in time, although it is difficult to see how to provide the user with more control over the searching in a comprehensible fashion.

An example of the search filters used by de should clarify the strategy used. First, let us consider the relatively simple filters used when searching for an organisation's entry. A sequence of up to four filters are tried in turn until some results are found.



```
objectclass=organization & organizationName=$org  
objectclass=organization & organizationName=$org*  
objectclass=organization & organizationName=*$org*  
objectclass=organization & organizationName~=$org
```

An exact match on the organisation's name is preferred over a leading substring match, is preferred over an any substring match, is preferred over an approximate match.

A more complicated example sequence of filters is used by de to locate entries for people. The format of people's common names in the Directory varies considerably: some have one or more forenames and surname; others as little as a single initial and surname. To cope with this de examines the format of the name entered. If the name contains one or more spaces, the concept of exact match is extended. De deduces that the first letter in the string entered is the first initial and that the last part of the name is a surname. Using this technique "p barker" is considered to *exactly* match against an entry with a name of "paul barker", and an entry of "paul barker" is considered to exactly match "p barker" or "paul fred barker". This technique could be built upon substantially if required. It has been suggested that the current approach is not as helpful with chinese names as with western names. Enhancements to the matching algorithms will be incorporated providing that there is no significant impact on search speed in the general case.

Explicit wild-card support is also offered as this allows the slightly more sophisticated user more control over the matching that de attempts. If explicit wild-carding is specified, only a single search is attempted in accordance with the given filter – there is no recourse to approximate matching. This explicit wild-carding is, of necessity, relatively simple because of difficulties mapping such requests onto search filters. It is not possible to map regular expressions onto the filters provided by X.500. The following forms are supported: *xxx*, xxx*, *xxx and xx*xx.

3.5. Presentation of Results

A key decision on presenting results is to display results to the user as the query progresses. Queries are resolved by first attempting to find a country with a name that matches the country name entered, then a matching organisation, etc. Queries will usually be typed as a set of strings which seem logical and convenient to the user, rather than as a set of relative distinguished names (RDNs). However, the user is shown the matched relative distinguished name. The following example (Figure 1) shows what a user might reasonably type to find the author's entry, and the layout of the results.

Some points are worth noting. The name of the country is not the RDN of the country entry. Country entry RDN's are the somewhat cryptic country codes specified in ISO 3166 [ISO81a]. The directory pilot provides an attribute "friendlyCountryName" which can be used for matching against user supplied names, but cannot be used meaningfully for display, as these names are often a set of multi-lingual alternatives. There is currently no way of selecting a name from this set of "friendly" names which can be guaranteed to be in the user's native tongue: some sort of language tagging is required. De's solution is to allow for a tailorable set of mappings between the 2-letter codes, and longer names meaningful for local users.

As stated above, names are shown to the user as the query is resolved. As soon as a match for "uk" is found, the string "United Kingdom" is

```

Person's name, q to quit, * to list people, ? for help
:- barker
Dept name, * to list depts, <CR> to search all depts, ? for help
:- cs
Organisation name, * to list orgs, ? for help
:- ucl
Country name, * to list countries, ? for help
:- uk
United Kingdom
  University College London
    Computer Science
      Adrian Barker
        electronic mail      A.Barker@uk.ac.ucl.cs
      Paul Barker
        telephoneNumber      071-380-7366
        electronic mail      P.Barker@uk.ac.ucl.cs
        favouriteDrink        16 year old lagavulin
                               guinness
        roomNumber            G21

```

Figure 1:

displayed, and so on. It is hoped that the indentation makes the results easy to interpret. It is important to realise that the names presented do not necessarily constitute all the RDN parts of the entry sought, but merely the matches on the entered strings.

The attributes shown to the user are configurable, according to the object class of the entry. There is no option to return all attributes. This is deliberate as it prevents the returning of audio and photo attributes, which are inappropriate for this type of interface and have a severe impact on performance given limited network bandwidth. The attribute keywords are configurable, and so can be made comprehensible for humans, and even language independent. Some attributes are handled specially: for example, electronic mail addresses can be displayed according to local conventions; telephone numbers can be displayed according to a local format.

If a search results in a large number of entries being found, a restricted subset of attributes is shown in the following format (Figure 2).

3.6. The Help System

Since it is anticipated that users of de will often not have recourse to manual pages, either on-line or on paper, the help system is critical. Approximately 15 help screens have been provided at the time of writing, and can be enhanced and added to freely with no need to recompile the system. A user is initially shown a brief "welcome" screen which informs the user the questions which will be asked, how to get more help, and how to get out of the interface.

```

United Kingdom
  University College London
    Computer Science

```

Got the following approximate matches. Please select one from the list by typing the number corresponding to the entry you want.

1 Geraint Jones	G.Jones@uk.ac.ucl.cs
2 Hefin Jones	H.Jones@uk.ac.ucl.cs
3 Mark Jones	071-387-7050 x3673 M.Jones@uk.ac.ucl.cs

Figure 2:



Requests for help are made by typing an initial '?' character in response to any of the prompts, optionally followed by a keyword indicating a help topic. If a single '?' is typed, the help is context sensitive: e.g. if the '?' was typed at the prompt for a person's name, help on entering a person's name is given. "???" gives "help about help", and lists all the help screens available. "?wildcards", or even "?wi", gives help on the use of wildcards in searches. Every help screen includes information on how to get further help.

Since learning by example is often an efficient way of getting started, some example queries are given and described in detail.

A convention is used throughout the help screens whereby any word which has an associated help screen appears in CAPITALS.

3.7. Resetting and Escaping the Interface

For a user to feel confident with an interface, it is essential that a number of important criteria are addressed. The following points were design goals for de.

- A user must be able to escape from de easily, at whatever point the user is currently at in the interface.
- It shouldn't be *too* easy to accidentally escape from the interface.
- A user shouldn't be faced with a plethora of questions of the sort "Are you sure that you want to ...?"
- A user must be able to correct erroneous input.
- A user must be able to abandon a query which has been initiated, but which has not yet returned results.

The above goals are achieved by a simple two-phase escape mechanism.

Phase one: Control C resets the interface such that it redisplay the first prompt and waits for input for a person's name. If a search is in progress, the search is abandoned. If the user is part way through specifying a query, that input is abandoned and the prompt for a person's name is displayed.

Phase two: A second interrupt character typed when awaiting the input of a person's name will cause the interface to exit. 'q' typed at this point will also cause de to exit.

Note that if the interface is awaiting the input of a person's name, then only a single interrupt character is required to cause the interface to exit.

3.8. Miscellany

This section discusses a number of other design issues which are considered important.

Experience has shown that the initial connection to the Directory can be rather slow relative to the time taken to perform actual operations. The paging in of processes (OSI processes can be quite large) and setting up of connections at all layers of the protocol stack can be sluggish. To circumvent this as much as possible, de binds asynchronously to the Directory, such that the binding is interleaved with the entering of the initial query. The approach seems helpful, although it seems that there is no substitute for keeping DUA and DSA processes paged into primary memory to obtain good performance.

However, the rest of de's operations are performed synchronously. There are two reasons for this. First, the synchronous model is easier to code! Second, it is not clear to the author what benefits are derived from asynchronous behaviour in most cases. If an exact match search, a substring search and an approximate match search are sent to a DSA "in parallel", the DSA has more scheduling to do, and may do much unnecessary work if a good match can be found. A measure of asynchrony may be provided by the DSA anyway as queries are passed on to other DSAs. It would certainly be useful and interesting though to thoroughly evaluate the possibilities in this area.

De does not set X.500 time or size limits. Past experience has shown that users are often frustrated by the intervention of limits. Administrative size limits imposed by DSA managers cannot, of course, be circumvented. If an administrative size limit is reached, the user is informed that there is more data than they have been shown, and told that this is a policy decision by a data administrator rather than a technical limitation. "List" operations will often be constrained in this way: the user is then invited to try and guess the name of the entry they want.

The absence of a time limit means that an operation may "hang" for some time, if a remote part of the network is unavailable. De allows the configuration of alarm times which trigger the display of a message informing the user that an operation is taking longer than expected. The user is told how to abandon the operation if they do not want to wait any longer. The delay before this warning message is shown to the user is configurable for local and remote queries.

4. Future Work

While it is tempting to add more and more features to de, the principal goal of simplicity must remain. However, it will probably be possible to add in some additional features without complicating the model. The intention is that future work will primarily be guided by user feedback. However, some areas which are already under active consideration include:

- Incorporation of user-friendly naming – de could recognise the syntax and resolve one-line queries using the ufn algorithms.
- Display more information on the progress of a query. This is desirable, but it is difficult to achieve this without affecting the display of the results.
- Sets of results are presented to the user in the order they are received from the DSA. Quipu DSAs return results lexically ordered, although results are inherently sets of data and so can sequencing can be assumed. A more sophisticated approach than that used by de is probably desirable: for example, names of people would be best presented in surname order.
- Approximate matching can sometimes deliver results which mystify users. A possible solution is to indicate to the user if the results are derived from "fuzzy" matching.
- De doesn't search for localities beneath country entries in the DIT. Currently this is only a problem with a small amount of US data, but this will not remain the case for long. However, at the moment it is not clear what sort of objects will be localities in the DIT. The situation is under review!

5. Software Availability

The software is available as part of the PARADISE package. Information on this package can be obtained from:

helpdesk@paradise.ulcc.ac.uk

De is also available as part of the ISODE software.

6. Acknowledgements

The author would like to thank Angela Sasse of the Computer Science department at University College London for her assistance with HCI aspects. In addition, Angela's early testing of the interface led to substantial and beneficial modifications of the design. She also made a valuable contribution to the structure of the help system.

Thanks are also due to Caroline Leary of the University of Sussex whose comments on de have led to changes whereby de is now more usable by ordinary mortals.

References

- [Fin90a] Andrew Findlay, Damanjit Mahl, and Stefan Nahajski, *Designing an X.500 User Interface: One Year In*, UKUUG, Cambridge, 1990.
- [Goo91a] D. Goodman, *PARADISE International Report*, University College London (May 1991).
- [ISO81a] ISO, "Codes for the representation of names of countries," ISO 3166 (1981).
- [ISO88a] ISO, *The Directory – Overview of Concepts, Models, and Service*, International Standard 9594-1, December 1988.
- [Kil88a] S. E. Kille, "The QUIPU Directory Service," *IFIP WG 6.5 Conference on Message Handling Systems and Distributed Applications*, pp. 173-186, North Holland (October 1988).
- [Kil91a] S. E. Kille, *Using the OSI Directory to achieve User Friendly Naming*, University College London (March 1991).
- [Ros90a] M. T. Rose, *The ISO Development Environment: User's Manual (version 6.0)*, Jan 1990.

Managing the International X.500 Directory Pilot

Colin J. Robbins

X-Tel Services Ltd

C.Robbins@xtel.co.uk

Abstract

For over two years now there has been an X.500 Pilot Directory Service spanning many countries. The operational management of the pilot service has been ad-hoc, coordinated by the author. In March this year the *PARADISE* project, which is part of the European *COSINE* initiative, started to manage the top level country data for the participating *COSINE* countries, and to coordinate with the North American and Australian pilots. This paper discusses the problems with the service before the *PARADISE* project, the steps that have already been taken by *PARADISE* to manage the top level DSAs and the work that will be needed in the future to manage the expanding X.500 pilot project. Extensions to X.500 needed to keep the pilot running reliably are discussed.

1. Introduction

X.500 is the joint ISO/CCITT OSI distributed Directory service [ISO88a, CCI88a]. In the current pilot the Directory is mainly used for storage and retrieval of information about people, such as telephone numbers, email addresses and photographs. The data is hierarchally structured in a Directory Information Tree (DIT) with people typically belonging to organisational units (departments), which are represented below organisations, which are in turn found below countries. Countries are at the root level. The more advanced pilot sites are also storing application entity information in the Directory.

The data is distributed amongst a number of directory system agents (DSAs), typically one DSA per organisation. A DSA holds all the local data for that organisation and knowledge of how to contact some or all of the other DSAs that make up the directory information base. At the higher levels of the DIT, data about all the organisations in a country and knowledge of the DSAs representing these organisations (level-1 data), is typically held by one DSA.¹ The data is highly replicated and most organisational DSAs will take a copy. At the root level (level-0) there is a similar DSA which holds all the data about the root level, such as data representing the countries and which DSAs to contact regarding these countries.

¹ It does not have to be a single DSA but the technology is made significantly easier if this one simplifying assumption is used.

To query the Directory, a directory user agent (DUA) uses the directory access protocol (DAP) to connect to a (usually local) DSA. DSAs inter-communicate using the directory system protocol (DSP).

Apart from the extremely brief description above, this paper assumes the reader is familiar with the basic concepts of the Directory and makes no attempt to describe X.500 itself. For the uninitiated [Ros91a] is an excellent book covering all aspects of X.500.

1.1. The Pilot

The International Pilot first started to take shape in 1988, when the first public demonstration of distributed X.500 was made at ESPRIT communications week using the *QUIPU* implementation [Kil88a, Kil91a]. *QUIPU* was developed at University College London under the ESPRIT project *INCA* with continued funding from the *Joint Network Team*. It is openly available as part of the *ISODE* package and is designed to run a wide range of UNIX systems.

In the early stages of the pilot, little management as such was needed. It was very much an experimental system. As the pilot grew, the quality of service expected grew, and in turn meant more management was required. In the early days this meant monitoring the pilot looking for problems (such as unavailable DSAs) and informing the managers, managing the higher level DSAs, and producing statistics on the use of the service.

In November 1990, the *PARADISE* project under the European *COSINE* initiative started. The major goals of the project include: provision of the infrastructure necessary to bind the European and other international pilots, such as the US white pages project, together; firmly establishing the pilot across Europe; and investigating the transition from a pilot to a full service. This paper looks at some aspects of X.500 that need extending, and some management tools that are needed to be able to run the pilot as a reliable service.

2. Central DSA

The first phase of the *PARADISE* project was to take over the running of the root DSA. Previously, this DSA had been run by the Computer Science department at University College London (UCL). This initial arrangement had become inappropriate as the pilot had expanded in scale, and the DSA was moved to the *PARADISE* machine² running at the University of London Computer Centre (ULCC). The *QUIPU* implementation was chosen as it was the only DSA available at the time that had been sufficiently proven in the pilot. As the old root DSA was seeing a reasonable level of use and was a critical part of the pilot, it had to be moved with care so as not to break the service. This was achieved by use of replication and careful knowledge management. The *PARADISE* service started to run in March this year with the 6.8 version of *QUIPU*. The UCL DSA is still running as a complete replica of the *PARADISE* DSA, and is used to provide DSAs with out of date knowledge with a reference to the new DSA.

The *PARADISE* DSA is now providing four basic services. As the use of these services increases it will become necessary to split this DSA into a number of DSAs to spread the load. Splitting the DSA into four

² A Sun 4/330 with 32MByte main memory, 60MByte swap space and 600Mbyte disk. It has direct Internet, Janet, IXI and International X.25 access.

separate DSA, one for each service, is one way of doing this. Such a split can be achieved by careful management of knowledge references. These four basic services are now considered.

2.1. Data and Knowledge Management

Perhaps the most important service, is the management of the top level data.

Countries not involved in the pilot will want to be added to the root node and some international organisations will want to be added. One of the potential problems is to decide which organisations should be added at the root and which should be added below country nodes.

It is a fundamental requirement of X.500, that names at any one level of the DIT are unique, thus organisations named at the root must have unique names. At the pilot stage we can continue in an ad-hoc fashion, adding organisations with the names they wish to use, resolving any name clashes when they happen: some useful suggestions on naming are given in [Bar91a]. However, a registration authority will be needed to decide which organisations can use which name at the root level. (Adding countries is not a problem as country entries must be named using the names specified in ISO 3166.) There is a similar problem at the country level, although most countries have a single corporate registration authority used to assigning companies unique "official" names. These can be used as the basis for a directory name. If such a body does not exist then the situation becomes more complex: for example consider the approach described in NADF-123 (North American Directory Forum) which describes a possible naming policy for the United States.

Knowledge

As well as adding the data for the entries themselves, DSA knowledge, such as the OSI address of the DSAs holding data further down the DIT, needs to be stored by the DSAs. The 1988 version of the standard did not fully define how these knowledge references should be stored, so *QUIPU* uses the mechanism described in [Kil88b] which uses the Directory itself to store the references. This has two advantages: firstly, no new database is needed to store the references; secondly, the standard X.500 operations can be used to access and update the knowledge references. DSAs will move from time to time, and so being able to have easy access to the references is vital to keep them up to date.

The alternative is to store the knowledge in local configuration files. Such an approach was used by the *Thorn* system [Kil86a], which used the ECMA TR/32 protocol [ECM85a], a forerunner of X.500, in the "Large Scale Pilot Exercise" [Kil88c]. This proved to be difficult to manage and maintain consistency.

In summary, storing the references in the Directory where they can be seen by DSA managers offers many benefits.

2.2. Parent DSA

The root DSA acts as the "parent" DSA of the whole Directory. When a DSA does not know which DSA to contact to access a particular entry it should use a superior reference: that is, it should pass the query onto a DSA that is one step closer to the root of the tree – this is called the

parent DSA. Potentially the operation can end up at the root DSA. Due to replication, this situation rarely actually occurs. If it does, it is often because the root DSA itself is needed, for example to perform a modification request on the root node.

2.3. Replication

Replication is vital to the pilot. Without replication the root and country level DSAs would soon become overloaded. It is unfortunate that replication is not defined by the 1988 version of X.500 but it is being addressed by the 1992 standards work [ISO90a].

Until the standards work is complete and the new replication mechanism is implemented, the pilot is using an interim replication protocol [Kil91b]. Any implementation joining the pilot should consider using the interim protocol. This defines how copies of all the entries at a single level of the DIT are passed between DSAs. Each DSA with a copy of an entry can answer queries authoritatively about the copied entry, and thus perform operations such as *read* and *list*. The root DSA has three roles to play in replication, which can all be split into different DSAs if the need arises:

1. Distribute copies of the level-0 data it holds a master copy of.
2. Collect copies of all the level-1 data from the country DSAs both to act as a replica and to re-distribute.
3. Distribute copies of the level-1 data to sites that can not access the country DSA directly (for example, because they are on a different networks).

2.4. Relaying

A DSA relay (described fully in [Bar89a]) is a DSA connected to two or more network services and is prepared to chain requests for a DSA on one network service to a DSA on a different network service to allow the operation to continue.

In a pure OSI environment, such a concept is not needed as there is only one global network service, but such a network is not yet available and the pilot is using both the Internet and International X.25.

A large number of the operations handled by the root DSA would appear to be caused by relay operations (it is difficult to tell exactly why the DSA is asked to perform the chain operation, but relaying would seem to be the only logical reason for doing so). Therefore, this function is a good candidate for being split off into a separate DSA, and even a set of DSAs should the need arise.

A *QUIPU* DSA currently stores a reference, in its own entry in the DIT, to a single relay DSA. If the use of relay services increases, a more flexible approach will be needed. This could enable a DSA to make a better choice of which relay DSA to use, therefore putting less load on the central relay DSA currently being used.

3. Statistics of Directory Use

An important management role is to produce statistics of what a DSA is doing, and on behalf of whom. Using *QUIPU*, such information can be recovered from logging files produced by the DSA.

The statistics can be used for several different purposes including:

- Identifying nodes frequently accessed, replication of these nodes can then be considered.
- Identify areas of the Directory tree that are often unavailable.
- Analysing patterns of use to enable better system design in the future.

Already on the *PARADISE* DSA, log file analysis has shown how the DSA is being used and which services are in need of splitting into a separate DSA. The DSA was seeing a large number of unauthenticated DSA *bind* attempts. Supporting this sort of DAP access is not really the role of this DSA (it is probably being used because it is a "well known" DSA listening on a "well known" address). DAP connections should be directed at a local DSA which will then navigate the DIT as required. The *QUIPU* software has now been modified to give the ability to disallow such *bind* connections and insist that a DUA authenticates itself, and allow a DSA manager to determine *who* is accessing the DSA.

This is a simple form of authentication policy. A more complex system to allow/disallow particular groups of users access to different services will be needed. Access to services, such as replication and relaying, will need careful control.

4. DSA Availability

If the Directory is going to be successful, clearly it must be available to DUAs whenever a request is made. Using the standard X.500 model this means **every** DSA must be available at **all** times. If **any** DSA is unavailable, this would mean a part of the Directory is unavailable. For every DSA to be available all the time is a little hopeful. In practice there will always be some temporary network, hardware, software or database errors to contend with.

The replication requirement described in Section 2.3 will go a long way to improving this. Each individual DSA is not such a vital part of the overall Directory if all its data is replicated in another DSA and the DSA knowledge is sufficient to allow a set of DSAs to be referenced. There is a problem here as the 1988 standard does not fully recognise the use of replicated data, so only a single reference should be passed between DSAs. A simple misuse of the reference mechanism as defined in [Kil88b] allows multiple references to be passed using a standard subordinate reference. In this section we are concerned with DSA availability and not data availability, so the effects of replication are not considered any further.

Early experience from the UK pilot showed that DSAs were set up, and then left by DSA managers. If the service failed it went unnoticed. As the use of X.500 increases, DSA failures will become more noticeable to users and they will hopefully alert the system managers. What is more, systems such as X.400 mail may begin to use and rely on X.500, so the system **must** be available. Thus, system administrators will need to know as soon as possible when a DSA fails to be able to correct the fault quickly. One way of being alerted is by use of a probe.

4.1. A Probe DUA

A probe DUA is a specialised DUA that closely watches a set of DSAs: for example, this might be the set of DSAs which are needed to maintain the country level service.

For each DSA being monitored, the probe will periodically make a connection attempt, and monitor whether the attempt was successful. If the "connection" fails a manager can be alerted. Periodically the results of a set of probe results from different probe sites can be collected and analysed, to give a view of how well the overall system is performing and used to identify problem areas.

What does "connect" mean in this sense? Clearly the X.500 DAP protocol must be used, as this is the service we are interested in, and so initially must involve a DAP *bind* operation. There are two main parameters in the *bind* operation. Both should be chosen with care:

1. **Access point.** To be able to probe a DSA you need to know its access point and in particular its presentation address. The presentation address can contain many alternative network (NSAP) addresses. It is possible that a network fault will cause a DSA to be available on one NSAP but not on another, so each NSAP must be probed.

We do not yet live in the perfect OSI world of one totally connected network. Using the "interim approach to Network Addresses" [Kil91c], an OSI presentation address can be used to hold addresses representing more than one network service, for example, an Internet address or a private X.25 network. The *QUIPU* DSAs in the pilot store presentation addresses using this interim approach to allow multiple networks services to be used in the pilot.

It is often the case that the host running the probe does not have access to all the same network services as the DSA being tested. The probe needs to be aware of this and not probe (thus not report failures for) networks it is not connected to. Using the interim approach it is possible for a probe to determine which network services it is connected to, which network services the remote DSA is connected to, and hence determine which network services to attempt to probe.

2. **Authentication.** The probe will need to authenticate itself to the DSA. It could use anonymous simple authentication. This would establish that DAP links between the Probe and DSA work. Alternatively, a distinguished name (DN) and no password could be supplied. This is preferable as the remote DSA, by recognising the DN of various probes, can establish whether it is being probed, and thus exclude such count the *binds* from statistics on the *real* use of the DSA.

Finally, simple authentication with username and password could be used and is the preferred mode for use in the pilot.³ This has a dual purpose. The remote DSA should check the password is valid before accepting the association. If the entry representing the DN is not held by the DSA being probed, it will need to issue a DSP *compare* operation to check the password. Indirectly this has checked that the probed DSA is able to "get out" into other parts of the Directory as well as check the availability of the DSA itself, so may be considered a better test.

³ Strong authentication is not widely available in the pilot and so is not considered in the case of a probe.

If a probe attempt fails, it is important to establish the reason for failure. It would be unpopular to alert a set of DSA managers that their DSAs are not working, when in fact it is your local network connection that has failed. There are four main categories of failure⁴ that can be detected, each possibly requiring a different manager to be informed:

1. **Network failure.**
2. **Host failure.** The network is fine, but the host is unavailable
3. **DSA failure.** The host is contactable via the network, but the DSA application process is not.
4. **Authentication failure.** The probed DSA has accepted the DAP *bind* operation but is either unable to authenticate the probe (the DSA is there and working, but may not be fully connected to the DIT), or the authentication fails (a probe error!).

4.2. More than Simply Connecting

Is using the DAP *bind* operation sufficient? As pointed out in [Ros91b] when discussing the Internet *ping* program

“Of course, *ping* is useful only for testing the connectivity from the local network device to a remote one... However *ping* cannot report on the general health of an Internet.”

Similarly, the DAP *bind* essentially shows that a protocol connection between the probe DUA and DSA is possible. One simple additional test is to monitor the time taken to achieve the *bind*, and to consider times beyond a certain threshold as indicating that something was “broken”.

Any real use of the Directory will involve keeping a connection open for a reasonable period of time and transferring some data. Some early experience with the probe in the X.500 pilot, has shown that a DSA which may be considered available by the “bind test”, often fails (usually due to a timeout) when a *read* operation is attempted. It is suggested a probe might occasionally try a set of DAP operations, such as reading an entry on the probed DSA, to make sure the connection is able to pass sufficient data rapidly enough to be considered a service. However doing this on every invocation of the probe would probably create too much of an overhead on the system.

Other factors to consider when probing include:

- The probe needs to be run from a number of different sites, so that “network islands” can be identified.
- Probing too often will cause the DSA network to become overloaded with probe traffic, but not often enough may miss vital DSA absences. To cut down on some of the probe traffic, use of passive probing could be considered. If the probe worked in conjunction with a DSA, the probe would not have to test a connection the DSA knows to be working as it has recently used it. This also has the advantage of being more than a mere *bind* test. In fact a *QUIPU* DSA keeps a record of such information for internal use. See [Bar89a] for details.
- Probing should not be done at fixed times. A probe being used by various sites in the *PARADISE* pilot [Tit91a] comes with an example UNIX *cron* script. This arranges for a probe to be run on the hour, every other hour throughout the day. Thus DSAs

⁴ It is possible for there to be a failure at other levels such a session or presentation but the author believes these are much more rare than the other categories given, and will in the most part be caused by protocol errors which would not occur in a service environment.

being probed from many sites suddenly see a mass of connections coming in. Most DSAs will have a limited number of connections they can accept at any one time, and so some probe calls will be rejected – the probe could effectively be causing denial of the service it is trying to help keep available!

5. DIT Counting

How big is the DIT? This is one of the questions frequently asked when discussing the pilot. Because of the distributed nature of the Directory, this is not easy to answer. The first real question that needs to be answered is: what do we mean by the DIT size?

- Is it the total number of entries held anywhere in the DIT?
- Is it the total number of entries visible to a user at a given time?
- Are entries that can not be accessed due to the access control policy of the directory management domain (DMD) counted?
- Do you count entries that are held in a DSA that has been unavailable for an extended period?

The answers to these question are implicit in the counting mechanism used, so need to be considered when deciding how the Directory is to be counted. Unavailable data can not be counted by a mechanism that relies on accessing it.

There may be many different reasons for representing an entry in the DIT. Some entries may be provided as service for users, but others may be experimental. The paper [Kil91d] defines a set of attributes that can be used to define the level of service a subtree in the DIT is providing and can be used as a basis for deciding whether to count the subtree. When analysing the results of probing or DIT counting it may be appropriate to only analyse the “service” DIT.

There are two basic methods that can be used for counting. First, for any particular node, each subtree of the node can be counted (somehow), and added together with the number of leaf entries, to provide a subtree total for the node itself. Secondly, the number of entries held by each DSA can be counted (somehow) and (somehow) merged to establish a subtree or DIT total.

5.1. Counting by Subtree

To count a subtree, *all* you need to do is recursively add up the subtree size of the subtree(s) referenced by the sibling nodes, a leaf node has a subtree size of one. A specialised DUA could attempt to walk the DIT recursing down each subtree, counting each node as it is passed. However, it is very likely that such a DUA would be unable to access every node in some subtrees as the DMDs may impose some administrative limits of the number of entries returned by *search* or *list* operations.⁵ In any case, such a DUA would not scale beyond the DIT piloting stage. However, this approach does have the advantage of counting the “visible” DIT.

The real problem is trying to do the count of the DIT, all in one go, from one place. If the problem could be spilt into smaller components it may well be possible.

⁵ An aggressive DUA might try to get around this by repeatedly refining the search criteria to get around the limits, but such a DUA is not seriously considered as the attempt to break the limit it is likely to be detected and rejected by an “advanced” DSA.

Subtree Totaling

One such method would involve adding special "subtreeSize" attributes⁶ to all non-leaf entries. This would indicate the number of entries held in the subtree. The counting DUA would trust this value and then use it instead of proceeding down the DIT. This value could be maintained either directly by the DSA, or by the DMD manager, (although there is the possibility of a subtree faking its size). If a DSA knew about these attributes for each level of the DIT it held, it could add up the sizes of all the sibling subtrees (by looking at the attributes) and calculate a size for the subtree it holds. This could then be propagated up the DIT automatically (the mechanisms to do this are not necessarily easy, as the DSA may not hold the entry for the subtree itself!). A totally automatic system would be desirable, so that a DAP *read* of the "subtreeSize" attribute or the "root" node would give the DIT size!

Any subtrees that do not contain these attributes could be ignored and counted as zero.

5.2. Counting by DSA

There are two main pieces of information needed for this approach: how many entries does a DSA hold; and which subtree(s) are held?

If a *QUIPU* DSA receives a DAP *read* of a specific attribute, with a specific setting of the X.500 service controls (as defined in [Kil89a]), then the DSA will return a single attribute, with a string value describing the number of entries it holds. If a DSA does not support such a feature, then the figure could, for example, be placed in the DSAs entry in the DIT by the DSA manager and then read by a DAP *read* operation.

To work out which subtree(s) the DSA holds is more complex. Again this can potentially be done by looking at the DSA entry in the DIT. An alternative is to understand how the knowledge references of the DSAs are managed and to work out which subtrees a DSA is supposed to manage. Unfortunately storage of knowledge references in the Directory were not fully standardised in the 1988 standards work, so each implementation might (and probably will) represent them differently. In *QUIPU* each non-leaf node in the DIT has a knowledge reference in its own entry, so you can establish which DSA is responsible for the next level in the DIT (The procedure for doing this is defined fully in [Kil88b]).

Of course this counting mechanism does not always work correctly. The DIT and DSAs used to hold it are not always structured in such a neat and tidy way. Suppose you wish to count the size of the "X-Tel" subtree below GB. You may find two DSAs in the knowledge, and thus you count the entries in these two DSAs: let's say this gives a total of n . However, one of these DSAs also holds data on the locality of "Nottingham" below GB. So the best you can do is say is:

The subtrees of X-Tel and Nottingham contain n entries between them.

This is a simple case. In the real DIT there are some much more complex examples. This is the major failing of this counting approach.

⁶ In reality such an attribute might contain more information than just an integer size, such as the type of information held.

5.3. What is Needed?

As we have seen neither of the two approaches are easy. Both need the full cooperation of all (or at the very least a significant number of) DSAs or DMDs involved in the subtree.

Until this is achieved with some official mandate, counting will have to continue with an ad-hoc method such as that used to produce [Goo91a]. This was based on the DSA counting method discussed, as it is the only way possible with the current software in use in the pilot. As the number of implementations involved in the pilot grows, this will become much harder to achieve.

There is one final problem! An organisation may consider that the number of entries held in its database to be sensitive data (as it may reveal the number of employees for example) and may not be prepared to allow the DSA to reveal such information, so it can not be counted.

As the DIT (dynamically) grows, it is quite possible that a true count is not possible, and only an estimate can be achieved. Perhaps the best you can realistically aim for is a count of the number of organisations or other higher level entities (such as localities) represented in the DIT.

6. Only the Beginning...

The mechanisms described in this paper so far are only the beginning. Up until recently, effort has been on getting the services running and the protocols in place and sufficiently robust. As the pilot grows and the number of different implementations involved increase, there will become more to manage. It is important to get the mechanisms in place now. Some such mechanisms have been discussed here. As time goes on, it may be necessary to remove the "one master DSA per level" simplification being used in the pilot and by the *QUIPU* implementation. When this happens, issues such as counting and effective knowledge management will become very much more complex. There is plenty more work to do!

7. References

- [Bar89a] P. Barker and C. J. Robbins, "Directory Navigation in the QUIPU X.500 System," pp. 235-244 in *UNIX & Connectivity*, National UNIX Systems User Group, The Netherlands (November 1989).
- [Bar91a] P. Barker and S. E. Kille, "Naming Guidelines for Directory Pilots," INTERNET-DRAFT: draft-ietf-osids-dirpilots-00, University College, London (February 1991).
- [CCI88a] CCITT, *The Directory – Overview of Concepts, Models, and Service*, Recommendation X.500, December 1988.
- [ECM85a] ECMA, "OSI Directory Access Service and Protocol," ECMA TR/32, ECMA TC 23 (December 1985).
- [Goo91a] D. Goodman, *PARADISE International Report*, University College, London (May 1991).
- [ISO88a] ISO, *The Directory – Overview of Concepts, Models, and Service*, International Standard 9594-1, December 1988.

- [ISO90a] ISO, *The Directory – Part 2: Information Framework – Addendum 3: Replication*, ISO 9594-2 PDAM 3, December 1990.
- [Kil86a] S. E. Kille, "THORN (The Obviously Required Nameserver)," *IES News*(5.), pp. 11-14, Esprit (August 1986).
- [Kil88c] S. E. Kille, "The THORN Large Scale Pilot Exercise," *Computer Networks and ISDN Systems* 16(1), pp. 143-145, North Holland (September 1988).
- [Kil88a] S. E. Kille, "The QUIPU Directory Service," *IFIP WG 6.5 Conference on Message Handling Systems and Distributed Applications*, pp. 173-186, North Holland (October 1988).
- [Kil88b] S. E. Kille and C. J. Robbins, "Distributed Operations in QUIPU," *Esprit Communications Week* (November 1988).
- [Kil89a] S. E. Kille, C. J. Robbins, and A. Turland, *The ISO Development Environment: User's Manual*, Volume 5: QUIPU, April 1989.
- [Kil91b] S. E. Kille, "Piloting Directory Services: A transition to full service," *The Electronic Directories Conference*, Online (April 1991).
- [Kil91c] S. E. Kille, "An Interim Approach to use of Network Addresses," University College, London. Research Note RN/89/13 (January 1991).
- [Kil91d] S. E. Kille, "Handling QOS (Quality of service) in the Directory," INTERNET-DRAFT: draft-ietf-osids-qos-00, University College, London (February 1991).
- [Kil91a] S. E. Kille, *Implementing X.400 and X.500: The PP and QUIPU Systems*, Artech House (1991). ISBN 0-89006-564-0
- [Ros91b] Marshall T. Rose, *The Simple Book: An introduction to Management of TCP/IP-based internets*, Prentice-Hall (1991). ISBN 0-13-812611-9
- [Ros91a] Marshall T. Rose, *The Little Black Book: Mail Bonding with OSI Directory Services*, Prentice-Hall (1991). ISBN 0-13-683210-5
- [Tit91a] Steve Titcombe, *DSA Monitoring*, University College, London (May 1991).

Appendix

For more information on the PARADISE project and obtaining the QUIPU software (which is part of the openly available ISODE package – commercially supported by X-Tel) used throughout the pilot contact:

helpdesk@paradise.ulcc.ac.uk

or X-Tel Services Ltd.

A Design Overview of XLookUp

Damanjit Mahl

*Manufacturing & Engineering Systems,
Brunel University, UK*

Damanjit.Mahl@brunel.ac.uk

Abstract

XLookUp is an interface to the X.500 directory service which runs under X Windows. XLookUp is intended to cater for many modes of directory usage: casual look up of addresses, administration of local data, as a public access service. Thus the interface needs to combine a high level of functionality with simplicity and user friendliness. It is still under development at the present time (June 1991), although a release is expected before August 1991. This paper offers an overview of the current design and presents some examples which give an idea of the appearance of the interface.

1. Background

User interfaces to the OSI Directory are many and varied. Those provided with ISODE (including some written by the author) do not possess the user friendliness or functionality required to convince users that the Directory is a worthwhile service. These interfaces represent early efforts, and suffer from a number of problems:

- Do little to hide the hierarchical structure of the Directory Information Tree (DIT) from the user.
- Require too much user participation in the search process
- Do not provide adequate and friendly facilities to update Directory information.

Many of these problems were caused by a lack of asynchronous access to the directory, thus preventing complex queries to be made without being overly time consuming, and by a lack of experience in presenting X.500 to the user. Asynchronous DUA access has recently become a reality in ISODE, experience has been gained from earlier efforts. Added to the increasing amount of data being made available in current pilot schemes and continuing improvements in reliability, this means that DUAs presenting usable and worthwhile services should start to appear.

XLookUp is the fourth in an evolutionary series of X based DUAs. Two of the preceding three were released as part of ISODE. One of these prototypes is still in wide use and both have resulted in ongoing feedback on DUA interface structure and functionality. This feedback

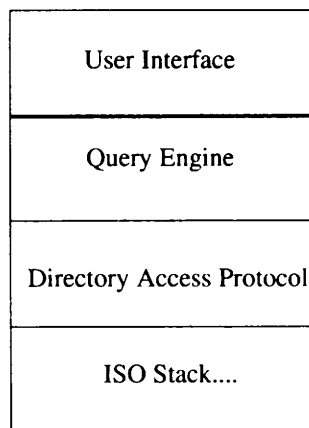


Figure 1: *XLookUp internal structure*

has been incorporated into all aspects of the current design, along with the ability to perform directory queries on an asynchronous basis.

This paper gives an overview of XLookUp, focusing on two major aspects of the design:

- The directory query formulator (referred to as the *query engine*)
- Configurability of the user interface

2. The Query Engine

Figure 1 shows the internal structure of XLookUp. The query engine provides directory services at a layer above the Directory Access Protocol (DAP) layer.

The services provided by the query engine are:

- Two distinct search strategies
- Read entry
- Complete information updating facilities, i.e. modify name, modify entry, add and delete entry.

At present browsing is discouraged, and hence no interface to the X.500 list operation is provided, though this may change given significant negative feedback. The idea being that a user must know what she or he is looking for, and has some idea of where the required information might be.

The most important of the services described are the search facilities. These attempt to find an entry given some set or sequence of values supplied by the user. The queries formulated often require many individual X.500 requests to be made. The new asynchronous approach allows such queries to be speeded up, because multiple requests can be made simultaneously, and allows the user to continue with other work in the application which is not held up whilst the query is being performed. This makes the strategies employed feasible, where they weren't in the earlier synchronous mode of usage.

This section is devoted to the directory search strategies employed by the query engine: user friendly naming (UFN), and form based searching.

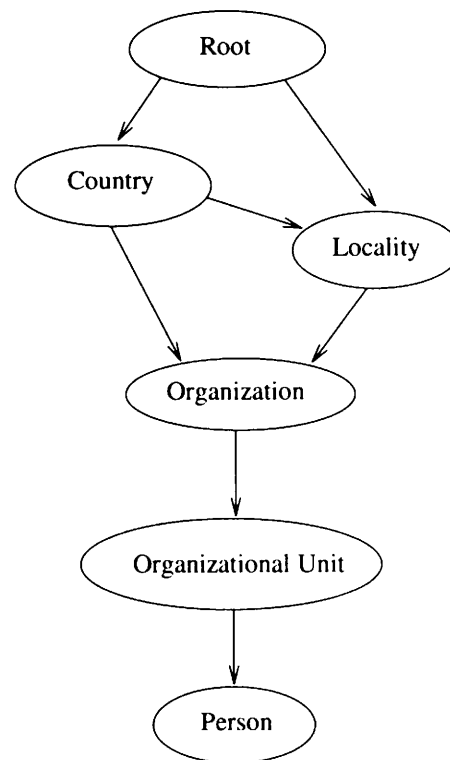


Figure 2: Assumed DIT type hierarchy

2.1. User Friendly Naming

This is derived from the user friendly naming scheme proposed by Kille [Kil91a]. This method can be classified as an *untyped and ordered* naming scheme, as defined in a UK Proposal to the ISO/CCITT Directory Group on "New Name Forms" [Kil89a]. This means that the naming information comprises an ordered sequence of name components each of which is some value which matches an attribute in the named entry.

The ordering in this case is based on the reverse of the Relative Distinguished Name (RDN) sequence in the DN of an entry. For example the UFN "damanjit mahl, manufacturing & engineering systems, brunel, gb" resolves to the author's DN

```
qc=GB@o=Brunei University@ou=Manufacturing &
Engineering Systems@cn=Mr D S Mahl
```

A DIT hierarchy, shown in Figure 2, is assumed in order to resolve the types of name components.

As well as catering for the exact case, as illustrated in the example showing the author's directory name, UFN also attempts to resolve partial, inexact or ambiguous names. These are referred to as purported names.

Purported names can vary from the exact case in a number of ways:

- **Abbreviation** The most significant name components can be excluded, e.g. "Steve Kille, Computer Science" is missing components "University College London, GB".

<i>UFN Environment</i>
1 Name Component
Manufacturing & Engineering Systems, Brunel University, GB Brunel University, GB GB (root)
2 Name Components
GB Brunel University, GB (root)
3+ Name Components
(root) GB Brunel University, GB

Table 1: *UFN environment for a personal DUA*

- **Component Omission** An intermediate name component may be omitted, e.g. "Steve Kille, University College London, GB" where the component "Computer Science" has been omitted.
- **Approximation** Approximate values of name components can be supplied, e.g. "a findl" matches "Andrew Findlay".

The following sections describe elements of the UFN algorithm which are used to resolve purported names.

2.1.1. Environment

UFN employs a set of lists of directory names which form a local environment. The environment is defined in terms of the number of name components in a given purported name. The number of components then corresponds to an appropriate list of directory names which can be used as a starting point for name resolution. For example the local environment for a purported name with one component might initially consist of a department, i.e. the given name is at first assumed to be a user in a department, if this fails the next element of the environment would be tried, and so and so forth. Table 1 shows an example environment specification, in this case the environment used by the author.

2.1.2. Name Resolution

Name components are matched sequentially. If a single DN is matched against an intermediate or initial name component the name resolution continues as if a full DN had been supplied. If one or more DNs are matched against an intermediate name component these are all explored in order to attempt to resolve the ambiguity.

2.1.3. Use of Subtree Searching

In the general case X.500 searches are performed as single level DIT operations. When a match fails, if the previous name component has matched against an entry of type *organization* then a subtree search is

attempted. This is reasonable in most cases, but may fail when attempted in larger organizations.

2.2. Form Based Searching

This, in contrast to UFN, is a *typed and unordered* approach to searching. Here values are supplied in association with specified entry types. Ordering is resolved by assuming a DIT hierarchy (a typical hierarchy is shown in Figure 2). The user is asked to fill in a form showing a list of context types and a target type. Some fields may be omitted, though this is constrained. Situation where value omission is not reasonable are:

- **Intermediate types** Although ordering is not apparent to the user it is an inherent part of DIT structure. Thus the search fails when, for example, country, department and person values are supplied, and a value for organization is omitted. As an alternative this particular set of values may be regarded as something that amounts to a browsing request, i.e. list matching persons in said departments in an arbitrary list of organizations which are present in the said country. It is not clear whether this kind of browsing should be allowed. The author's feeling is that it shouldn't as it represents a rather dubious method of looking up information together with being an unnecessary load on directory services.
- **Target type** A missing target value means that a search cannot be completed! An identified target type is necessary as the entry types are unordered, thus making it difficult to choose a particular value as the objective. The current on screen representation of this is shown in Figure 3.

2.2.1. Environment

Inexact information is handled similarly to UFN, the main difference being the way in which the local environment is handled. Here defaults are associated with nodes in the assumed DIT type hierarchy. A typical set of defaults is shown in Figure 4.

A particular set of defaults may in some cases be given precedence over others. This is denoted by thicker lines in Figure 4, hence, given the target type organizationalUnit, when no values for country, organization or locality are given, the algorithm chooses the defaults associ-

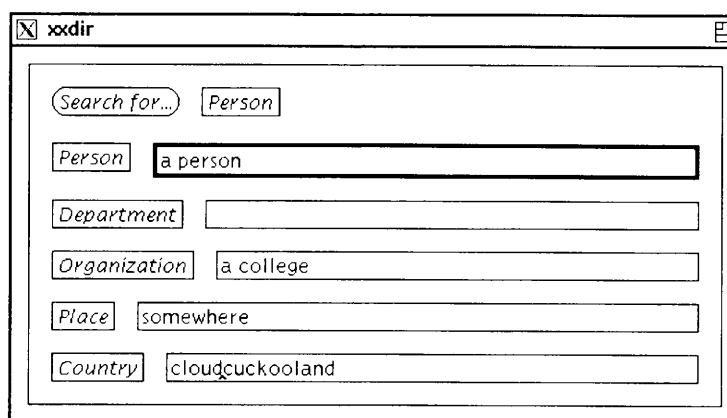


Figure 3: Form based searching

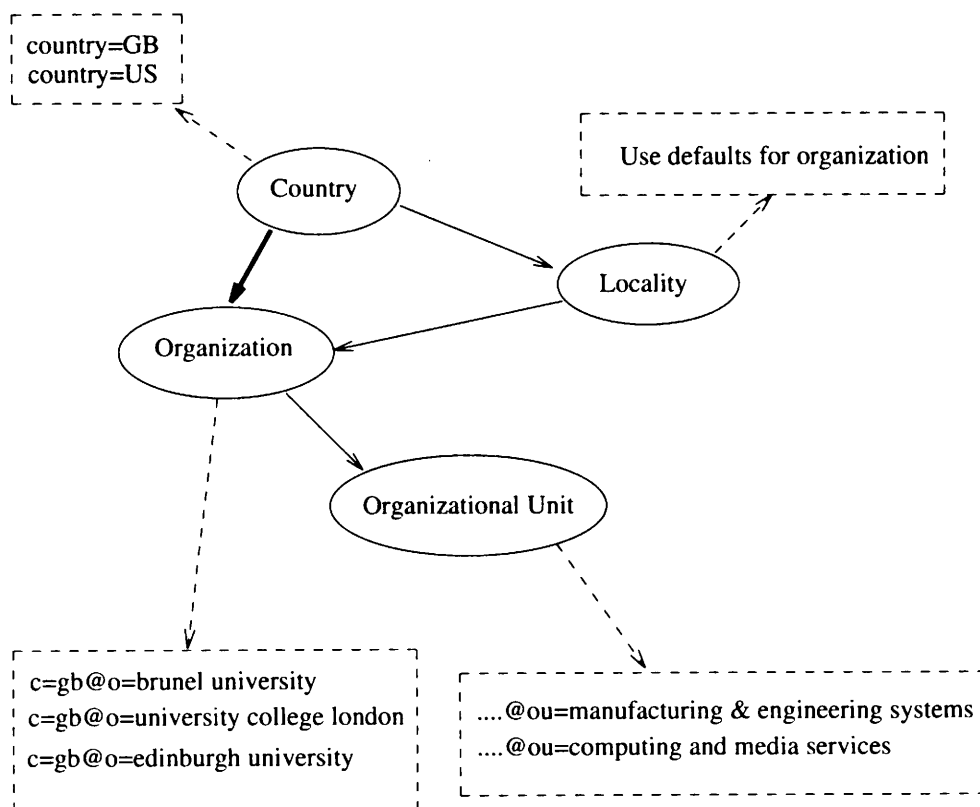


Figure 4: Defaults for form based search

ated with the organization type because the path from country to organization is given precedence over the path going via locality.

Notice that defaults attached to a particular node do not have to be of the same type as the node itself; the defaults are just a way of informing a DUA of the best place(s) to start a search from.

3. Configurability of the User Interface

The X.500 Directory is a large and rich store of information; there are many types of directory entities e.g. people, applications, mailing lists etc., each entry can contain a potentially large number of information types (attributes of the entry) e.g. names, e-mail addresses, machine addresses, phone numbers, document types etc.. Many different kinds of information imply many different usages of the directory. This begins at low end usage, such as casual address look up, and ends at the more complex needs of a site administrator. This presents a problem when also considering that user friendliness is of great importance.

The solution taken in XLookUp is to make the front end and associated functionality as configurable as possible. The content, functional and visual, of any XLookUp window is definable. This has the advantage of allowing the interface to be tailored to personal tastes. More importantly, this renders means of allowing DUAs to be built that are tailored to specific tasks or needs and have the minimum of functional redundancy apparent in the on screen interface.

3.1. The Kit of Parts

The approach taken allows windows within the interface to be constructed as groups of defined window objects (referred to as primitives from now on). This forms a *kit of parts* from which interfaces can be built.

The current list of defined primitives is:

- **Status Bar** – Current status and or error display
- **Title Bar** – Contains an arbitrary label, usually the title of a window
- **Read Display** – Display for directory entry information
- **Lookup** – Form based search dialogue
- **Moveto** – Directory entry naming dialogue
- **Entry List** – List box containing lists of entries
- **Help** – Window displaying selected help texts
- **Modify Entry** – Form based entry modification dialogue
- **Add Entry** – Form based entry addition dialogue
- **Rename Entry** – Entry renaming dialogue
- **Button** – Button attached to a defined command
- **Error Display** – A viewer for errors resulting from directory operations

Each primitive may have one or many attached *actions*, e.g. the action attached to the *Moveto* primitive is the Query Engine operation which performs a UFN match. Each primitive also has a number of associated resources: height, width, bitmap (in the case of buttons and title bars only), initial sensitivity to user events, information used when a window is resized and a name used to identify particular instantiations of a primitive. This is additional to the standard application resources used by the X Toolkit

3.2. Constructing Windows

Windows can be constructed as nested groups of horizontally or vertically arranged sequences of primitives. The sample configuration in Program Listing 1 defines a window called *startup* shown in Figure 5.

The “P.” (as in P.Button) convention is used to distinguish primitive types from named instances. The resources set here are the visible labels, dimensions, primitive instance name, and interface commands (these are explained in the following section). Note that labels, amongst other resources, can be set in the XLookUp configuration file or in a standard X resource file by using configured instance names.

3.3. Defining Command Structure

Any primitive that can be activated by the user, e.g. a button, can have a user defined command associated with them. Each command is composed of a sequence of atomic operations. These are:

- **Quit Application** – Quit from the application
- **Make Sensitive** – Make a named primitive or primitive type sensitive to user events

```

LAYOUT : startup : VERTICAL
{
    P.TitleBar      ((LABEL "XLookUp - Directory User Agent")
                    (WIDTH 350))

    start_buttons
}

LAYOUT : start_buttons : HORIZONTAL
{
    P.Button        ((LABEL "Quit")
                    (COMMAND quit_command)
                    (NAME quit))

    P.Button        ((LABEL "Look Up An Entry")
                    (COMMAND raise_lookup)
                    (NAME look))

    P.Button        ((LABEL "Find A Named Entry")
                    (COMMAND raise_moveto)
                    (NAME moveto))

    P.Button        ((LABEL "Help")
                    (COMMAND raise_help)
                    (NAME help))
}

```

Program 1: Definition of window startup

- **Make Insensitive** – Make a named primitive or primitive type insensitive to user events
- **Raise Window** – Raise or create a named window object
- **Close Window** – Close a window
- **Keep Window** – Pin a window to the desktop
- **Abort Directory Operation** – Abort a directory request
- **Activate Named Primitive** – Activate the command associated with a particular primitive

A brief description of these follows.

3.3.1. Raise, Close and Keep Window

These operations allow the user to manage the creation and updating of application windows. The *keep* operation pins a window to the desktop and prevents it's contents from being overwritten, i.e. to maintain some particular piece of information. *Close* pops down a window, whether it has been pinned down with a *keep* operation or not. The *raise* operation causes a specified window to be either created or raised to the top of the window stack. This can be performed in a way that obeys one of four rules:

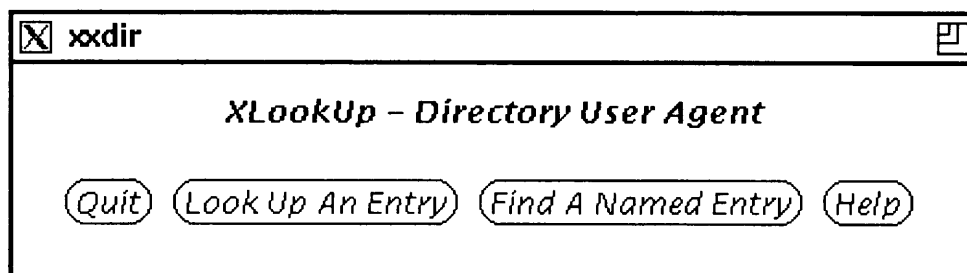


Figure 5: Appearance of window startup

- **Globally Managed** – A window is created if there are no other windows of the same type currently popped up and none of these have been kept, otherwise raise the first window of the same type found.
- **Locally Managed** – As previous rule except applied only to children of a particular window
- **Global New Always** – Always create a new window
- **Local New Always** – As previous rule except applied only to children of a particular window
- **Global New When None** – A window is created if none of the same type currently exist
- **Local New When None** – As previous rule except applied only to children of a particular window

3.3.2. Activate Named Primitive

This operation initiates the action of a named primitive and tells the primitive where the results of the action must be sent. Program Listing 2 defines the activation of action `lookup` (a directory look up) in a primitive of type `P.Lookup` and directs output to a `read` window if one match is made, or to a `list` window if more than one match is made. The final parameter shown in the fragment declares the rule to use when choosing which particular window instantiation to send the results of the operation to, as there may be several or no potential output windows already in existence. The parameter list is defined by the type of action being initiated, for example the action `read_dn_attribute` reads the entry pointed to by a DN attribute within an entry, e.g. a secretary attribute, and can only send output to one type of window as there is only one possible result to a successful action, the contents of an entry.

3.3.3. Make Insensitive and Make Sensitive

Given certain states in a user interface, some primitives may need to be inactive. These operations permit named primitives to be made active or inactive under certain conditions or during the course of a user defined command. An example of this can be seen in Figure 6, where a directory look up is being performed. Here all active areas except the *Abort Lookup* button have been made insensitive.

The configuration text which defines this is shown in Program Listing 3. This shows how the defined command `lookup_command` is attached to the button labelled `StartLookup`. In this command all buttons are made insensitive, then instances named `abort` are made sensitive. The inverse is performed when the activated action `lookup` has completed.

```
A.Activate
(P.Lookup,          # Primitive type
lookup,            # Name of action to initiate
read,              # Send output to if one match made
list,              # Send output to if many matches made
GlobalManaged)    # Window management rule
```

Program 2: Startup Window Definition

3.3.4. Abort Directory Operation

This will abort the last directory query issued from the originating window, i.e. the window which issues the abort request. The configuration regime does permit windows to have multiple outstanding requests to the Query Engine, although this behaviour is deprecated as the abort operation will only abort the last query issued from a window. In the standard configuration files requests are limited to one outstanding per window by making activatable primitives insensitive during the course of a request, as shown in the previous section.

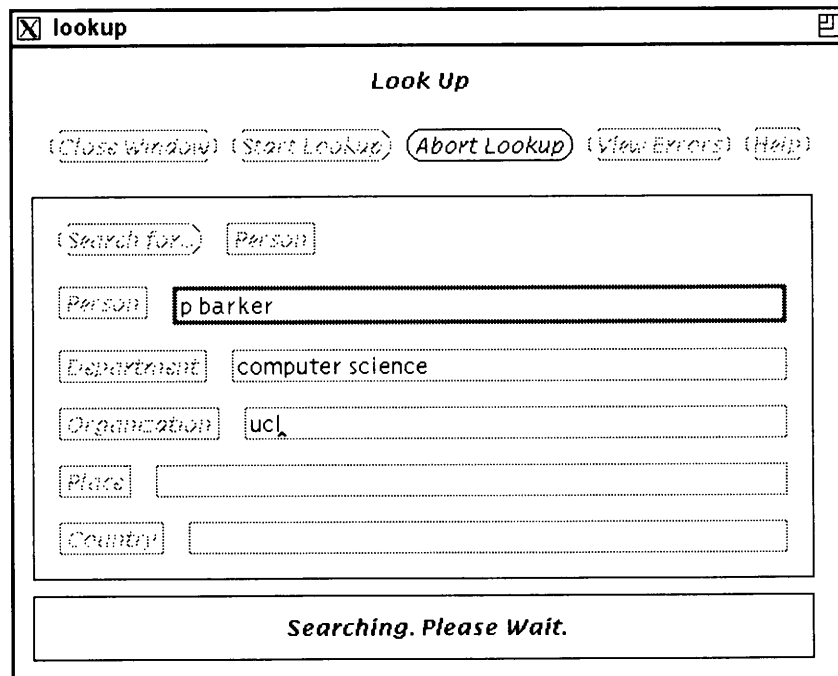


Figure 6: Controlled sensitivity

```

.
.
P.Button      ((LABEL "Start Lookup")
              (COMMAND lookup_command))
.
.
COMMAND: lookup_command: (A.MakeInsensitive(P.Button),
                          A.MakeSensitive(abort),

                          A.Activate
                          (P.MoveToEntry,
                           lookup,
                           read,
                           list,
                           GlobalManaged),

                          A.MakeSensitive(P.Button),
                          A.MakeInsensitive(abort))

```

Program 3: Controlling sensitivity

3.4. Handling Directory Output

Program Listing 2 shows how data created as the result of directory operation can be redirected to specific application windows. In an extension to this, data, which may be entry names or the contents of an entry, may be redirected to other kinds of output receptors. At present the design caters for three types of receptor:

- **X selections** – This enables interaction between the directory and other information handling agents, e.g. mail user agents, local databases etc. This is planned to integrate with the XUA X.400 mail user agent implemented at Nottingham University.
- **External filter processes** – An example of this might be a directory entry to business card convertor.
- **Address Book** – Output to a local address book.

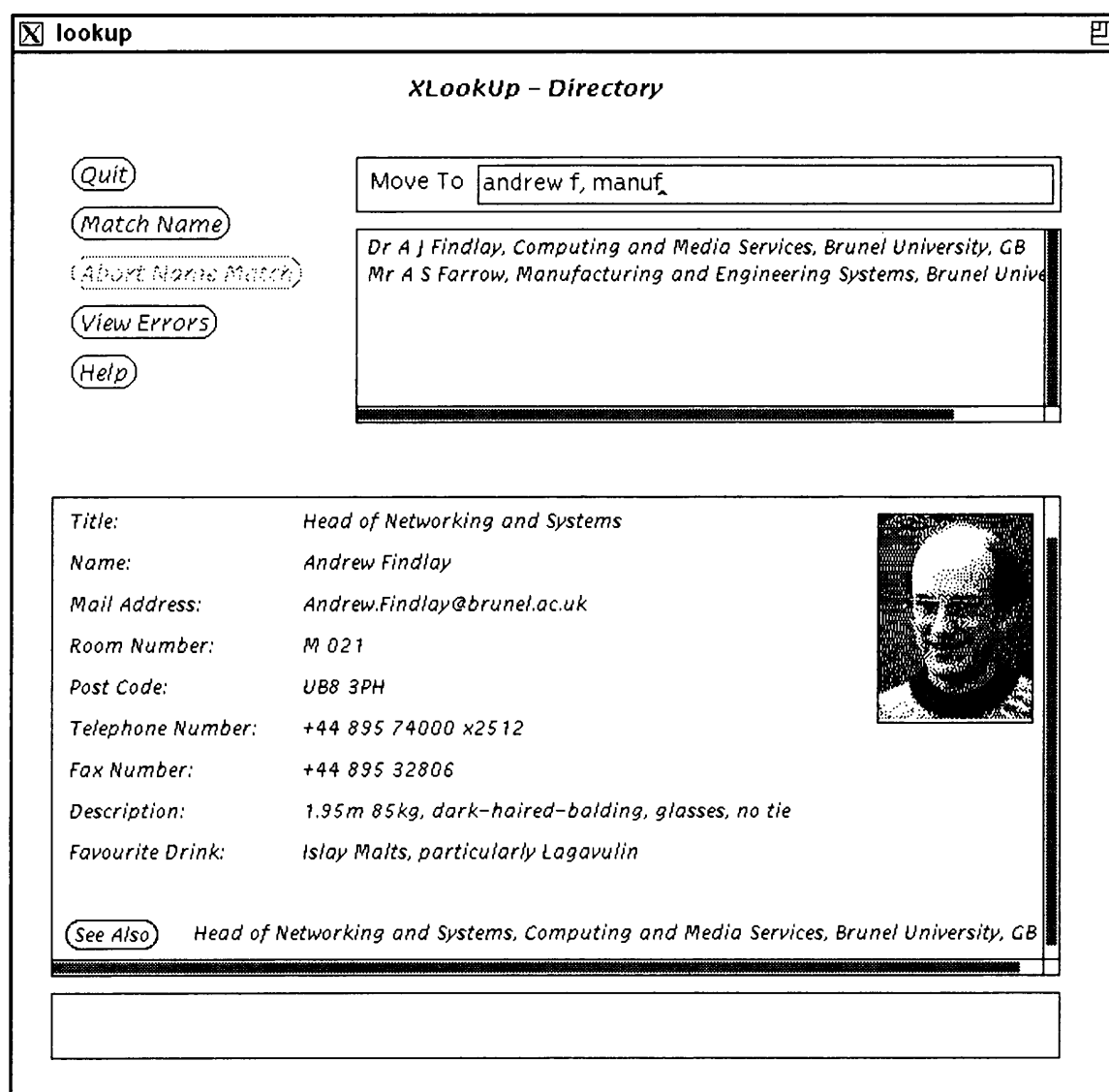


Figure 7: XLookUp configured as a single window application

From a configuration point of view this can be handled in the same way as output redirect to a window object. Thus the configuration text shown in Program Listing 4 specifies the sending of a search result into an X selection referred to for configuration purposes as C.Names. The final parameter shown in the A.Activate operation is in this case unnecessary and is thus ignored.

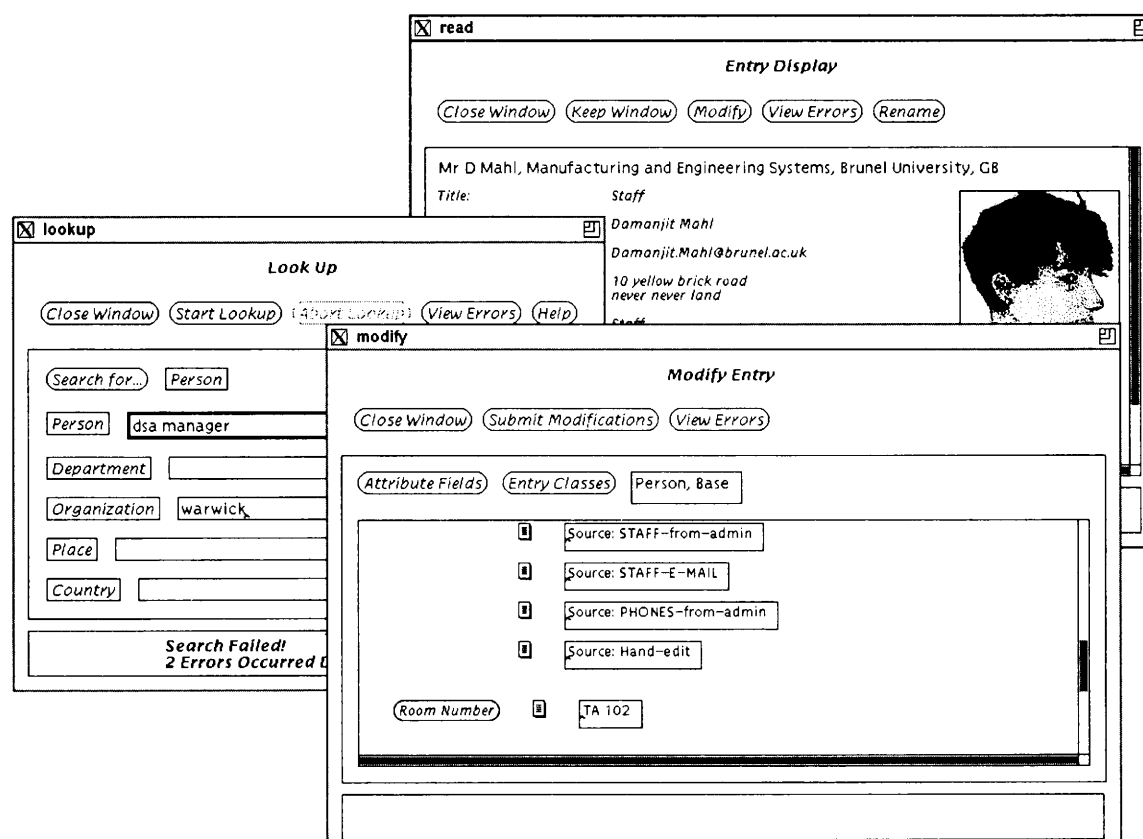


Figure 8: XLookUp configured to multiple windows

```
A.Activate
(P.MoveToEntry,
lookup,
C.Names,
C.Names,
GlobalManaged)
```

Program 4: Sending output to a cut buffer

External output processes can be defined with the OUTPUT configuration option.

```
OUTPUT :  
    business_card :      # name of output type  
    card :              # name of output process  
    commonName,         # selected attributes of entry  
    textEncodedORAddress,  
    photo,  
    DN
```

Here the second field is the name tag to be used in the configuration file, the third is the name of the output process itself and the final field contains a filter for the data required by the output process.

At time of publishing this idea has not yet been implemented, though it is expected to be ready in time for inclusion in the initial release.

3.5. Example Setups

Figures 7 and 8 show two application shots, each using a different configuration. One shows an all-in-one approach to life, the other makes use of multiple windows. These are not standard configurations, they only serve to illustrate the flexibility outlined in this paper.

4. Implementation status

The implementation is now near completion. Beta tests are currently scheduled for late July 1991 (as of June 1991). The initial version uses the Athena widget set. A further implementation will be made using the Motif widget set, though this is not expected until early 1992. Further information can be obtained from the address:

x500@brunel.ac.uk

References

- [Kil89a] Stephen Kille, *New Name Forms*, ISO/IEC/JTC 21/WG4/N797 UK National Contribution to the Oslo Directory Meeting, May 1989.
- [Kil91a] Stephen Kille, *Using the OSI Directory to achieve User Friendly Naming*, UCL, London (January 1991). Internet-Draft

An Implementation of a Process Migration Mechanism using Minix

Sylvain R.Y. Louboutin

University College, Dublin, Ireland

Louboutin@ccvax.ucd.ie

Abstract

This paper describes an implementation of a process migration mechanism realised on a network of PCs running under Minix.

The design of Minix incorporates modern operating system design concepts (micro kernel, message passing, client-server model) and insures a good process encapsulation which is necessary for such a realisation.

Remote execution is achieved by using surrogates or stub processes so that the lack of integration of the platform can be overcome. Despite the drawback of leaving a residual dependency on the node where the migrating process has been created it appears to be an appealing approach particularly suitable to this system. The isomorphism between the built-in message passing mechanism and the add-on Amoeba communication protocol implementing RPC has been extensively used to this purpose.

Minix, even enhanced with the Amoeba network communication facility is certainly not a distributed system. However, it features some of the properties which makes feasible the realisation of a process migration facility. Furthermore, Minix does not require an actual network to develop and test networking software which can be done on a standalone system. It therefore is a suitable and inexpensive platform to conduct such experiments.

1. Background

A system made of independently owned workstations connected by a fast local area network provides a large potential computing power which is frequently under-utilized [Liv82a, Wan85a, Nic87a, Cov88a, Lit88a, The88a]; some workstations are overloaded while other are idle. This makes it highly desirable to share the work load among the different stations.

The distribution of the work load may be simply achieved by automatically initiating any new process on an idle host where it executes until completion. It has been shown [Eag86a] that even a trivial adaptive load sharing policy yields dramatic performance improvements by using the otherwise wasted computing power.

Krueger and Livny [Kru88a] have further shown that the introduction of a pre-emptive process migration mechanism significantly improves the performance of a system already capable of non-pre-emptive placement of processes. A process migration mechanism makes possible the implementation of receiver initiated strategies [Eag86b], and multiple relocations which reduces the granularity of the distribution.

Process migration could also be used as a tool to achieve fault tolerance (variation on check-pointing techniques), or more generally as a way to deal with a lack of locally available resources i.e. not only CPU. But this work focuses specifically on the use of a process migration mechanism as a tool to implement a load distribution policy.

2. Introduction

A prototype has been implemented on a network of PCs running under Minix (version 1.3) and communicating over Ethernet. This paper shows that even if Minix is far from being a genuine distributed operating system [Tan85a], or may even be considered as a toy operating system (dixit Tanenbaum author of Minix), it still exhibits the qualities which makes feasible the implementation of a process migration mechanism. Therefore Minix provides an inexpensive and suitable platform for such experiments.

3. Process Migration

A process is basically a program in execution [Tan87a]. A process migration mechanism makes it possible to transfer a process from one machine to another one at any time during its life time. This entails a pre-emptive extraction of a process from its original host and its re-installation into another one where its execution must resume. The amount of information which actually has to be transferred, is called the process state.

If the migration is performed before any computation has started, i.e. in the special case of a process placement, the process state consists of just the code of the program executed. Otherwise the process state is made of the whole address space of the process (code, heap and stack), and of some information maintained by the operating system. These two parts of the process state are sometime named respectively swappable and non-swappable parts in analogy with operating systems featuring process swapping. This naming convention will be used in this paper despite the fact that Minix does not provide process swapping.

Furthermore, the execution environment must provide a mechanism whereby processes may transparently keep on accessing their resources while executing remotely. Ideally, in a truly integrated distributed operating system, resources would be accessible regardless of their actual location in the network. A system made of stations running Minix lacks this level of integration and a way to achieve transparent remote execution must be provided.

Process migration has been implemented on a number of experimental systems [Pow83a, Art86a, The86a, Zay87a, Smi88a, Dou87a] with sometimes different goals and using different approaches. But systems where it has been successfully implemented exhibit two main characteristics which are:

- **Process encapsulation.** Interactions between a process and its environment are performed via a message passing mechanism. Systems which feature light weight processes restrict the granularity of migration (e.g. in V-System where light weight processes share the address space of one logical host the logical host constitutes the unit or grain of the migration.)
- **Indirection between physical and logical addresses.** Messages are sent to location independent logical addresses but not to physical addresses. The mapping between the latter and the former is handled by the operating system so that the references a process has to its environment may be transparently rebound after its migration. For instance messages are addressed to logical hosts or a logical process group in V-System, through links in DEMOS/MP or Charlotte or to ports in Accent. Sockets for example (BSD UNIX) would not be suitable because they are bound to location dependent addresses.

4. Minix

This chapter provides a brief description of Minix. It emphasises the similarities of form between its built-in message passing mechanism and its add-on Amoeba like communication facilities which have been used as the basis for the realisation of the remote execution facility.

The different components of Minix act as independent processes communicating via a message passing mechanism adopting a blocking unbuffered rendez-vous like semantic. Minix uses the client-server model and is made of a hierarchy of four layers of processes with the user processes on the top. Processes may use the services provided by processes belonging to the same layer or to the layer immediately below.

The Minix built-in message passing facility appears to user processes as three primitives which are:

```
send(dest, &message);
```

to send a message to the process dest identified by its slot number in the kernel process table.

```
receive(source, &message);
```

to receive a message from a specific process source or from **any** process.

```
send_rec(src_dst, &message);
```

to send a message and wait for a reply.

Services usually performed by the kernel in other monolithic systems, are provided to user processes by two server processes which are the Memory Manager (MM) and the File System (FS), both executing outside the kernel. The kernel itself is made of a set of (uninterruptible) tasks which are the device drivers.

Technically, Minix has only one system call (implementing the three primitives described above), used by user processes to send or receive messages to or from one of the two servers MM or FS. A procedural interface is provided by the standard library to ensure the compatibility with UNIX V7. It is important to stress that all interactions between a process and its environment i.e. outside its address space goes through this unique mechanism.


```

/* A Minix server providing three services arbitrarily named FIRST, SECOND
and THIRD */
message m;
int status;

while (TRUE) {
    receive(ANY, &m);
    switch (m.m_type) {
        case FIRST:      status = do_first(<some-arguments>;      break;
        case SECOND:     status = do_second(<some-arguments>;     break;
        case THIRD:      status = do_third(<some-arguments>;      break;
        default:         status = ERROR;
    }
    m.m_type = status;
    send(m.m_source, &m);
    /* the field m_source designates the client and is automatically
updated */
}

/* A client requesting the SECOND service. */
message mess;

mess.m_type = SECOND;
mess.<some-field> = <some-value>;
sendrec(SERVER, &mess);
<some computation using the results>

```

Figure 1: A typical Minix-server and a client

MM and FS are not stateless servers. They both maintain information about their clients. The Kernel in charge of message passing and context switching must also maintain its own information about processes. In Minix a process state is fully defined by the address space of the process, which is made of three parts: the text, data and stack segments; and by the content of three tables maintained separately by MM, FS and the Kernel (the information held by MM and FS in their respective tables correspond roughly to the u-area in other UNIX systems.)

Minix uses an adaptation of the connectionless network communication protocol initially developed for the Amoeba distributed operating system. This protocol implements Remote Procedure Calls (RPC) and is based on a four layer model (instead of the seven layers of the ISO OSI model) the physical layer being the Ethernet.

A Port uniquely identifies a server process and provides a logical address to which all messages are sent. The location of a service i.e. the mapping between a port and the actual location of the server process listening to it is handled by the port layer. This is implemented in Minix by the kernel which maintains a table of hints which is eventually updated by broadcasting a location request to other kernels. The port layer provides a datagram service that is the delivery of 32 kilobytes datagrams but with no guarantee on delivery.

The actual message service i.e. the reliable transport of bounded 32 kilobytes requests and replies is handled by the next layer named the transaction layer. The interface of this third layer (partly handled by MM) to applications is a set of four primitives which are:

```
getreq(hdr, buffer, size);
```

used by a server to accept a request.

```
putrep(hdr, buffer, size);
```

used by a server to send back a reply.

```

/*
  An Amoeba server providing three services arbitrarily named FIRST, SECOND and THIRD.
  This server listens to the port named &RMyServ&S.
*/
header hdr;
char buffer[BUFSIZE], reply[BUF2SIZE], *strncpy();
ushort size, replysize, status, getreq(), putrep();

signal (SIGAMOEBA, SIG_IGN);
while (TRUE) {
    strncpy(&hdr.h_port, &RMyServ&S, HEADERSIZE);
    size = getreq(&hdr, buffer, BUFSIZE);
    if ( (short) size < 0 ) {
        handle_error();
        continue;
    }
    switch (hdr.h_command) {
        case FIRST:    status = do_first(<some-arguments>);    break;
        case SECOND:   status = do_second(<some-arguments>);   break;
        case THIRD:    status = do_third(<some-arguments>);    break;
        default:       status = ERROR;
    }
    hdr.h_status = status;
    putrep(&hdr, reply, replysize);
}

/* A client requesting the SECOND service */
header hdr;
char buffer[BUFSIZE], *strncpy();
short size;
ushort trans();

strncpy(&hdr.h_port, &RMyServ&S, HEADERSIZE);
hdr.h_command = SECOND;
timeout (100); /* the transaction will fail if the service is not
                * located within 10 seconds */
size = (short) trans(&hdr, buffer, BUFSIZE, &hdr, buffer, BUFSIZE);
if (size < 0)
    printf(&RTransaction failed (%d)OS, size);
else if (hdr.h_status == ERROR) <deal with error>;
<some computation using the results>;
}

```

Figure 2: Typical Amoeba-server and a client

```
trans(hdr1, buffer1, size1, hdr2, buffer2, size2);
```

used by a client to perform a transaction i.e. send a request and wait for a reply.

```
timeout(time);
```

used by a client to set a timeout to the transaction (time to locate the service and not to perform the request.)

The last layer or service layer defines the actual semantic of the service provided by a server process. In the example (Figure 2) three services are provided namely FIRST, SECOND and THIRD.

Two different message passing mechanisms coexist within the same system i.e. the built-in Minix one and the Amoeba add-on one. Messages exchanged using the former will be named Minix-messages whilst messages exchanged via the latter are Amoeba-messages. Accordingly, a server process using such a mechanism will be called either Minix-server or Amoeba-server. The similarities between both message passing mechanisms will be taken advantage of in the realisation of the remote execution facility.

5. Implementation

The implementation will be described by first introducing the naming convention used throughout the rest of this paper and by giving an overview of the sequence of events (or scenario) involved in a migration.

The places where the action takes place are:

- The original node (or host) is the birth place of the migrating process.
- The *source node* is the place where a migration is initiated.
- The *destination node* is the place where the migrating process will eventually resume execution.

The cast of characters are:

- The old incarnation of the migrating process. The old incarnation on the original host becomes the surrogate.
- A target process on the destination host which will be possessed to become the new incarnation of the migrating process. It is usually forked off by the migration daemon on the destination host.
- The migration daemon on the destination host is in charge of re-installing the new incarnation of the migrating process. It is an Amoeba-server acting on the behalf of the triggering process.
- The triggering process initiates the migration and is in charge of the extraction of the migrating process. It resides on the source node and is a client of the migration daemon.

The scenario is (events are numbered chronologically, number 4 and 4' may take place simultaneously):

1. The triggering process on the source node initiates the migration by extracting the non-swappable part of the migrating process and then requesting the migration daemon to install it on the destination node.
2. The migration daemon forks off a new target process and possesses it using the non-swappable part of the migrating process it received from the triggering process. The target process becomes a ghost.
3. The triggering process and the migration daemon then cooperate to ship the address space of the migrating process.
4. The triggering process sends the SIGMIG signal to the old incarnation which then executes the surrogate program (if appropriate i.e. if the source node is the original node.) The surrogate listens to its private port waiting for the new incarnation requests.
- 4'. The migration daemon sends the SIGMIG signal to the new incarnation which resumes execution in remote execution phase.

The action is made of two phases:

- The transfer phase starts when the triggering process initiates the migration. During that phase the migrating process is frozen and its state is shipped across the network.
- The remote execution phase starts when the new incarnation resumes execution (starting after 4 and 4'.)

These two phases could be addressed independently but the strategy adopted for remote execution dictates what actually needs to be

transferred from the process state. Remote execution is thus described first.

5.1. Remote execution

From a user process point of view, a UNIX-like system provides only two abstractions which are the processes and the files. Processes are organized in a tree like hierarchy with "Init" as the root; files are themselves organized in another tree like hierarchy with "/" as the root (I/O devices, directories or pipes which appear as special kind of files do not constitute a separate abstraction.) This concept appears in the design of Minix which clearly separates the roles within the operating system between MM (in charge of processes) and FS (in charge of files.)

The lack of integration of a system made of UNIX-like workstations partly comes from the fact that each node has its own processes-tree and its own files-tree. Systems like NFS or Newcastle Connection [Cou88a] partly overcome this by providing ways to unify, or at least connect, the different file trees; a system like Chorus/MIX [Arm89a] allows the different process trees to span over the network.

Migrating a process will involve pruning and grafting. The view a process has of its environment (as well as the view the environment has of the process) depends heavily on its relative position in the process hierarchy. Therefore to execute transparently on a remote node, a process has to maintain its position in its original hierarchy.

In the proposed prototype a migrating process is replaced in its processes hierarchy by a surrogate. A surrogate acts as a private Amoeba-server for the migrating process and performs system calls on its behalf. This has already been adopted by systems such as Sprite. It imposes a residual dependency on the original host, but there only. It does not leave dependencies on any of the nodes a migrating process may have successively visited as in DEMOS/MP where a process leaves behind a trail of forwarding addresses.

A process executing remotely forwards system calls by embedding the content of the Minix-messages which would have otherwise been delivered locally to either MM or FS within a Amoeba-message which is sent across the network to its surrogate. The surrogate passes it on to the local appropriate Minix-server. The surrogate is thus mostly a straight forward Amoeba-server. A process likely to be migrated must be linked with a modified version of the standard library where the procedural interfaces to each of the system calls are provided.

Three difficulties remain and must be addressed:

1. Minix indulges itself into passing large arguments by reference. In that case (for instance for read() or write() system calls,) the content of the buffer must be sent along with the message (simple marshalling/unmarshalling.)
2. A process executing remotely must receive signals which may have been sent to it on its original host. To achieve this, the surrogate provides a handler for all signals and maintains a bitfield with one bit for each signal received since the last system call. This bitfield is sent along with the reply to the next system call the remote process will eventually perform. The remote process may then act accordingly.
3. When a process executing remotely forks, its state is copied back to its original node. The surrogate may then create a child identical to the migrating process. The processes-tree on the ori-

```
int migrate(process, buffer, map)
/* returns the actual size of the buffer */
int process; /* pid of the target process */
char *buffer; /* buffer for non-swappable part of the process state */
struct mem_map *map; /* memory map */
```

Figure 3: Declaration of *migrate()*

ginal node evolves as if no migration has occurred. It is assumed that the migrating process is aware of the fact that it is executing remotely and that it has been provided with the name of the private port of its surrogate (see transfer.)

5.2. Transfer

As a consequence of the method adopted to achieve remote execution, only the content of the tables of MM and Kernel needs to be respectively extracted from the original node and re-installed into the remote destination.

The modification of Minix consists of three new system calls and a new signal. This new system calls are provided by MM and require two new services from the Kernel freeze and resume. *Migrate()* takes care of the extraction and *metaexec()* of the re-installation. *Getorg()* is used by the migrating process itself and a new signal *SIGMIG* is used for the resumption of both incarnations. Next we describe these new tools in the sequence in which they are used.

5.2.1. Migrate()

This system call is used on the original host by the triggering process, typically a command activated by the shell, to extract the migrating process.

It accepts as argument the pid of the migrating process. It returns the portion of the non-swappable part of the process state belonging to MM and the Kernel along with the memory map of the process (data structure describing the memory layout of the process.) This system call freezes the migrating process.

5.2.2. Metaexec()

This system call is used on the destination host by the migration daemon to install the new incarnation of a migrating process.

It accepts the pid of the target process, the non-swappable part of the process state and the memory map of the migrating process. This system call does not create a new process but requires a target process which becomes possessed by the new incarnation of the migrating process. The memory layout of the target process is modified to match the memory requirements of the new incarnation.

```
int metaexec(process, buffer, size, map)
/* returns the actual size of the buffer */
int process; /* pid of the target process */
char *buffer; /* buffer for non-swappable part of the process state */
int size; /* actual size of the buffer */
struct mem_map *map; /* memory map */
```

Figure 4: Declaration of *metaexec()*

Private Port	Tag	Mode of Execution
<none>	LOCAL	I. Default for any process
<none>	REMOTE	II. What's left after an intermediate migration
<PORTNAME>	LOCAL	III. This is a surrogate
<PORTNAME>	REMOTE	IV. This is a new incarnation

Figure 5: *Different modes of execution*

As the content of the address space has yet to be transferred, the new incarnation remains frozen in a special state called ghost. Metaexec() returns the updated memory map of the actual memory layout of the ghost, the physical addresses of the different segments being changed to match the new location. That way the migration daemon then knows where to install the address space of the migrating process.

Two new fields are added to the information held by MM about each process. The first one is the name of a private port eventually used by a process executing remotely to communicate with its surrogate; the second one is a tag indicating whether the process is executing locally or remotely. Both fields are updated by the migrate() and metaexec() system calls depending on their previous value. The combinations of possible values of this two fields define the mode of execution.

When migrate() is called upon a process in mode I., a unique private port is created for it. The process is tagged as being LOCAL and is then changed to mode III. Migrate() fails if it is called upon a process in mode III because a surrogate must not be migrated. The mode of execution is not affected by the execve() system call and remains unchanged.

On the destination host, when the target process is possessed by the new incarnation of a migrating process (with a call of metaexec()), it is tagged as being REMOTE. As it inherited the private port from the migrating process it is in mode IV. If migrate() is called upon a process in mode IV i.e. for a subsequent relocation, the field containing the private port name is cleared. The process is then in mode II.

It is possible to restore a process in its previous state using metaexec() with a special set of parameters. This is necessary to recover from a migration failure or when migrate() and metaexec() are used to perform a remote fork() (typically to move from mode II back to mode IV.)

A migration daemon (an Amoeba-server) resides on each node willing to accept incoming migrating processes. It is in charge of installing a migrating process and transferring the address space. The address space of a migrating process is shipped in chunks which must be smaller than 30,000 bytes (to fit in an Amoeba-message). Once the whole process state has been transferred, SIGMIG is sent to both incarnations.

5.2.3. SIGMIG

SIGMIG cannot be ignored. If not caught, the receiving process is killed. Sending SIGMIG is the only way to unfreeze both incarnations of a migrating process. A process expecting to migrate must then provide a signal handler for SIGMIG. The signal handler uses getorg() system call to determine what its mode of execution is (see Figure 5).

In mode I., the handler returns without any further action (the signal may have been sent to recover from a failed migration or after a remote

```
int getorg(org)
/* returns either LOCAL or REMOTE */
port *org;    /* private port of the surrogate */
```

Figure 6: Declaration of *getorg()*

fork()). In mode II, the process must terminate to avoid leaving any residual dependency on a node visited en route. In mode III, it executes the surrogate program. In mode IV, it just resumes execution in remote execution phase.

5.2.4. **Getorg()**

This system call is used by the signal handler that the migrating process must provide for SIGMIG. It returns both the value of the tag and the private port of the calling process. The migrating process is then able to determine what is its mode of execution and what is the port name of its surrogate.

6. Discussion

The proposed mechanism lacks the transparency claimed by other implementations. The process candidate for migration must have been linked with a special library and more importantly must provide a handler for the new SIGMIG signal.

Remote execution is accomplished using a surrogate with the drawback of leaving a residual dependency on the original host.

The semantics of signaling is slightly altered when the process executes remotely. A process should receive a signal in two cases [Bac86a]: when it returns from a system call or when re-scheduled after having been pre-empted by the scheduler. A process executing remotely can receive a signal only in the former case.

A migrating process has full access to all its environment regardless of its location and there is actually no restriction on which kind of programs may be executed (it might not be a very palatable for highly interactive programs such as an editor, but it would work.)

The modification of the operating system is minimal and most of the transfer is actually handled by user processes (the triggering process and the migration daemon.) The content of the table maintained by FS does not need to be either extracted or replaced. Whilst all the messages addressed to FS must be forwarded to the surrogate, only a few of the messages addressed to MM (e.g. the special case of *exit()* which has to be executed both locally by the terminating process and by its surrogate) need to be.

This mechanism should be suitable for different kinds of load distribution policies. The replacement of a CPU bound process (typical candidate for migration) by a surrogate program which has an I/O bound behaviour should remove a significant load from the original node (no experiment of load distribution policy using this mechanism has been conducted yet.) Any node on the network may become a satellite or processing server of any other one. The mechanism is sufficiently flexible to allow each node to adopt its own policy (basically by customizing the triggering process and migration daemon.)

7. Conclusion

The design of Minix provides a good process encapsulation and clearly separates the roles between the management of processes (via MM) and files (via FS.) The network communication protocol borrowed from Amoeba clearly separates the notion of logical addresses (ports) from actual location. These properties made possible the implementation of a process migration mechanism.

The isomorphism between both the message passing mechanism used internally in Minix and the message passing mechanism used for communication over the network has been extensively used for the remote execution facility. But this isomorphism is not sufficient and the mere fact that two distinct message passing mechanisms coexist is itself an obstacle to a true integration which had to be overcome by using surrogates.

Nevertheless, Minix provides a suitable and inexpensive platform to conduct experiments involving non trivial issues of distributed operating systems.

Acknowledgement

Thanks to Gabriel McDermott (UCD) for his valuable help and advice in the writing of this paper.

References

- [Arm89a] Francois Armand, "Revolution 89 or Distributed UNIX Brings it Back to its Original Virtues," in *Proceedings of the '90 Ft Lauderdale workshop on distributed and multiprocessor systems* (1989).
- [Art86a] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel, "Processes Migrate in Charlotte," Report No 655 (August 1986).
- [Bac86a] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall (ISBN 0-13-201799-7 025) (1986).
- [Cou88a] George F. Coulouris and Jean Dollimore, *Distributed Systems – Concepts and Design*, Addison-Wesley Publishing Company (ISBN 0-201-18059) (1988).
- [Cov88a] Luis L. Cova and Rafael Alonso, "Distributing workload among independently owned processors," Technical Report No. CS-TR-200-88 (December 1988).
- [Dou87a] Alfred Douglass and John Ousterhout, "Process Migration in the Sprite Operating System," University of California at Berkeley, Technical Report No UCB/CSD 87/343 (February 1987).
- [Eag86b] D. Eager, E. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Server-Initiated Dynamic Load Sharing," *Performance Evaluation* 6(1), pp. 53-68 (April 1986).
- [Eag86a] Derek L. Eager, Edward D. Lazowska, and John Zahorjan, "Adaptive load sharing in homogeneous distributed sys-

- tems," *IEEE Transactions on Software Engineering* **12**(5), pp. 662-675 (May 1986).
- [Kru88a] Phillip Krueger and Miron Livny, "A Comparison of Preemptive and Non-Preemptive Load Distributing," in *Systems*, San Jose, California (June 1988).
- [Lit88a] Michael J. Litzkow, Miron Livny, and Matt W. Mutka, "CONDOR - A Hunter of Idle Workstations," pp. 104-111 in *Systems*, San Jose, California (June 1988).
- [Liv82a] M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," in *Sigmetrics* (April 1982).
- [Nic87a] D. Nichols, "Using Idle Workstations in a Shared Computing Environment," in *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Austin, Texas (November 1987).
- [Pow83a] Michael L. Powell and Barton P. Miller, "Process Migration in DEMOS/MP," in *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (1983).
- [Smi88a] Jonathan M. Smith, "A Survey of Process Migration Mechanisms," *ACM Operating Systems Review* **22**(3) (July 1988).
- [Tan85a] Andrew S. Tanenbaum and Robbert van Renesse, "Distributed Operating Systems," *ACM Computing Surveys* **17**(4) (December 1985).
- [Tan87a] Andrew S. Tanenbaum, *Operating Systems - Design and Implementation*, Prentice-Hall International Editions (ISBN 0-13-637331-3) (1987).
- [The86a] Marvin M. Theimer, "Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems," Technical Report (PhD thesis) No STAN-CS-86-1128 (CSL-86-302) (June 1986).
- [The88a] Marvin M. Theimer and Keith A. Lantz, "Finding Idle machines in a Workstation-Based Distributed System," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, California (June 1988).
- [Wan85a] Y. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Transactions on Computers* **34**(3), pp. 204-217 (March 1985).
- [Zay87a] Edward R. Zayas, "Attacking the Process Migration Bottleneck," pp. 13-24 in *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Austin, Texas (November 1987).

HAWKS – A Toolkit for Interpreted Telematic Applications

Carl Verhoest

Télesystèmes Innovation, Paris, France

cave@telesys-innov.fr

Abstract

On-line information retrieval systems offer a wide range of information. Many companies rely on these informations for their strategy planning. Since most of these systems differ in their structure and interface, their use is not as trivial as wished. The workstation HAWKS, presented in this paper, offers a unique interface for information retrieval on a large selection of services. This interface can be used directly by end-users in a fully graphical supported environment. At the same time, HAWKS offers an environment which enables the rapid development of interpreted applications in the domain of on-line information retrieval, without the need to know the targeted service in detail. At the basis of both environments are the HAWKS Access Language and the Global Dictionary. In this Global Dictionary all the On-line services are modelled according to a conceptual model, and implemented in a C++ objects persistent environment.

1. Introduction

More and more, now a days, companies relay their strategy on a constant information flow. Not only the information about the companies internal state is concerning, but information about concurrents, concurrent products, and marketing aspects is used. These latter informations change on a regular (daily, weekly ...) basis, and therefor require a constant following from the companies concerning. Uptodate information has become a major key in company policies, implying a huge information need.

To be able to meet this information need, companies have numerous media at their disposal: printed matter (newspapers, domain oriented magazines etc.), information brokers (consultants, marketing specialists etc.), local database- and information- systems, and On-line Information Retrieval Systems (OIRS). Local database/information systems and OIRS often have the advantage of providing the information seekers (end-users) with some information selection mechanism; instead of having to go through all the available information, the wanted information can be directly selected. This is normally done by means of query formulations, resulting in quicker, and therefor more cost effective, information retrieval. As stated above, one major concern is the time-relevance of the information. It is clear that OIRS have a better chance

to be more up to date than local systems, as far as it concerns information from outside the company.

In this article we are concerned about the usage of OIRS systems; the way of selecting the right service, finding the right information and reusing the found information.

In the next sections we will state the general problems one encounters using OIRS, provide a global solution at the end-user level and present one practical implementation, in a UNIX environment, of this solution: HAWKS (the Homogeneous Access WorKStation). The conclusions will be followed by a set of objectives for the near future of this project.

2. OIRS Troubleshooting

Considering that OIRS exist since the sixties, and the great variety of services potentially available to the end-user, the actual use of these systems has been much lower than expected. The following reasons can be identified:

- There is no comprehensive global picture of the services being provided (a global directory), therefor the user has no easy means of identifying which service could provide the sought information, how to log-on to it, how to use it, etc.;
- Each service has its own data structure and its own access language; therefore the end-user has to master a new set of rules, for each service he wants to access;
- The Access Language provided by these services (besides being different from service to service) are usually very cumbersome, slow and unfriendly. It is usually keyword oriented and makes little use of modern techniques, such as those which are available in graphic interactive interfaces;
- The services have been conceived for access by simple devices (dumb terminals). Information is therefore usually sent as a stream of characters. Modern business applications require, instead, the possibility of organizing this information using popular application packages (spreadsheets, databases, word processors etc.) or the possibility of reusing the data retrieved from the remote service in local applications, developed with conventional programming languages;
- There is a growing need to provide, in conjunction with the requested information, additional information, very often of a sophisticated graphic nature.

3. TOOTSI

On-line services are to be accessed by dumb terminals or pc's running some communication software and a terminal emulation. It is clear that the locally available hardware can be exploited at a much further extent than it is done actually.

The Esprit 2 project 2109 TOOTSI (Telematic Object Oriented Toolkit for System Interfaces) has for aim to provide a toolkit enabling rapid application development in the domain of On-line services. Such applications should run on the locally available hardware, and fill the gap between the end-user and the OIRS. These applications should also

provide an added value to the basic work of information retrieval such as integration with existing software on the workstation.

One first condition [TOO89a] for this toolkit is that it provides an application developer with a uniform view of all targeted On-line services. This uniform view has to include features as: communications, query languages and information restructuring.

A second condition is derived directly from the On-line service environment: since the On-line services are known to change frequently their interfaces, query languages etc. it is clear that the application, build with the toolkit, is to be adaptable to these new changes without recompiling.

The toolkit will support the development of applications at two levels:

1. Compiled applications

Here the applications are developed by a so called TOOTSI application programmer. The toolkit presents itself as a library of classes, providing a unique, object oriented, view of the On-line services, as well as a set of graphical objects to construct the User Interface (OSF/MOTIF) for the application. The latter assuring all TOOTSI applications to have the same "look and feel". The application is developed using these classes in a C++ environment. The resulting product will apply to a specific domain of interest; some flexibility of the existing On-line service(s) will be exchanged for the ease of use by the end-user. The application programmer in this case is a programmer knowledgeable in object oriented programming, and the end-user's need.

2. Interpreted applications

There are two main reasons for enabling interpreted applications:

- a) In a compiled application all the user steps are foreseen and limited to the specifications of the application. Hence, only a subset of functionalities of the remote service are available and they are ordered in some scenario. Foreseeing all the possible end-user wishes in a compiled environment asks for elaborate context checking by the application programmer. In an interpreted solution some of this context checking will be relayed upon the end-user.
- b) Compiled applications are fast and easy to use, but they foresee no eventual change in information needs. A change of an information need may result in redesigning the application, and therefor the intervention of an (external) application programmer.

4. HAWKS

4.1. Objectives

The, above mentioned, two considerations (lack of flexibility for the end-user, and lack of adaptability of the application) converge to the need of an interpreted toolkit. This toolkit will present itself as a workstation providing two main features:

- a) A unique access language to interrogate the On-line services; this interpreted language can be used directly by the end-user, and

will be mapped onto the native language of the connected service, according to the context.

- b) An application development environment, enabling the construction and checking of interpreted applications.

In this paper we will present such a toolkit for interpreted applications: HAWKS.

4.2. Software Functionalities

4.2.1. Multiple Database Connections

Now a days, an end-user is always limited to access one database at a time within one communication program. HAWKS provides multiple, simultaneous, connections to different services. Each connection is being considered as a session. Within HAWKS there is the possibility of data exchange between sessions. Not only does it facilitate cross-database searching, but cross-service searching becomes a trivial operation. The number of simultaneous sessions possible, depends on the available communication hardware.

Since several communication events can occur at the same time, on different lines, the communication layers are implemented using an event-driven approach. The X11-server events are used to inform the application about incoming data, line errors etc. This implementation has the advantage that it permits the communication layers to reside on a different machine in a network.

4.2.2. The Global Dictionary

As stated before, one of the main requirements for HAWKS was the flexibility vis-à-vis the changing of On-line services or the changing of the application requirements in terms of targeted services; if an On-line service changes its interface, or the application wants to access a service which was not foreseen, the workstation is to adapt to this new situation in a quasi automatic way. This means that all the information describing an On-line service is to be retrieved from a support, different from the compiled core. The solution proposed by TOOTSI, and used in HAWKS, consists in creating a so called GLOBAL DICTIONARY (GD) [TOO90a].

The GD contents can differ for every installed HAWKS, enabling a client specific configuration. The following types of information are typically stored in the GD:

- What On-line services can be accessed by this workstation
- For each accessible On-line service:
 - How to access it (automatic connection parameters)
 - Description of the native language of the service
 - What residing databases on the service can be accessed
 - Information about the databases domain
 - Description of the structures of the databases (formats, indexes etc.)
- What applications are available
- Which users are known to HAWKS and what are their rights in terms of connections and applications

- What are the local hardware constraints (especially for communications)

It is clear that these informations have a strong semantic correlation among them: not all the users may access all the On-line services, the native language contains functionalities which will not be applicable to all the databases, some databases can be found on different servers, applications can be restricted to one or more users, databases or servers, etc. In order to control the consistency, and to avoid redundancy the information is modelled by using a conceptual tool. The model is an extension of the Entity-Relation model [Col88a].

After the design phase of the contents of the GD, by means of the conceptual model, the model is mapped onto a C++ objects environment.

What information is needed by HAWKS depends on the context of the workstation; this context may, and will, change at run-time. In order to avoid a congestion of the local hardware the GD is implemented on a C++ objects persistent environment; objects are loaded into main memory when needed and restored, if modified, on a physical support. For workstations having limited hardware resources, a GD server working in a LAN environment, and a GD in a TOOTSI Gateway (WAN) are envisaged.

For minor changes to the GD (changes not interfering with the structure of the implemented conceptual model) a so called GLOBAL DICTIONARY ADMINISTRATION MANAGER (GDAM) is resident in the toolkit. The access to this GDAM is reserved for people having a profound knowledge of the conceptual model used, and of the targeted On-line service.

4.2.3. The Access Language

Probably, one of the main barriers for an end-user accessing an OIRS, is the systems access language. Some systems provide a menu driven interface to ease its use, but mostly some procedural language is proposed [UKO89a]. To know the access language is one thing, but to use it in a sophisticated way is another: it requires a profound knowledge of the databases structures.

Manipulating a procedural language and knowing the structure of the database are inherent to the use of OIRS. HAWKS does not alter that situation, however, it provides the user with ONE procedural language and ONE general database structure to access a wide range of On-line services.

The HAWKS Access Language (HAL), is a procedural language consisting of four major groups of commands:

- *Common Command Language (CCL)*

The CCL was defined by the International Standardization Organization (ISO) in 1988. It was conceived as a standard interface for OIRS, and adopted by On-line services such as ESA-IRS and EURONET-DIANE. It provides the end-user with one structure for all the databases and a procedural language (CCL) to interrogate these databases. The CCL and the database structure are adopted in HAL. The main features found in the database structure are the following:

- FILES; databases consisting of
- RECORDS; a group of data usually treated as a unit, consisting of

- INDEXES; fields in a record, a specific area used to store specific information. The main functionalities found in the CCL are:
- Connection to a database, and information about this database
- A browsing of existing indexes for the selected database
- A browsing of possible search terms for each index
- A query statement allowing the following operators:
 - ◆ BOOLEAN operators
 - ◆ TRUNCATION operators (left-handed, embedded, right-handed, limited and unlimited number of characters)
 - ◆ PROXIMITY operators (limited and unlimited number of words, specified and unspecified sequence)
 - ◆ SUFFIX limited search (general purpose index specified)
 - ◆ PREFIX limited search (numeric index specified)
- A Selective Dissemination of Information (SDI) functionality
- An On-line browsing of records found, in some format
- An off-line print on the OIRS of the records found, in some format
- A history of the search strategy and results
- A thesaurus with the operators NARROWING and BROADENING

As stated in the definition of the CCL [ISO88a], these functionalities represent not all the possibilities on each particular service; it is merely a subset. On the other hand, the CCL does not guarantee all its functionalities to be applicable to all On-line services and their databases. HAWKS provides an end-user with two possible ways of using the CCL part of HAL:

a) A textual mode:

The end-user is to type all CCL commands. This is like being connected to the On-line service by means of a dumb terminal. Before translating the command into native language and sending it to the service, HAWKS will check the use of functionalities according to the current context, and signal the end-user any errors resulting from using a CCL functionality which is not applicable to the current service and/or database.

b) A graphical mode:

End-users not familiar with the CCL language are guided, according to the current context, while constructing their commands. Not only does this mode prevent errors as signaled in the textual mode, but also it provides the user with knowledge about the current database structure, such as possible indexes or possible print formats. Once the end-user validates his command, the system will generate the appropriate CCL command, and provide it in textual mode, hence having an educational aspect.

Both modes can be used at any time during an end-user session. The control on functionality constraints is done using the GD model.

- *The Interpreted Applications*

Any application build with HAWKS will present itself as being a command of HAL. This means that end-users do not have to realize the fact that they are calling an application, but can consider the command as being alike to all the primitive commands. These commands can be issued in both textual and graphical mode, and take parameters. We will go deeper into the use of parameters below. These applications can be context dependant, i.e. they will generate a local error in textual mode, or not present themselves in graphical mode, whenever the current context is not justified. Take for example an application which calls for a sophisticated sorting of records on the on-line service; it will only be applicable on the services providing this functionality.

- *Locally executed commands*

These commands can be divided into two subgroups:

- a) End-user commands:

These are the commands that are not trivial for On-line database searching, but aid the end-user in his work. We state some of them:

- Saving of received data on the workstation in some specific format
- Automatic recording of scenarios, which can be "replayed" at some other time
- Cut and past facilities with other applications

- b) Application developer commands:

Commands available for application development. A priori they are not available to the end-user. We will categorize these commands in detail in Section 4.2.4.

- *Native language commands*

The CCL language does not provide all existing functionalities on On-line services; this being a direct result from the fact that it should be applicable to a wide range of services and therefor is a subset of functionalities. Apart from the available databases on a service, these On-line service specific functionalities distinct one service from an other. If HAWKS was to offer only the above mentioned groups of commands, some existing, sometimes really powerful, functionalities on OIRS would be no longer employable by the end-user. Thence, the use of the service's native language can be mixed at any time with the above defined commands. The native language is recognized by HAWKS and checked with the current context. Syntax checking, a lex and yacc implementation, is performed locally in order to prevent the, often so cumbersome, errors generated by the On-line service.

4.2.4. The Application Development Environment

So far, only the end-user point of view has been presented. In this section we will look at the toolkit for the development of interpreted applications.

For the application developer an application presents itself as a MACRO. A macro is a sequence of commands which are interpreted by

HAWKS. All the commands from HAL can be used to build a macro. The following example is a macro which connects to the wanted database, searches for all records about cardiovascular agents, and downloads the first 10 records:

```
connect Questel
base wpil
search cardiov! adj agent
show 1-10
disconnect
```

The fact that an existing macro presents itself as a HAL command, implies that a macro can call another, existing, macro and pass parameters to it (recursion is prohibited and tested for). In this way, an application programmer is able to layer his macros, and build its own libraries. Inside macros, variables can be used. These variables can be given an initial value by assigning them or by passing parameters to the macro. If a variable is referenced by the interpreter, and its value is not yet known, the end-user will be prompted to provide a value. Variables are all of type string, but interpreted, there where possible, as numbers (real or integer).

In 4.2.3 we already made reference to the locally executed commands for the purpose of macro building. They can be distinguished in the following groups:

- Variable affecting
- Control structures:
These are the standard control structures found in any interpreted language: WHILE, IF-THEN-ELSE and GOTO
- Data selection and extraction:
If one wants to build powerful macros, one can not rely on the consecutive execution of HAL commands only. At some stage it should be possible to look at the received data and act upon accordingly. One might even think that the macro is to continue while using earlier found data (i.e. cross-searching). HAL provides an application programmer with primitives to search and retrieve specific data in the received information, and affect this data to variables. In this way this data can be used later in the macro.

The following example prompts the end-user for a search term and downloads all records if there are less than 20 answers. If more, the end-user will be prompted to indicate the number of records he wants.

```
search $search_term$
cut("items retrieved:", $records$)
if $records$ > 20
then show 1-$show_many$
else show 1-$records$
```

Applications can be context dependant, according to the application programmers intentions. The appropriate context is a result of the used language inside a macro: the use of native language binds the macro to a specific On-line service, the use of some operators can bind the macro to a specific database, whilst the use of the data selection primitives refers to a specific database structure. Generally speaking, a context independent macro uses no more than the CCL-, variables affection- and control structure commands. It is the programmers responsibility to type his application for the proper context. Once typed, an application will be stored and handled by the GD.

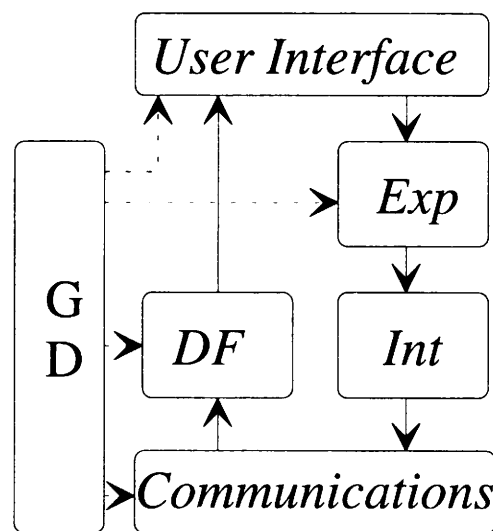


Figure 1: HAWKS global architecture

The application programmer has the following tools to build his application:

- A dedicated macro editor accessible through HAWKS
- A syntax checker:

The integrity of the macro vis-à-vis the associated context is checked as well as the syntax of the used commands without running the macro.

- A debugger:

The debugger enables the programmer to execute his macro step by step, debug called macros, see the contents of the variables, insert additional commands, jump to another command of the macro etc.

4.3. The Architecture

In Figure 1 the global software architecture of HAWKS is depicted.

6 main modules have been conceived:

1. The User Interface

It does what its name suggests: interface HAWKS with the end-user and the application programmer. The interface was conceived according to OSF/MOTIF.

2. The Expander

Before a macro is executed, all the internal macro calls are resolved and expanded. A variable environment is created and all the commands are tested against the current context.

3. The Interpreter

Each command is interpreted in sequence. If it concerns a command issuing a communication, the next command will be executed only after the reception of the On-line service's response.

4. The Dataformatter

This module handles the received information. It searches for specific situations in the information, and is capable of indicating

what actions are to be taken in what situation. These actions are so called "system macros" and are interpreted by the interpreter. Before doing so, the interpreter, puts his actual environment on a stack and pops it when finished. All this is done transparent for the end-user. The Dataformatter is aware of the functionality asked to the OIRS, and will restructure the incoming information according to the GD contents. In this way each HAL command has a unique return value, independent of the connected service. Containing all the search mechanisms on the received information, the Dataformatter is the module responsible for the data selection primitives used in macros.

5. The Communications

This module provides a transparent access to a service up to the data transfer level. The underlying protocols may be: asynchronous, X25 or ISDN.

6. The Global Dictionary

This module has links with all (except the Interpreter) other modules. It is according to the current context constraints that the modules perceive the information needed. Its link with the Userinterface is to enable the context dependant menus, with the Expander to verify the use of commands (macros), with the Dataformatter to be able to recognize the proper situations in the received data, and with the communications to indicate the type of connection needed and the associated parameters.

4.4. Hardware Requirements

HAWKS (Homogeneous Access WorkStation) is being developed, at the TELESYSTEMES' department INNOVATION in Paris. The main target environment is a 386/486 machine under SCO SYSTEM V, and running the X11.3 server. HAWKS will be able to connect to the On-line services, through divers communication mechanisms: asynchronous communication using a HAYES compatible modem or ISDN board, and by 1992, X25 using a direct access on the national PSDN (Packet Switched Data Network), and X25 on ISDN.

5. Conclusions

From the start of the project, it was clear that the context dependency of certain commands and operators was of great importance. By means of the Global Dictionary, with its conceptual model and its persistent environment, we have created a tool which enabled the handling of such contexts. With the aid of the CCL, a unique interface to a large category of OIRS is constructed. With HAWKS, interpreted applications, can be build without having to consult (external) programmers.

6. Future Objectives

The Global Dictionary will become available as a LAN server based upon TCP/IP. The restructuration of the documents actually found on the OIRS and downloaded in HAWKS will be done according to the DAP Q112 of the ODA standard. This will enable the retrieval of Group 4 scanned images on certain services.

References

- [Col88a] N. Collart and M. Joris, *Etude et pratique d'extensions du model Entité-Association*, Institut d'Informatique – Facultés Universitaire Notre-Dame de la Paix, Namur (1988).
- [ISO88a] ISO, *Draft International Standard ISO/DIS 8777*, International Organization for Standardization (1988).
- [TOO89a] TOOTSI, “Application requirements,” Deliverable T.R.SA.12.1, ESPRIT Project 2109, TOOTSI (1989).
- [TOO90a] TOOTSI, “The Global Frame Definition of the Global Dictionary,” Deliverable T.R.SA.18.1, ESPRIT Project 2109, TOOTSI (1990).
- [UKO89a] UKOLOG, *Quick Guide to On-Line Commands*, 1989.

Virtual Swap Space in SunOS

Howard Chartock Peter Snyder

Sun Microsystems, USA.

howard@sun.com peter@sun.com

Abstract

The concept of swap space in SunOS has been extended by the abstraction of a "virtual swap file system", which allows the system to treat main memory as if it were backing store. Further, this abstraction allows for more intelligent choices to be made about swap space allocation than in the current system. This paper contrasts the existing mechanisms and their limitations with the modifications made to implement virtual swap space.

1. Introduction

SunOS provides a unified set of abstractions and interfaces for interacting with virtual memory (VM) facilities both from user programs and from within the kernel. One of the major features of this architecture is that address spaces are constructed out of mappings to virtual memory objects. These mappings cause main memory pages to cache the contents of the virtual memory objects and each physical page is named by the VM object that backs it [Gin87a].

One common type of virtual memory object is an ordinary file. Establishing a mapping to a file makes its contents directly accessible at the address of the mapping. For example, the kernel creates a mapping to an executable file in order to execute it; mappings are also used by applications to access file contents without the copy overhead inherent in the read and write system calls.

A second commonly mapped VM object is known as *anonymous memory*. This term is used because, unlike file mappings, the names of the backing objects are unknown to the client. Anonymous memory mappings are backed by swap space; each physical page in the mapping is randomly assigned a name from the system's pool of available swap space at the time the page first comes into existence. The system uses anonymous memory for several purposes: for private copies of data created during copy-on-write faults, for process data and stack segments, and as a storage resource for the *tmpfs* file system [Sny90a].

Although the existing anonymous memory scheme provides a service that is both powerful and flexible, it has some significant limitations. One is that physical backing store must always be reserved for anonymous memory mappings, even if the client's environment or application doesn't use it. The system requires backing store for anonymous memory so that page frames can be written out and reused

when memory contention increases. Thus, for example, to run an application with a large data segment the system must be configured with lots of swap space, even if pages of the segment will seldom need to be written out to backing store.

A second limitation is that the algorithm for associating backing store with an anonymous memory page is, while very simple, limited and inflexible. The backing store for an anonymous page is chosen at random from the pool of available store when the page is first accessed, and can never be changed afterwards. If the backing store could be chosen later or dynamically moved to different locations, a more intelligent method could be used to make allocation decisions. Such decisions could, for example, employ information about page usage patterns to choose backing store locations that would increase I/O performance during paging and swapping.

To address these issues, the concept of *virtual swap space* was introduced into SunOS. Virtual swap space provides a layer of abstraction between anonymous memory pages and the physical store that may back those pages. The system's virtual swap space is equal to the sum of all its physical (i.e. disk-backed) swap space plus a portion of the currently available main memory. Because virtual swap space does not necessarily correspond to physical storage, the need to configure a system with large amounts of physical swap space can be significantly reduced. Also, because virtual swap space sits "in between" anonymous memory pages and physical swap space, the virtual swap space object can make more flexible and dynamic decisions than in the current system, about what physical backing store to allocate for a page.

To implement the concept of virtual swap space, a new pseudo-filesystem type called *swapfs* was added to the system. Swapfs is a pseudo-filesystem because it does not actually control any physical storage. Rather, its purpose is to provide names for anonymous memory pages. Whenever a part of the system, for example the pageout daemon, invokes a file system operation on a page named by swapfs, it gains control of the page. This gives swapfs great flexibility in deciding the page's fate; for example, at this time it may, if it chooses, change the name of the page so that it is backed by real physical store.

This paper describes the existing anonymous memory mechanisms in SunOS and the modifications that were made to them to implement the virtual swap space abstraction.

2. The Existing Implementation

The current anonymous memory mechanisms consist of an *anon layer*, which provides anonymous memory services to the rest of the kernel, and a *swap layer*, which provides backing store services to the anon layer. These layers interact heavily with file system objects, the page layer, and with mapping objects that employ their services.

2.1. Vnodes and Pages

In SunOS, the *vnode*[Kle36a] is the fundamental structure used to name file system objects. Vnodes provide a file system independent abstraction that allows access to the data comprising a file object. The

vnode object provides a variety of methods for manipulating file objects, some of which are used to interact with the VM subsystem.

SunOS partitions main memory into a number of *page frames*, each of which has a corresponding *page structure* which describes the page frame. Each main memory page in the system is named by the backing store for that page. The name of the page, as stored in its associated page structure, is a (vnode, offset) pair; this name describes the location of the page's backing store.

SunOS uses main memory as a cache of file system objects. The vnode object provides the principal mechanism for manipulating this cache. The VM subsystem performs file system-related operations on a page using the *putpage()* and *getpage()* methods of the vnode which names the page. The *putpage()* operation causes a specified page to be written to backing store. The *getpage()* operation returns a page with a specified name; this may entail allocating a page frame or reading the page data off disk.

2.2. Mappings

A process's address space is composed of a number of mappings to virtual memory objects. Each mapping is represented by an object referred to as a *segment*. The most important service a segment provides is the handling of faults on an address within the segment. The segment is responsible for resolving a fault, if necessary, by filling a physical page from the backing store that the segment maps.

Several different types of segment objects exist in the current system. Perhaps the most heavily used is the vnode segment type (*segvn*), which provides both shared and private mapping to files [Mor88a]. A shared mapping always writes the current data of its mapped object. The same is true for a private mapping except that when it is first written, the VM system's anonymous memory facilities are used to create a private copy of its backing object. Finally, the vnode segment type may be used solely to map anonymous memory; for example, the *exec()* system call creates a vnode segment in this manner to provide a stack segment for a process.

2.3. Anon Layer

The system provides anonymous memory services through the *anon layer*. The anon layer provides operations that allow the client to reserve and unreserve anonymous backing store, to allocate or release a page of anonymous memory, and to create and fill a page backed by anonymous storage.

The system enforces the policy that clients must reserve anonymous backing store up front and only thereafter can allocations for individual pages be made against this reservation. The reservation policy guarantees that, in the face of insufficient anonymous memory, a process will fail synchronously (on return from a system call such as *exec()* or *mmap()*), rather than asynchronously (on a failure to resolve a fault due to a lack of backing store). Reservations are made against the total pool of physical backing store that has been added to the system as swap space.

A segment typically allocates backing store the first time a fault occurs on a page within the segment. The segment calls the anon layer to request a single page-sized unit of backing store from the pool of phy-

sical swap space and obtains an opaque handle for the allocation, an *anon slot*.

The anon slot contains the name of the backing store which will become the name of the associated anonymous page. In the current system, the name is implied by the memory address of the anon slot data structure. This tight binding makes it quite difficult to change the backing store for a particular anonymous page while that page and its associated anon slot are in use.

An anonymous memory client often wishes to use several anonymous pages. Typically the client stores the anon slots for these pages in an *anonmap* structure which contains an array of slot pointers.

The client passes an anon slot to the anon layer to perform subsequent operations on its associated anonymous page. For example, to get the main memory page associated with an anon slot, the client invokes the anon layer's *getpage()* operator using the anon slot as an argument. This *getpage()* operator, similar to that provided by vnode objects, first translates the anon slot into the name for the backing store. With this name in hand, the anon layer calls the underlying vnode *getpage()* operator to acquire the page.

2.4. Swap Layer

The *swap layer* manages pools of physical swap space and allocates and de-allocates page-sized units of space on demand from the anon layer. Swap devices can be added to the pool of available swap space using a system call interface. When a device is added to the pool, the total amount of swap space available for reservation is increased by the size of the device. At this time the swap layer also creates an administrative data structure for the swap device, called a *swapinfo* structure, which includes an array of anon slots. When the anon layer requests a new unit of backing store, the swap layer consults the *swapinfo* structures for the configured swap devices and returns a free anon slot. The position of the anon slot in the *swapinfo* array matches its offset into the swap device.

2.5. Summary

To summarise the above discussion consider the example in Figure 1 in which a process is executing and accesses a byte in the second page of its data segment.

When the process issues an *exec()* system call, the system creates a vnode segment backed by anonymous memory (the "anon client" in the figure) for its data segment and reserves an amount of anonymous memory equal to the size of the segment. A fault occurs when the process touches a byte in the second page for the first time. The page fault handling code directs the fault to the appropriate segment, in this case a vnode segment, which then calls the anon layer to allocate a page of backing store. The anon layer in turn calls the swap layer, which allocates the store (in this case from the swap device named by vnode "swapvp") and returns an anon slot for it, corresponding to offset *off2* on the device. The anon layer translates the anon slot to the name of its backing store, creates a page with this name, and zeroes the page before returning it to the segment. The segment stores the returned anon slot in its *anonmap* for future use. At some later time the system may call the *swapvp putpage()* operator to push the contents of the

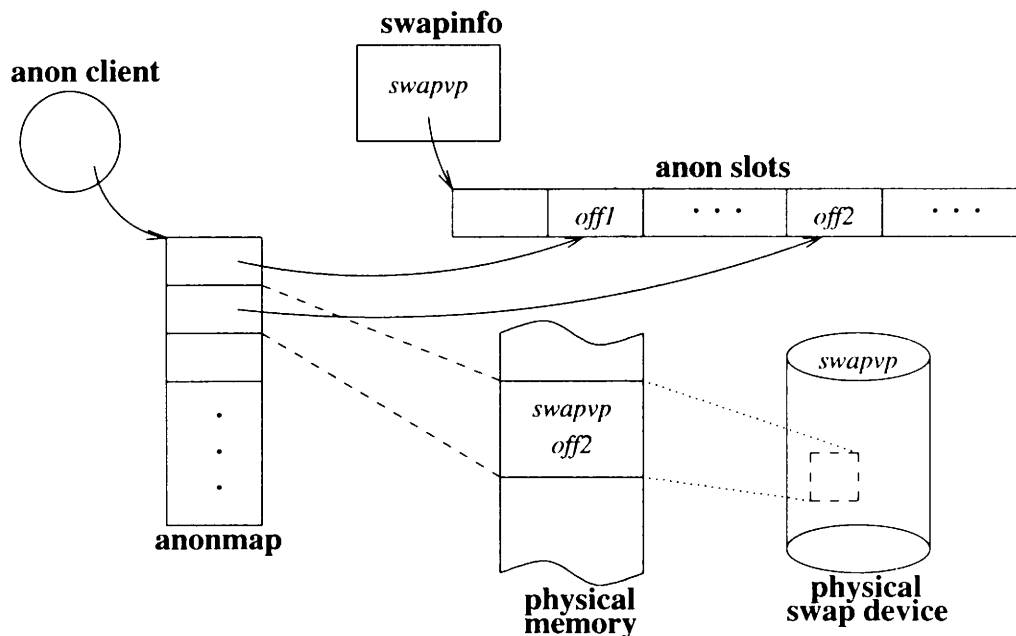


Figure 1: Existing swap/anon layers

page out to its backing store on the physical swap device associated with *swapvp*.

3. The New Implementation: Virtual Swap Space

To address the limitations of the current anonymous memory scheme discussed in the introduction, the concept of *virtual swap space* was developed. Virtual swap space is an abstraction presented by the swap and anon layers to clients of anonymous memory. This abstraction causes anonymous memory pages to appear to the rest of the system as though they are backed by real swap space when in fact they are not.

As discussed above, vnode objects are used to name pages and associate them with backing store. Thus, to implement the notion of virtual backing store, a new vnode type, *swapfs*, was created to present virtual backing store to clients of the anon layer. Swapfs can hand out names for pages just as can any other file system type. However, unlike those of other file systems, these names do not correspond to real physical storage. Instead, they allow swapfs to gain control over the fate of the page when other parts of the system invoke file system operations on it. At such times swapfs can make decisions about renaming the page to be backed by real physical store.

The following discussion explains the changes in structure and mechanism used to implement the abstraction of virtual swap space.

3.1. Anonymous Memory Reservation

In the current system, a client of anonymous memory must reserve backing store up front before it attempts any allocations. The presence of swapfs in the system does not remove this requirement. However, swapfs does expand the pool of reservable swap space to include not just physical devices, but available main memory as well.

With swapfs, the total amount of reservable swap space becomes equal to the sum of all the space on the physical devices in the swap pool plus

the amount of currently available main memory minus a safety factor. The ability to use main memory as allocatable swap space in conjunction with swapfs's ability to dynamically rename anonymous pages, allows users to run with reduced physical swap space and yet still have pages written out when needed.

The amount of available memory in the system is all that is not considered "wired down".[†] Wired down memory includes space that the kernel has dynamically allocated for internal data structures as well as pages a user process has locked via the `mlock()` interface. Similarly, when swap space is reserved against main memory, this memory is also wired down, as the pages cannot be paged out.

The new reservation algorithm always reserves against physical swap space first. Only when this has been completely exhausted is the available main memory used. Regardless of how much main memory is available, the anon layer is never allowed to reserve a certain quantity (computed as a fixed fraction of the total main memory) for swap space. This policy insures that there will always be adequate memory for kernel data structure allocation. Once the anon layer begins to reserve against main memory, anonymous pages may be created in the system for which no physical backing store is available; these pages will be named by swapfs. If swapfs is unable to find physical store for such pages, they will effectively be unpageable; i.e., this memory will be unavailable to the rest of the system, until physical store is freed up.

When a swap space reservation is released, any main memory reserved by the anon layer is released first, and only when all main memory so held has been released, does the algorithm begin to release physical swap space. The algorithm works this way on the assumption that it is desirable to take main memory away from the system only as a last resort and to give it back at the earliest opportunity. Note that a reservation does not reserve a particular chunk of swap space, it simply guarantees that some is available somewhere in the total pool.

3.2. Allocating Backing Store

As discussed above, a client that has reserved swap space calls the anon layer to allocate a page-sized unit of backing store, and the name of that storage is returned to the client in the anon slot. In the current system, this allocation was made at random from the pool of available physical space, but in the new allocation scheme, it is always returned from swapfs, giving swapfs initial control over all anonymous pages in the system.

Thus, all clients of the anon layer receive backing store names from swapfs. Only swapfs itself requests real physical backing store from the swap layer, which it does when it wants to rename an anonymous page to physical store.

In the current system, the anon slot is returned by the swap layer with the name of the backing store implied in the memory address of the slot data structure. However, binding the name to the memory address of the data structure makes it quite difficult to change the name while the anon slot is in use. The anon slot is held by a client as long as the client makes use of the associated anonymous page; thus, if swapfs is to change the backing store while the page is in use, it must also change the name to which the anon slot refers.

[†] Physical memory that is "wired down" is not pageable, and hence not available for use by the rest of the system.

To facilitate renaming, a level of indirection was added to the anon slot. The name of the backing store associated with the slot is now stored as *vnode* and *offset* fields in the slot data structure, allowing the values of these fields to be changed while the slot is in use. The swap layer no longer hands out anon slot data structures from per-swapinfo arrays; instead, the anon layer creates them on demand. The swap layer also no longer uses anon slots to track allocations; instead, it employs a bitmap of free slots for each swap device. The use of this bitmap solves a problem with current versions of SunOS, which keep unused slots on a freelist for later reuse. The algorithm for ordering this list can sometimes lead to poorly ordered swap allocations, which in turn cause poor paging performance.

3.3. Swapfs

The virtual swap space abstraction is implemented by the swapfs pseudo-filesystem type. Pseudo-filesystems have been used before to layer a new abstraction on top of what appears to the rest of the system as a file system [Ros90a]. For example, *tmpfs* layers a filesystem on top of anonymous memory.

As is true for any other file system object, swapfs presents a set of methods for manipulating its objects. In constructing this new file system type, we were concerned only with its use within the kernel by the VM system. Thus, we did not build a full set of file operations which would, for example, allow a user to mount a swapfs file system, or give names to swapfs files. In fact, swapfs provides only three significant operations: *swapvp()*, which returns a vnode to the swap layer for use as a swap device; *getpage()*, which returns a page to the system backed by swapfs; and most importantly *putpage()*, which “pushes”[†] out an anonymous page backed by swapfs.

3.3.1. Swap Layer Modifications

Swapfs provides only one vnode to the system which is made available to the swap layer through the *swapvp()* operation when the system boots. For the most part, the swap layer treats swapfs as it does any other swap device. It is added to the list of swap devices managed by the swap layer and an administrative data structure, including an allocation bitmap, is created for the vnode.

Special-case modifications to some swap routines were needed to reflect some of the differences between swapfs and other swap file system types. For example, the interface routine called to allocate swap backing store, has been modified to take flags specifying where the backing store is to come from. As discussed above, all external clients of anonymous memory allocate backing store via the anon layer, which always requests this store from swapfs. Only swapfs itself makes allocation requests for physical swap space, when it wants to change the backing store for a given page.

3.3.2. *getpage()*

When a process faults on an anonymous page, the fault is directed to the segment that maps the address. On the first fault, the page may not exist, and there may be no backing store. In such a case, the segment calls the anon layer to allocate backing store. The backing store’s

[†] For most file system types, “pushing” out a page typically entails writing it out to its backing store, marking it as “clean” (i.e. unmodified) and adding it to the free list of pages for re-use.

name is extracted from the returned anon slot. The *getpage()* operator on this vnode is then called to create the page. Because client allocations of anonymous backing store are always satisfied by swapfs, this will result in a call to the swapfs *getpage()* operator. Swapfs satisfies this request by simply creating a page with the requested name and handing it back to the segment. Unlike some other vnode objects, swapfs has no need to perform additional functions at this point, such as allocating physical disk space.

When a subsequent fault occurs on the same address, the fault will find its way to the same segment, which will again use the name in the anon slot to call the *getpage()* operator to retrieve the page. Most vnode *getpage()* operators have to be prepared to deal with the fact that the page may no longer be in memory. In this case they must create a new page and do I/O to fill it. Swapfs pages, however, once modified by the client, remain in memory unless renamed and paged out.

When a page has been modified, the system marks it "dirty". The system will not attempt to reuse the page until it has been marked "clean", and this is completely under the control of the vnode that owns the page. Typically a vnode marks a page clean in its *putpage()* operator when it pushes the page out to backing store. However, because swapfs represents virtual swap space, it does not page out pages itself and hence never marks them clean. Swapfs pages out pages by renaming them to be backed by real physical store, and the vnode to which the page is renamed then becomes responsible for cleaning the page.

Modified pages will never be reused by the system as long as they are named by swapfs. Because of this, when the swapfs *getpage()* operator is called, it knows that if a page has been modified it will be able to find it in the page cache. If the page has never been modified, then it may have been reused, but in such a case the client cannot possibly care about its contents, so swapfs simply creates a new one and returns it.

3.3.3. *putpage()*

At certain times the system will push pages to their backing store. This may happen, for example, when the pageout daemon pushes pages out to try to accommodate memory demands on the system, or it may happen when the scheduler decides to swap out a process. The pageout function entails getting the name of the page and calling the *putpage()* operator of the associated vnode. For swapfs, this operator does some very unusual things. Swapfs makes a call to the swap layer and asks it to allocate a page of physical backing store. If this request fails, nothing further is done with the page; it remains modified and simply stays in memory. However, if the swap allocation succeeds, swapfs renames the page to the new backing store and then calls the *putpage()* operator associated with the new store to actually push the page to disk.

This operation must be performed with great care, because during the rename other parts of the system may be accessing or trying to access the page. It is perfectly legitimate for a process to be reading or writing the page during the rename. For example, the existing system already allows a user to access a page while the page is being written out. However, changing the name of the page or its corresponding anon slot is not allowed while I/O is in progress or while clients are using the old name to try to find the page. The page layer provides locks to protect against this happening to the page, however, the name of the page is stored in the anon slot as well. Thus both the anon slot and the page must be renamed atomically to insure that a client will not use the wrong name to get to the page while the rename is in progress.

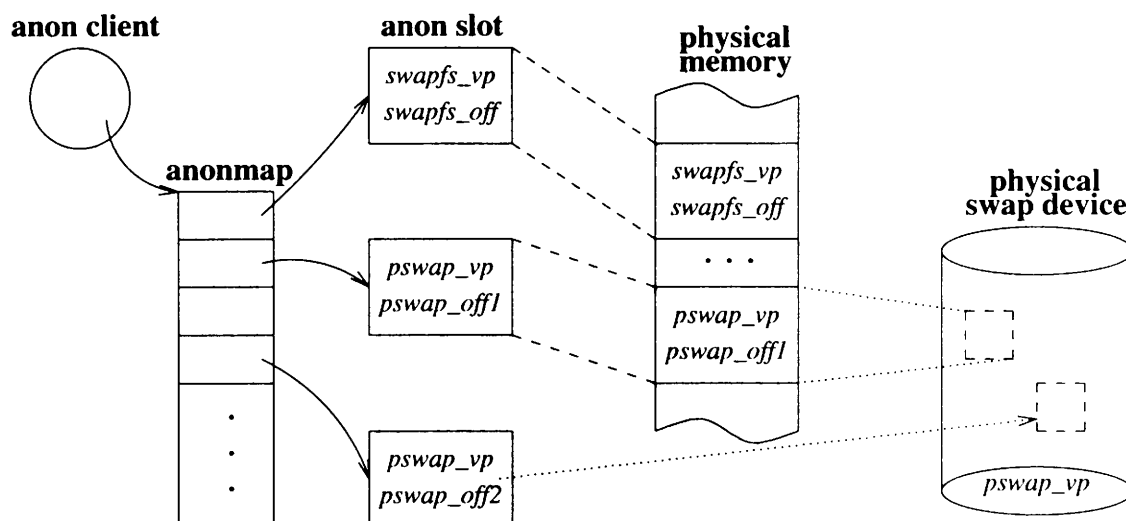


Figure 2: Swap renaming

The anon slot contains a lock that protects its backing store name fields, among other things. To guarantee atomicity, both the anon slot and the page are locked, both are renamed, and then the locks are released. Any process trying to manipulate the page in the kernel will acquire one or both of these locks first and will thus be guaranteed a consistent view of the name of the page and its associated anon slot.

3.4. Summary

To summarise the interaction of swapfs with the rest of the VM system, consider Figure 2 which shows some different states of anonymous memory. The first page in the figure (named *swapfs_vp*, *swapfs_off*) has been allocated and named by swapfs. Note that because it is named by swapfs it is memory resident and has no physical backing store.

The second page in the figure (named *pswap_vp*, *pswap_off1*) shows an anonymous page that has been renamed to a physical backing store location. This page was originally named by swapfs, until the *putpage()* operator was called to push out the page. At this time swapfs called the swap layer, which allocated a page with backing store on swap device *pswap_vp*, at offset *pswap_off1*. This new name was returned to swapfs, which renamed the anonymous page and the slot to (*pswap_vp*, *pswap_off1*) and then called the *putpage()* operator of *pswap_vp* to write the page to disk.

The third page (named *pswap_vp*, *pswap_off2*) has also been renamed to physical backing store and the corresponding main memory page has been freed for another use in the system. The data for that anonymous page may be retrieved via the *getpage()* operator for the physical swap device.

Note that in contrast to Figure 1, anon slots are no longer explicitly tied to a particular swap device and that the name of the backing store is now stored explicitly in the slot.

4. Performance Discussion

The initial performance goal for this project was that swapfs not degrade overall system performance. In particular we required a system running swapfs with as much physical swap space as the current SunOS default to perform as well as the existing SunOS implementation. Another concern was that configuring a system with small amounts of physical swap relative to the amount of main memory might degrade overall performance as memory became clogged with anonymous pages that could not be freed for reuse. However, it was a pleasant surprise to find that, even with a small percentage of physical swap relative to main memory (as little as 20%), there is very little degradation in system performance on standard benchmarks. This may be explained by the observation that, for a variety of workloads, the system tends to allocate only about 2/3 of the swap space it reserves; thus, even if all of this space is allocated from main memory, a substantial remainder is available to the rest of the system.

5. Future Work

Adding swapfs to the system opens the way to some interesting enhancements. For example, the current pageout daemon pushes out pages one at a time, and many of these pages are backed by swapfs. Slight modifications to the swapfs rename mechanism would provide the capability to rename a batch of pages to a continuous stretch of physical backing store; then all the pages in the batch could be pushed out in a single I/O.

Another interesting enhancement involves the notion of multiple swap vnodes. Although swapfs currently provides anonymous memory names from only one swap vnode, simple modifications to the existing interface could allow each client (e.g., a process) of the anon layer to have its own unique swapfs vnode; backing store allocation requests from a particular client could then always be satisfied from the swapfs vnode belonging to that client. This, in turn, could provide a basis for client-oriented clustering of anonymous pages on physical backing store.

6. Conclusions

Swapfs is a virtual swap file system that can, on demand, dynamically rename anonymous pages and push them out to physical backing store. This capability allows the system to treat main memory as swap space, but also allows it to reclaim this memory by pushing pages to available physical swap space when contention for memory increases.

7. Acknowledgements

We would especially like to acknowledge Bill Shannon who contributed many of the ideas that are central to this work. He and other members of the Systems Group at Sun including Glenn Skinner, Anil Shivalingiah, and Dock Williams, also provided many helpful suggestions.

References

- [Gin87a] Robert A. Gingell, Joseph P. Moran, and William A. Shannon, "Virtual Memory Architecture in SunOS," *Proceedings of the Summer 1987 Usenix Technical Conference*, Phoenix Arizona, USA, pp. 81-94, USENIX Association (June 1987).
- [Kle86a] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File Systems Types in Sun UNIX," *Proceedings of the Summer 1986 Usenix Technical Conference*, Atlanta Georgia, USA, pp. 238-241, USENIX Association (June 1986).
- [Mor88a] Joseph P. Moran, "SunOS Virtual Memory Implementation," *Proceedings of the Spring 1988 EUUG Conference*, London, England, EUUG (Spring 1988).
- [Ros90a] Davis S. H. Rosenthal, "Evolving the Vnode Interface," *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, California USA, pp. 107-117, USENIX Association (June 1990).
- [Sny90a] Peter Snyder, "tmpfs: A Virtual Memory File System," *Proceedings of the Autumn 1990 EUUG Conference*, Nice, France, pp. 241-248, EUUG (October 1990).

The Art of Automounting

Martien F. van Steenbergen

Sun Microsystems Nederland B.V.

Amersfoort, The Netherlands

Martien.van.Steenbergen@Holland.Sun.Com

Abstract

Who isn't familiar with the problem of installation and management of application software, users, home directories, etc. The strive for a consistent, scalable, efficient and simple working environment has our continuous attention.

Problems are known to be caused by architecture dependencies, no distinction between various classes of data, disk partitions that fill up and disk fragmentation. Furthermore, various variants and versions of the same application software, different locations of data also cause headaches sometimes.

All this and more is covered in this paper, leading to a "Plug and Play" computing environment that leaves system and network administrators time to do more important things.

1. Introduction

Wouldn't it be nice if, when working with computers on a day to day basis, we had a general, uniform and consistent computing environment, especially with respect to the file system layout. Furthermore, we would also like this environment to be scalable from stand-alone systems to large heterogeneous environments.

Besides that, one would also like to be able to use the available disk space in the network as efficient as possible, avoiding fragmentation of this valuable resource. It is very frustrating to discover that you cannot save your file because a file system is filled up, while on other places in the network more than 1 GB appears to be available!

Also, being able to find what you are looking for in the file system on an intuitive fashion would reduce confusion in your user community. Separating the various classes of data available on the network and providing this data in a logical and consistent way reduces confusion and requires less help from the system administration department. A complete separation between private data (stored in home directories) and more project related data would reduce the need to snoop around in someone else's home directory. After all, most of us also don't like others to peek and poke in our private drawers, let alone suitcases.

The frustration of having to change your command search path every time a new (version of a) software application is installed, is causing

headaches sometimes. Or, even worse, the search path depends on the underlying hardware architecture, e.g. Sun-3, Sun-4 and Sun386i.

All this and more is covered in this paper. Just by setting up some conventions and guidelines about where to put what and why is 90% of the job. And please note that setting up the system described here does not require any additional software to be bought or installed. You can just use the tools and technologies available at your fingertips right now.

Three main objectives that have been the most important during the definition of the guidelines are the same as those used for the design of the OPEN LOOK GUI: *simplicity*, *consistency* and *efficiency*.

- **Simplicity**

Conventions and guidelines must be simple to understand and implement. Applying those guidelines must also be very simple. So simple, that even non-administrators have a clear picture of what is going on. So simple, that even non-administrators can execute the steps required to perform a certain operation with success.

- **Consistency**

Having a consistent environment for both end-users and system administrators will alleviate their work. If the same concepts apply in all circumstances, this will greatly enhance the overall quality and understanding of the environment, and concepts once learned can be applied everywhere.

- **Efficiency**

Besides being simple and consistent, the system must also be efficient to use and apply. The overall performance and usability of the system is not allowed to suffer too much from the first two objectives. This means that in some cases exceptions have to be applied in order to ensure better performance.

Initially, the first two goals mentioned above are the most important and this paper will focus on those. However, research has shown that if these rules are strictly applied, performance loss may result. In general you could say that in order to improve performance, you should try to minimize the number of symbolic links that you use.

Besides the general concepts and guidelines explained in this chapter, it will go into some specific requirements that apply when installing or building applications that nicely fit in the frame work. It details about resolving architecture dependencies, different versions and variants of the same application etc.

A more complete publication called "The Art of Automounting" is also available from the author. This publication also contains step by step procedures that you need to perform in order to implement the concepts described in this paper. As an example, the complete and detailed building and installation of GNU Emacs is described. Furthermore, a section on "Choosing a name for your computer" helps you picking the right names. And, finally, it includes a section that serves as a primer on how to "Treat system management as a real project".

In order to obtain a PostScript version of this document, simply send your request via email to Martien.van.Steenbergen@Holland.Sun.COM.

For more information on performance issues, please refer to the article "NFS Client Server Performance" by Jos van der Meer, Frans Wessels and Maarten Westenberg from Sun Microsystems Nederland B.V.

Who Should Read this Document?

This document is written for both novice and experienced system administrators. It is assumed that you are familiar with UNIX and SunOS in general, the Network File System (NFS) and the Network Information Services (NIS).

These tools and technologies will not be explained in this paper. For more information about SunOS, NFS and NIS, please refer to "System & Network Administration".

2. The Art of Automounting

Organizing your file system so that you always know what is stored where on your disks and why, could be considered as an art. However, once you have done that, you cannot live without it anymore.

This chapter contains the guidelines and procedures that help you set up and structure the data that you have to maintain for your customers – the end users (note that you yourself are also an end user).

A golden rule for successful system management is: *document the conventions that you use and the things you change and the reasons why.*

The Sample Configuration

The sections below are based on the following configuration of computer systems interconnected by Ethernet (Table 1).

Host name: bach Usage: Main application server Type: SPARCserver 2	Host name: chopin Usage: Mirror application server Type: SPARCserver 2
Partition: /xenon Usage: Third party and unbundled software	Partition: /neon Usage: Redundant copy of bach:/xenon
Partition: /argon Usage: Free and locally built software	Partition: /fluor Usage: Redundant copy of bach:/argon, beta software and older versions of software for compatibility
Host name: mozart Usage: Production data server Type: SPARCserver 490	Host name: liszt Usage: Your workstation Type: SPARCstation 2 GX
Partition: /krypton Usage: Home and project directories	Partition: /gold Usage: Very old software and private data

Table 1: Computer Configurations

2.1. The Problem Defined

First of all, what are your most important problems right now?

/krypton: file system full

Probably you run out of disk space somewhere in your network almost every day. Besides that, you know that there is a lot of unused disk space available in the network on the various drives local to workstations that you would like to use.

You will probably also need to change the user's command search path once in a while, in order to make a new application available to them. With tens or even hundreds of users and without some organization, this might become a nightmare.

Looking for the files you need all over the place, and not being able to find them is a major frustration. Even worse, if you have to peek in your colleague's private directories – because you know that he or she stores things there instead of a project or workgroup directory – you might feel uncomfortable and hope that you will not be caught snooping.

Plug & Play

Also, installing (and uninstalling and moving them around) applications, users, systems and other entities on your network is sometimes a hassle. You would probably like to plug and play, meaning that you unpack your new hot box, plug it into the mains and into the network and run! The setup should be so, that users and applications do not suffer from the various underlying hardware platforms and operating systems, users should be able to use them transparently.

2.2. Data Classes and Instances

Most of the problems are caused by the various classes, versions and variants and locations of data. Below, you will find a few examples of data classes and some considerations about how to use them.

Home Directories: /home/user

To start with, home directories are a good example of a distinct data class. Preferably, a home directory looks something like **/home/mary**. This is easy to remember, refers to a real person and does not have any "foreign" data clobbering it.

For instance, in the conventional Sun file system set up, you use home directories of the form **/home/mozart/mary** where **mozart** is the name of the NFS server of the home directory. However, you will have to do a lot of work if you decide to move Mary's home directory to another NFS server. So why don't you leave out this sensitive information in the first place?

This means that in general, home directories have the form **/home/user**, where **user** is replaced with the login name of that user.

Note that if you are planning to implement this concept, you will have to be careful not overmounting the conventional **/home** directory, which is normally used as mount point for the home partition, e.g. **sd0h**.

Project Directories: */project/project*

Another specific data class could be the various projects that are being done in your environment. You should apply the term “project” for a wide scope of activities, not just software engineering projects.

For example, a project that studies NFS performance could be called **nfstune**. Another project that deals with personnel administration could be called **humadm**. Yet another project is responsible for system administration (yes, that’s you), let’s call it **sysadm**.

Having these examples, it would be no more than logical (and intuitive) to use separate project directories for them, and their corresponding names then would be **/project/nfstune**, **/project/humadm** and **/project/sysadm**.

In general you will have **/project/project**.

Applications, Tools, Utilities – Read-mostly Data: */vol/volume*

As a third example, consider the various applications, tools and utilities that you make available for your users. In general, these “things” are used in a read-only fashion. For example, you could support tools like FrameMaker, GNU Emacs, Console Tool, Rolo Tool and Catcher on your network. Also, you want these tools to be available for everyone on the network.

These applications are normally made available through a path name that contains the application’s base name as last part. FrameMaker is then called **frame**, GNU Emacs is called **emacs**, and the others **contool**, **rolotool** and **catcher** respectively.

Following the trend set in the first two examples, you could consider providing these applications through, say, **/applic**, **/tool** and **/util**. However, most of these applications are used in more or less the same way and perhaps even some other data (i.e. non-applications, like include files, icons, FrameMaker document templates) are used in the same read-only fashion. So why not just keep it simple and make them available as volumes through **/vol**: a clod of data that can be treated as one single unit.

To complete our example, you would have directories like **/vol/frame**, **/vol/emacs**, **/vol/contool**, **/vol/rolotool** and **/vol/catcher** respectively.

In general you would have **/vol/volume**.

A major advantage of the single unit approach is that you can move these units around on the network to the most convenient location very easily, while still maintaining the same logical access path. (If you are wondering how? Read on.) Besides that, treating applications and the like as units, means that they do not “infect” your file system on other places than just the installation directory. This means that uninstallation becomes a piece of cake: just remove the installation directory and any references to it. You don’t have to track down any other places in your file system that may have been touched during the initial installation of the application.

Software Distributions: */distrib/distribution*

Suppose your business includes distributing software. For instance, you deal in FrameMaker, you have your own developed software that you want to cut tapes from, you support a few distributions for friends, etc. Consider making these things available through **/distrib/distribution**. For instance **/distrib/frame** (note the difference

between this version and **/vol/frame**), **/distrib/nfstune** (the product that results from the corresponding project discussed above), **/distrib/xview** (the XView source code that you want to be able to give your friends).

Again, these distributions are used in a read-only fashion and they can be treated as units. You could develop scripts that follow this convention in order to create tapes or floppies ready for distribution. You could also consider putting less frequently used distributions on partitions that are normally not used to store production data. For instance, you could use the partitions local to workstations and that are normally not used for other purposes.

And more: /source, /beta...

Using the same concept, you could go on. Supporting **/source** for building and installing public domain and free software, **/beta** if you want to distinguish between general (or "production") applications and beta releases of software that you want to make available.

In general: /class/instance

In general, you will have */class/instance* as a path name to get to a specific instance of data in your network. Note that it is only two levels deep! This is easy to remember, especially if you use the right names and set it up consistently.

You can promote data at the instance level to class level if the need is there. For example, you start out with supporting **/vol/distrib** for software distributions. But after a while, this directory becomes so large and unmanageable that you decide to split it up into separate distributions. This is the moment that you promote **/vol/distrib** to **/distrib**. Of course, the other way around is also possible.

2.2.1. Versions and Variants

Other major problems are the various versions and variants of applications.

For example, at a certain point in time you must support FrameMaker 1.3b SunView (because there are still users that did not have the upgrade training to 2.0), FrameMaker 2.1 SunView (because there are still users who are not running OpenWindows) and FrameMaker 2.1 for the X Windows System, the default.

At the same time, OpenWindows 2.0 is the default windowing environment and OpenWindows 3.0 beta is available for those who cannot wait.

Besides that, you need to support a few free software tools – like GNU Emacs – on multiple architectures, Sun-4, Sun-3 and Sun386i, say.

All of these instances are variations of a functional equal application. How do you organize these things? A very simple approach would be to always let */class/instance* be the default or current version of a data instance. For example, **/vol/frame** corresponds with the FrameMaker 2.1 X Window System version, **/vol/openwin** corresponds with OpenWindows version 2.0 and **/distrib/xview** corresponds with the XView 2.0 source code contribution of Sun to the MIT X Windows System distribution.

Now, if you also want to make non-default versions available and still be able to distinguish between them, add the version number to the default name, separated by a hyphen. So, FrameMaker 1.3b (the Sun-

View version) would then be available via `/vol/frame-1.3b` and `/vol/openwin-3.0-beta` is the beta release of OpenWindows version 3.0.

The hyphen that separates the base name from the version number helps you distinguish between the two, e.g. consider using Lotus 1-2-3 version 1.0: `/vol/lotus1231.0` versus `/vol/lotus123-1.0`.

In order to maintain consistency, you should also support explicit version numbering for the default version. This means that in the example above, you would have both `/vol/frame` and `/vol/frame-2.1` available (which of course both refer to the same installation).

2.2.2. Putting it All Together

The conventions described in the previous section can be realized by using techniques and tools available in SunOS, NFS and NIS. To be specific, it requires both servers and clients to be able to use NFS and NIS and the program that glues them together and exploits their capabilities: `automount(8)`. The automount process, once started from `/etc/rc.local` during boot time, acts like a name to location server. Its behavior can be almost completely controlled by the contents of some specific NIS maps. The next section, "How to set up your system" shows you the details on how to do this.

2.3. How to Set Up Your System

This section gives you a look under the hood. It details on how to partition your disk, exporting and importing data and how to glue it all together using the automounter, NFS and NIS.

This section applies to the general case and does not describe on how to organize your own applications or public domain or free software. For more information on the latter topic, please refer to "Application management".

2.3.1. Partitioning Disks

When partitioning your disks, you should consider whether or not the data that you are going to store in that partition is going to be used in a read-write or read-only fashion. If it is going to be used as read-only in most of the cases, then you will never have to include that partition in your backup scheme. You only need to archive the data once, so that you can recover from loss of data.

Note that installed third party applications and public domain and free software, as well as software distributions, local include files, etc., are used read-only most of the time. Besides that, these installations can total up to tens, even hundreds of megabytes. Megabytes that you do not have to backup! You only have to backup "production" data like home and project directories and perhaps databases.

Another point of consideration is separating your data from third party data. Third party data is data that you get from some other party, for example a new version of the operating system, or a new version of your publishing software, but also a new version of a locally built tool. After having installed third party software, you often have to customize it in order to match your local needs. If you need to customize it, try to avoid changing files in the installation directory, or keep the required changes to a minimum. Every time that you re-install the software you have to re-apply the changes. Instead, try storing those changes in a directory separate from the installation directory and tell the third party

software to look there for specific files, e.g. by setting the appropriate environment variables or by creating the appropriate symbolic links in the installation directory.

As an example, consider the customization of FrameMaker with respect to the use of locally developed document templates. FrameMaker normally looks for document templates in its own installation directory. If you replace the directory that FrameMaker looks in by a symbolic link to a place that you maintain, FrameMaker will use the contents of that location instead. In order to be compatible with the document templates that FrameMaker supplies, you could create a symbolic link back to the, now renamed, original template directory. Although you create two extra symbolic links, you have the advantage that the next time you install FrameMaker, you only have to create one symbolic link, and everything works as before. You don't have to worry about saving your templates before removing the old installation of FrameMaker.

2.3.2. Mounting Local File Systems

After having partitioned your disks and having created file systems in the appropriate partitions, you have to mount them at boot time to make them available to your local system. This is done by means of the file */etc/fstab*.

By convention, the mount points that are used in this paper use element names from the periodic table, like **krypton**, **xenon** and **helium**. The advantage of choosing real names that are unique within your domain or network is that you can uniquely identify a partition (or file system) in the network. There is never any doubt about which partition you mean when you talk about the fact that, say, */krypton* is full again. The use of the name class of elements from the periodic system should provide for enough distinct names in your network, it contains well over 100 element names.

The generic name *element* always denotes one single physical file system. Mapping this physical name to a more logical name is done during the export and import of (parts of) the file system by the automounter.

2.3.3. Subdividing Partitions

Within the local file systems, you can create the subdirectories for the data classes that you want to store on these partitions. For example, if you want to support your applications and other read-only data on **/xenon**, then you should create the directory **/xenon/vol**. And if you want to support home and project directories on a partition called **/krypton**, then you should create the directories **/krypton/home** and **/krypton/project**.

2.3.4. Exporting Data

If you have created directories and installed software, users and projects, and you want to make this data available to the network, you have to export them. If you do not export this data, others cannot import it.

Exporting data can be done at three levels: at the partition level, at the class level and at the instance level. For example, you could export the complete **/gold** partition. You could also export home directories at the class level by exporting **/krypton/home**, for example. Finally, you

could export volumes (applications, etc.) at the instance level: **/xenon/vol/openwin-2.0**.

The level at which you export data depends on the granularity of control you want to have on your data. The lower the level, the more control you have. For instance, if you export volumes at the instance level, then you can specify the export options on the finest level. In this case you can specify for each instance if it is exported read-only, read-mostly, with root access, which clients can access it, and so forth. Root file systems and swap files for diskless clients, for instance, are exported at the finest level. You could also use it in combination with *netgroup(5)* in order to allow or disallow certain groups of machines or users access to specific data.

2.3.5. Importing Data

If you are a client of data provided by file servers on the network, you should follow the conventions defined in the previous sections. This means that if, for example, you need FrameMaker to be available on your system, you have to import that volume from the server. This importing is done by issuing the *mount(8)* command or by specifying the appropriate entries in */etc/fstab*. A sample command to import FrameMaker would look like:

```
mount bach:/xenon/vol/frame-2.1 /vol/frame
```

assumed that FrameMaker is installed on the NFS server bach on a partition that is mounted on **/xenon** and that the local directory **/vol/frame** exists. In general you would use:

```
mount server:/partition/class/instance /class/instance
```

Instead of doing these mounts manually or during boot time, you can exploit the automounter's capabilities to automate this process. How to do this is covered in the following sections.

Please note that NFS servers can be NFS clients at the same time, even of them selves. In order to maintain consistency and simplicity, you should set up your system so, that NFS servers are treated the same way NFS clients are. There should be no difference, except for the fact that servers serve files.

Using the Automounter

Instead of importing or mounting NFS directory hierarchies manually or during boot time, you can use the automounter to automate this process. For a description of how the automounter works, please refer to the manual page: *automount(8)*.

By default, the automounter looks for a map with the name **auto.master** in the current NIS domain. If this map exists, it will consult this map and use this map as a list of initial automount maps (consider it a meta map). The layout of the auto.master map is as follows:

```
/mountpoint mapname [mount options]
```

By convention, all the maps that are specific for the automounter have **auto.** as a common prefix.

For instance, to follow the examples used in the previous section, our **auto.master** could look like this:

```

/vol      auto.vol      -ro,soft,nosuid
/project  auto.project  -rw,hard,nosuid
/home     auto.home     -rw,hard,nosuid
/distrib  auto.distrib   -ro,soft,nosuid
/net      -hosts        -ro,soft,nosuid
/-        auto.direct    -ro,soft,nosuid

```

As you will notice, most of the imports are done read-only, soft and without set UID on execution by default. Importing data read-only is cheaper than importing it read-write, and importing data without the set UID on execution semantics avoids one of the security violations that Trojan horse attacks like to use.

As we will see in later examples, you can override this default on special cases in the appropriate maps.

All of the maps used in the above example will be explained below.

To start with, the contents of the **auto.vol** map could look like this:

```

emacs      bach:/argon/vol/emacs-18.55\
chopin:/neon/vol/emacs-18.55
frame      bach:/xenon/vol/frame-2.1\
chopin:/neon/vol/frame-2.1
bin        bach:/argon/vol/${ARCH}/bin\
chopin:/neon/vol/${ARCH}/bin
man         -rw    bach:/argon/vol/man\
chopin:/neon/vol/man
openwin     -suid   bach:/xenon/vol/openwin-2.0\
chopin:/neon/vol/openwin-2.0
emacs-18.55 bach:/argon/vol/emacs-18.55\
chopin:/neon/vol/emacs-18.55
frame-2.1   bach:/xenon/vol/frame-2.1\
chopin:/neon/vol/frame-2.1
frame-1.3b  chopin:/fluor/vol/frame-1.3b
openwin-2.0 -suid   bach:/xenon/vol/openwin-2.0\
chopin:/neon/vol/openwin-2.0
openwin-3.0-beta -suid chopin:/fluor/vol/openwin-3.0-beta

```

Taking the first entry, **emacs**, as an example, here is how the automounter works. The automounter acts like a NFS file server. In this example, it intercepts any reference to **/vol**. As soon as a process refers to something (**emacs**) under this directory, the automounter searches the corresponding map, **auto.vol** in this case. When found, it will import the directory hierarchy from the location(s) specified in the last column. What it will do exactly in this example is shown in the following commands (assuming the reply to the import request is received from **bach** first):

```

mount -o ro,soft,nosuid bach:/argon/vol/emacs-18.55 \
/tmp_mnt/vol/emacs
ln -s /tmp_mnt/vol/emacs /vol/emacs

```

The **bin** entry will be explained in the section "Application management". Note that the **man** entry will be mounted read-write in order to be able to store formatted pages in the corresponding cat directories if necessary. Also note that **openwin** is mounted with the set UID on execution option turned on in order to support the "MIT-MAGIC-COOKIE" security.

The map used in this example also provides redundant locations for most of the volumes. In this case, they can both be imported from either **bach** or **chopin**. This redundancy can lead to a more reliable and robust environment. This kind of redundancy is of course only useful for read-mostly data.

Finally, note that besides the default version, also the explicit versions are provided for those who need them.

The exceptions to the rules set in the **auto.master** map stand out against the other entries. This gives you an instant overview of those parts that may require special considerations during the installation phase.

The **auto.project** map looks like:

```
nfstune    mozart:/krypton/project/nfstune
humadm     mozart:/krypton/project/humadm
sysadm     mozart:/krypton/project/sysadm
```

Nothing special about this one, except that all projects appear to be served by host **mozart**.

The **auto.home** map looks like:

```
john       mozart:/krypton/home/john
mary       mozart:/krypton/home/mary
graiG      mozart:/krypton/home/graiG
```

Et cetera. You could also consider importing home directories one level higher. This reduces the number of mounts necessary to import more than one home directory from the same base location more than once. The appropriate map then looks like this:

```
john       mozart:/krypton/home:john
mary       mozart:/krypton/home:mary
graiG      mozart:/krypton/home:graiG
```

In this case, when John logs in, **/krypton/home** will be imported from **mozart**. Then, when Mary logs in, only the appropriate symbolic link will have to be created by the automounter, since Mary's directory is already available on the system.

The **auto.distrib** map looks like this:

```
xview      bach:/xenon/distrib/xview-2.0
emacs      bach:/argon/distrib/emacs-18.55
xview-1.0  liszt:/gold/distrib/xview-1.0
xview-2.0  bach:/helium/distrib/xview-2.0
```

In this case, the XView version 1.0 distribution is also supported. However, it could be that this instance is stored on a file server that is normally not used for intensive file server operations. Perhaps it is stored on a disk local to a workstation somewhere in the network, just for convenience and without the need to make backups at the appropriate times (in this case **liszt**, *your* workstation).

The entry that specifies **/net -hosts** is special. It causes a reference to **/net/host** to mount *all* directory hierarchies exported by that host. Please note this can lead to a high system and network load and can take a long time if the host specified exports many directory hierarchies.

The automounter consults the NIS map **hosts.byname** for the host specified on the command line.

Finally, the **auto.direct** map is used to make single directories available in between existing ones. As an example, consider a direct mount of **/var/spool/mail** and **/var/spool/calendar** from a central file server that serves your mailbox and network agenda. The entries in the **auto.direct** map would look like this:

```
/var/spool/mail    mailhost:/var/spool/mail
/var/spool/calendar calendarhost:/var/spool/calendar
```

References to either directory by Mail Tool and Calendar Manager respectively, would cause the appropriate directory to be imported from the server.

As another example, consider a transition phase: you are in the process of switching from the old file system organization to the new one. In the old case, you supported a project that had its directory under `/usr3/zis`. In the new set up you are going to make this project directory available under `/project/zis`. What you can do is support both access paths during the transition phase so that the engineers can adapt their scripts and Makefiles to the new set up.

In order to do so, include the following line in **auto.project**:

```
zis          mozart:/krypton/project/zis
```

and include the following line in **auto.direct**:

```
/usr3/zis    mozart:/krypton/project/zis
```

As you can see, both locations refer to the same project directory.

Another convenience of the automounter is that it will create all directories needed for you. It will also remove them (and the corresponding links) automatically after a certain period of inactivity or when the system is shut down. You do not have to do a thing.

Automount Inconveniences

There are a few inconveniences when using the automounter that must be mentioned.

First of all, the automounter mounts all directory hierarchies under `/tmp_mnt`, and creates a symbolic link to that location. This means to you will often see paths that start with the unaesthetic `/tmp_mnt`. Try the command `pwd(1)` for example.

Second, the locations that the automounter watches are initially empty. Only on reference, entries are created. This means that if you type, say,

```
ls /vol
```

you will not see anything the first time. As another result, file name completion supported by some utilities (C and Korn shell and Emacs for example) does not work in these directories initially. You must explicitly type in the full name that you need.

Last, since the logical location contains nothing but symbolic links to the actual mount point, commands like

```
cd /home/mary/./john
```

are likely to fail. Remedy: always use full path names in this case.

2.4. Application Management

If you need to support software for more than one hardware platform – which is normally the case in a heterogeneous environment – a well designed directory structure is what you need.

This section discusses a model that allows you to do so.

First of all, remember that applications are made available as volumes in the network. The first part of this section describes how to organize such a volume.

The second part of this section describes a way to make the commands available to end-users in such a way that the end-user does not have to be bothered with the underlying hardware and operating system platform. It is completely transparent to him or her. Furthermore, the

second part will discuss a concept that avoids the need to change the shell's search path for every command that you want to make available.

2.4.1. Application Directory Organization

This section describes the preferred directory structure for applications.

In general, applications normally have architecture dependent and shareable, architecture independent data. Furthermore, it is considered good behavior if the application does not require to be able to write in its own installation directory during normal operation.

The directory structure described below results in a complete separation between shareable and architecture dependent data. The directory structure is as follows:

bin.arch	Architecture dependent directory containing executable end-user programs for that architecture. Contains symbolic links to end-user commands stored in ./script . You could also consider using hard links or real copies for that matter. Hard links are cheaper than soft links and are okay to use in this case, since you remain within the same file system. Real copies create unnecessary redundancy and should probably be avoided for the sake of consistency.
script	Directory containing scripts (shell, awk and others) that can be shared across architectures. These scripts are also part of the end-user commands.
etc.arch	Architecture dependent directory containing executable programs for that architecture. These programs are normally not used by end-users directly, but rather by the application itself or by an application administrator.
lib.arch	Architecture dependent libraries and other resource files that could be used by the application itself or perhaps by software developers.

Besides these general directories, you could of course support directories for help texts, fonts, lisp sources, etc. For more information, please refer to the complete paper on "The Art of Automounting".

Having this application directory structure, it can be made available as a volume. GNU Emacs for example, would become available as **/vol/emacs** and it would support the directories **/vol/emacs/bin.sun4** **/vol/emacs/etc.sun4**, **/vol/emacs/lisp** and **/vol/emacs/info**, say.

If the application itself needs to access its own files during operation, it must be built so that it refers to its installation directory, and not to some other place in the file system. For instance, GNU Emacs refers to its own Lisp files during execution, and it should use the path **/vol/emacs/lisp** to access them. Not something like **/usr/local/emacs/lisp**.

2.4.2. Making the Application's Commands Available

In general, applications, tools and utilities exist of one or more end-user commands. These commands can be typed at the shell prompt, and the shell will try to execute that command. The shell searches for commands in the directories specified in the PATH environment variable.

This means that one way to tell the shell that you have added a set of new commands is to add an entry to its search path. For example, if

you have added GNU Emacs as a volume, you could make the commands available by executing the following commands:

```
PATH="${PATH}:/vol/emacs/bin.$arch"
export PATH
```

However, this has major disadvantages.

First of all, you need to do this for all the users and all the shells (Bourne, C and Korn) that want to have the Emacs commands available. This can be quite a hassle and is hard to maintain.

Second, the search path gets larger and larger on every such occasion, finally resulting in a twisty little maze containing dead ends as well.

And last but not least, the search path contains architecture dependent parts. This is no real problem, but it is somewhat unaesthetic and it can be avoided as you will see below.

In order to solve the problems mentioned above, you can create a "convenience" **bin** directory that only contains symbolic links to the actual commands. To resolve the architecture dependency, you can create such a directory for every architecture that you need to support. This directory is also made available as a volume and users only need to incorporate this directory in their search paths.

Suppose you want to support the **emacs** command. Then, what you should have is the following symbolic link:

```
/vol/bin/emacs → /vol/emacs/bin.arch/emacs
```

Here, *arch* should be replaced with **sun3**, **sun4** or **sun386**, whichever is appropriate. The shell's search path must of course include **/vol/bin** in order for the shell to find the command.

Note that both **/vol/bin** and **/vol/emacs** are made available through the automounter.

The specific steps required to create this behavior in this example are described below. But please note that more general procedures are described in "General procedures" in the complete paper on "The Art of Automounting".

For example, suppose you want to add the Emacs commands for Sun-3, Sun-4 and Sun386i architectures to the environment.

First, you have to build GNU Emacs and it should have the directory structure described in the previous section. You make this instance of Emacs available as a volume on a file server:

```
mkdir -p /partition/vol/emacs-18.55
```

Then export this volume by adding the appropriate line to **/etc/exports** and running *exportfs(1)*. Finally, add the following lines to the **auto.vol** map and propagate the changes on the network:

```
emacs          server:/partition/vol/emacs-18.55
emacs-18.55    server:/partition/vol/emacs-18.55
```

Conclude this step with installing the previously built Emacs in this directory on the file server.

Next, what you do is create a volume for the architecture dependent bin directory on a file server if it does not already exist:

```
mkdir -p /partition/vol/arch/bin
```

Note that the architecture dependencies are reversed, or turned inside out, in this occasion. That is, the *arch* directory is located one directory level higher than the **bin** directory. This concentrates all architecture dependent parts under one single exported directory. If you need

to, you can import this single directory later on **/usr/local**. If you also support **lib** and **etc** directories, **/usr/local** becomes what you were used to until now.

As with the Emacs volume, export it and make it available to file clients by modifying and propagating the **auto.vol** map. The trick to resolve architecture dependencies is in this map! You should add the following entry:

```
bin server:/partition/vol/${ARCH}/bin
```

The variable **ARCH** is set by the automount process at start up (so, during boot time). It's value reflects the application architecture. For example, for a Sun-3 it will equal **sun3** and for a Sun-4 it will equal **sun4**.

This means that the directory imported depends on the architecture of the file client.

Finally, create the appropriate symbolic links in the **bin** directories:

```
cd /partition/vol/arch/bin
ln -s /vol/emacs/bin.arch/* .
```

Repeat these command for all the architectures that you support and you're done!

Please note the final dot at the end of the second command. It is essential that you specify it in order to create all the appropriate symbolic links in one step.

What Happens When you Invoke a Command?

Suppose you invoke the command:

```
emacs
```

The following things will happen. The shell searches through its path and will find **/vol/bin/emacs**. Since **/vol/bin** is handled by the automounter, it will be imported from the appropriate file server on first reference. Remember that **/vol/bin** is architecture dependent. That is, it will come from the location *server:/partition/vol/arch/bin*.

Because

```
/vol/bin/emacs
```

is a symbolic link to

```
/vol/emacs/bin.arch/emacs
```

/vol/emacs will also be imported by the automounter and the shell will finally execute the correct architecture dependent command.

The Emacs process itself will reference its private files also through **/vol/emacs**.

2.4.3. Volumes for Other Purposes

Besides using volumes for applications, you can also use them for other things like an architecture dependent **bin** directory (as described in the previous section), a network wide tmp directory, network wide user initialization and prototype files, include files, manual pages, etc.

This section discusses some of these topics.

A Network Wide tmp Directory

It is very convenient to have a network wide tmp directory available. It provides an easy way to transfer files between colleagues without having to use each other's home directories. Name this directory **/vol/tmp** and set the access permission to **rw-rw-rw-**. The physical location of this directory could be anywhere on the network and you never have to back it up. Consider putting it somewhere in a partition also containing swap files for diskless clients.

Make sure you remove stale files once in a while to avoid filling up the file system.

User Initialization and Prototype Files

Consider supporting a directory **/vol/default** with the subdirectories **init** and **proto**. You could put **.login**, **.cshrc**, **.profile**, **.openwin-menu** and a bunch of other files in the **init** directory that users source during their login sequence. You could install prototypes of these files in the **proto** directory that are used for new accounts added to your system. Et cetera.

Network Wide Shell Scripts

Consider supporting **/vol/script** for network wide shell, awk, sed, perl and other scripts.

Local Manual Pages

Consider supporting **/vol/man** with the same structure as **/usr/share/man** (or **/usr/man**). Create symbolic links to the appropriate places from other volumes that you support, e.g.

```
cd /partition/vol/man/man1
ln -s /vol/emacs/man1/* .
```

To let the **man(1)** command find these manual pages, set your manual search path accordingly:

```
MANPATH="/vol/man:/usr/share/man"
export MANPATH
```

Include Files, Icons and Others

Make your favorite local include files and icons available in **/vol/include** and **/vol/icon** respectively. Create support for **/vol/frame** (FrameMaker), **/vol/fmtemplates** (your local FrameMaker document templates), **/vol/wp** (WordPerfect), **/vol/guide** (Devguide), et cetera!

It's up to you. Feel free to exploit **/vol**! Use your imagination.

3. Summary

To summarize, the following table shows you the essence of this document.

Location	Path
logical	<i>/class/instance</i>
physical	<i>/partition/class/instance</i>

“For instances”

For applications you need something extra:

Applications	Path
logical	<i>/vol/applic-version</i> and <i>/vol/applic</i> for the default version
physical	<i>/partition/vol/applic-version</i>

Commands and architecture dependencies are handled as follows:

Command	Path
logical	<i>/vol/bin/command</i>
physical	<i>/partition/vol/arch/bin/command</i>

And the corresponding maps for the automounter look like:

Map name	Contents
auto.master	<i>/class</i> auto.class [<i>options</i>]
	<i>/-</i> auto.direct [<i>options</i>]
	<i>/net</i> -hosts [<i>options</i>]
auto.class	<i>instance</i> [<i>options</i>] <i>server:/partition/class/instance</i>

And to resolve architecture dependencies, versions and variants use:

Map name	Contents	Location
auto.vol	bin	<i>server:/partition/vol/\${ARCH}/bin</i>
	lib	<i>server:/partition/vol/\${ARCH}/lib</i>
	etc	<i>server:/partition/vol/\${ARCH}/etc</i>
	app	<i>server:/partition/vol/application-version</i>
	app-vers	<i>server:/partition/vol/application-version</i>
auto.direct	/usr/local	<i>server:/partition/vol/\${ARCH}</i>

Easy, consistent and efficient.

4. Conclusion

With the concepts laid out in this paper, plug and play is the way to go. Especially with SunOS 4.1.1 Revision B, life becomes easy. The only thing users have to do, is to get the Ethernet address of their new hot box – which is available from the customer information sheet in the plastic bag attached to the system unit carton – and give this to the system administrator. The administrator then uses this information to set up the NIS hosts and ethers maps accordingly. That’s all. The user can then turn on the power switch and play! All the relevant files are automagically available from the first moment the system is switched on.

How’s that for a change?

Monitoring Network Performance in a Heterogeneous Environment

Martin Beer Shaun Hovers

Department of Computer Science,

University of Liverpool, UK

{ mdb | shaun } @compsci.liverpool.ac.uk

Abstract

The widespread adoption of Local Area Networks has meant that network performance measurement, which was previously the sole preserve of large installations with expensive monitoring equipment, must now be performed by many more computer installations without such equipment and expertise, if acceptable levels of performance are to be consistently achieved. This paper will discuss the design and use of a monitoring and management system which by analysing the actual network stations to collect data on the current status of network traffic, uses the real data collected to provide a full description of current network performance. In addition, allows changes in both the physical network configuration and traffic densities can be simulated, by providing suitable additional data.

A suite of software to perform these functions is currently being developed in the Department of Computer Science, at the University of Liverpool. By providing much more accurate information about the present and anticipated future performance of the network, the approach discussed in this paper will considerably improve the quality of management decision making.

1. Introduction

There has been a dramatic increase in both the usage and size of Local Area Networks in recent years. Whereas previously, networks were either confined to large sites with the full technical resources necessary to fully manage them, or limited to a very small number of compatible machines, connected to share some expensive physical resource, such as a printer, or communication device, using only a small proportion of the available transmission capacity, the local area networks currently being installed, are often intended to provide a fully integrated working environment. This makes them central to the operational activities of that organisation. Not only are a number of independent resources shared between members of a working group, but important communication channels are also provided (either by use of electronic mail or some other communication facility, such as telex or fax), the use of which considerably enhances the efficiency of the members of the

organisation concerned. This means that the management of the Network is much more important to the smooth running of the organisation.

The growth of size and complexity of the network has meant that much more care must be taken in the configuration and layout of the network if adequate performance is to be achieved under all circumstances. This leads to the need to provide an adequate set of tools, which will allow the user to monitor the actual and predicted performance of their network, so that appropriate management action can be taken at an early stage. This paper discusses the design and implementation of such a set of tools.

The increased use of networks has not however led to a corresponding increase in skilled network management operatives. This has led to the situation where many installations using LAN's have them maintained by normal system operators who may not be able to spot faults in the network because of their lack of specific knowledge. Configuration and performance measurement must also be undertaken by normal systems personnel, rather than the networking specialists who have traditionally undertaken this role.

It is clear, therefore that appropriate tools are required, to assist both the operators and the system managers to:

- Initially configure the network, so that the specified performance targets can be achieved,
- Monitor the actual performance in service, so that proper management and control decisions can be made to achieve these performance targets, and
- Simulate the effects of proposed changes either in operating parameters, or in configuration, so that investment in both money and effort can be directed in such a way that it stands the best chance of achieving the objectives laid down...

if full and effective use of the network is to be achieved. Otherwise the design, configuration and management of effective networked systems will remain in the realms of guesswork and chance. Fortunately the need for additional information has already been recognised, and appropriate standards are currently being developed. These include SNMP [Ros88a, Ros88b, Cas88a] in the Internet community. Since these standards will shortly be available on a wide range of different manufacturers' equipment, the management of a heterogeneous environment will be considerably eased.

2. Management Requirements in a Heterogeneous Environment

The need to provide effective monitoring and management services across a heterogeneous network requires a coordinated approach, which effectively monitors all network services provided, independently of their origin or destination. This allows the network manager to obtain a clear picture of the actual activity taking place, rather than having to rely on experience and indirect evidence, as is usually the case. A set of three basic tools were therefore defined to provide the information necessary:

- A facility to monitor the network by inserting "probes" at various points. These probes can then be used to develop an effective coordinated model of total network activity.

- A mechanism by which the data collected by the various "probes" is collected, and analysed, so that an effective model of current network activity and performance can be derived.
- A simulator, which can take the data collected by the monitor, and allow the network manager to assess the effects of proposed changes in the network, with respect to the known performance requirements.

In addition, if the information provided is to be of use to a non-network management expert:

- It must be provided in a format that can be understood readily.
- It should give an accurate representation of the network activity at any given instance of time, that is it should give a real-time network view.
- It should ease the making of management decisions by providing some way of testing new policies before actually implementing them.

From these operating requirements it can be seen that what is required is a monitoring and management system which:

- Uses the actual network nodes themselves to collect and collate data on the traffic being routed through the network with various points on the network reporting regularly thus giving a constantly updated picture.
- Presents this data in the form of graphical output such as in the form of a network map, giving traffic densities and clearly reporting any faults.
- Maintains a full database of monitoring sessions, allowing for analysis of network changes and identification of any trends in activity.
- Allows simulation of alteration in both the physical network configuration and of traffic densities so that the consequences of these changes can be assessed before actually making them.

A monitor following these requirements would allow for full analysis of the consequences of such changes on the individual services provided by the network to be assessed at every stage and for management and investment decisions to be made with a proper understanding of the current service requirements, and future trends. In particular it would be possible by analysing the changes in activity, as they occurred, to identify adverse trends in both traffic densities, and the performance of individual services before they became so significant that they would adversely affect total network performance. It would also be possible to investigate alternative configuration strategies without actually physically altering the actual network.

The management problems do not disappear once the network has been installed. The model of the User Service on which installation was based, has to be maintained and indeed extended as:

- Software and hardware upgrades are installed,
- New services are provided, either by mounting additional software, or by adding further hardware, or some combination of the two, or
- Support for additional users is provided by adding workstations and server capacity.

Even when the network environment is apparently constrained as that found in undergraduate laboratories, changes in usage in apparently

distant areas can have significant effects on the user's view of the service provided. For instance, a large class using a complex windowing package, such as a Software Engineering Design toolkit, causes a significant degradation in the overall service provided on staff workstations, even though there is apparently no sharing of resources between them. This situation is of course unsatisfactory, as it implies that difficulties caused by service overload in one part of the network are being propagated widely, even when only a small number of resources are directly effected. It is expected that the development of a proper functionally-oriented model will identify the causes of such behaviour, which can then be dealt with in an appropriate manner.

3. The Design of the Network Management Tools

A variety of network monitors have been available for a number of years. These include:

- Hardware monitors of various types, some including remote probes so that monitoring can take place at various points in the network, away from the central monitoring station,
- Software programs that run on individual workstations, monitoring both incoming and outgoing traffic at that point, and
- Software programs that attempt to analyse throughput, by reading information from specially placed monitors.

The information gathered is often presented in a form that is extremely unclear, to all but the most experienced expert, also rely heavily on the principle that all traffic is equally important, something which is in practice not always the case. For instance, it is often necessary to guarantee the service for one type of traffic, whereas a similar service level is unnecessary for others that use the same links in the network.

Other problems that do occur include:

- Additional connections must be made to place a hardware monitor, changing the configuration of the network, unless the monitor remains permanently attached.
- Hardware monitors only look at a single point in the network at any one time (unless multiple monitors are available) so a complete picture cannot be developed, since moving the monitor can significantly change the performance of the network.
- The output provided by most monitors tends to be specialised, making interpretation a job for a specialised network expert, rather than a normal system operator.
- Where remote probes are used or coordinated readings are taken, the actual act of monitoring the network introduces additional network traffic, which is often significant enough to change the performance of the network, as perceived by the normal user.
- Many monitor programs completely omit certain types of network traffic, which may have a significant effect on overall performance.

The lack of a complete picture also means that it is impossible to build an effective simulation model, so that the monitor can only be used to measure changes in network operation when the physical or organisational change has actually been made. So costly and/or time consuming network changes may be made only to discover either no improve-

ment has taken place, or even worse, that network performance has in fact been reduced.

Hardware monitors are very expensive which means that it is difficult for small network users to justify the purchase of one, in the light of the likely savings that the improved monitoring that the use of such an instrument is likely to achieve. The constraints on most organisations budgets make it much more likely that any additional investment will be channelled into additional workstations, which are likely to cause overload problems on the network, rather than in the "non-productive" investment that the provision of such instrumentation. Only when performance deteriorates to such a level that it seriously effects the organisation's ability to function, does the proper management of the network become a priority issue.

In the Department of Computer Science at the University of Liverpool, work is currently underway to create a suite of software to perform the required monitoring functions [Hov91a]. Considerable attention is being given to making the system usable by system managers so it is not necessary to rely on networking specialists, who would not be available on many real sites. This software is designed to fulfill the requirements by:

- Incorporating the monitor probe into particular network nodes as a daemon process. These daemons are placed at strategic points around the network, so that the traffic types and densities can be monitored in a coherent manner.
- Recording the traffic passing into and through that particular node of the network, by means of that process. Rather than continuously passing information, as it is collected, the monitor probe collects data over a short, predetermined period, and then passes the results back to the collecting point at an appropriate later time.

One node (usually a workstation) is designated the collecting node and is used to collect the information supplied by all the individual daemon processes. The information is then collated and graphically displayed on a single display using an X-windows [Mor86a] system. The display would then give an accurate display of the current network situation including traffic densities and any faults or breaks which occur in the network.

Included in the information provided by the daemons are the source, destination, and type of each packet, together with the type of service required. This information, as well as being used to provide the active network view, is also kept to allow an accurate network model to be created. With the information the model can provide a method of testing out new configurations of the existing network and also of new additions to network hardware. In fact this information allows:

- Trends in network usage to be identified and assessed before they adversely affect network performance.
- The effect of changes in operating and other software to be properly analysed, by upgrading a limited number of machines at first, such that performance problems can be identified and remedied before finally committing the whole network.
- The effects of configuration changes to be properly assessed before any final decisions are made.

These advantages allow the network manager to properly assess the effects on both functionality and performance of changes in the work-

ing environment before before they become apparent to the normal user.

4. Conclusions

The tools discussed in this paper are designed to assist the network manager in providing a coherent and efficient working environment for all users of the target network. To this end the following design considerations have been used:

- Clear and simple monitoring facilities allow systems managers to make accurate operational decisions, without unnecessarily effecting other parts of the service.
- It is better to discover potential problems in capacity and performance before resources are finally committed, rather than after changes have been made. This is because it is often extremely difficult to downgrade a system when problems are discovered some time after the upgrade.
- The network is a system resource, like any other, and must be managed effectively if the maximum organisational benefits are to be achieved.

The set of programs described in this paper is an attempt to provide the necessary tools for the network manager to deal with these problems before they become plainly apparent because of their devastating effects on the normal user service.

References

- [Cas88a] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A Simple Network Management Protocol," *RFC 1067*, TWG (August 1988).
- [Hov91a] S. Hovers and M. Beer, "Monitoring Network Performance in a Heterogeneous Environment," *Networks: Design, Planning and Standards, Proceedings of the Networks'91 Conference, Blenheim-Online*, London, UK, pp. 373-379 (1991).
- [Mor86a] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Donelson Smith, "Andrew: A Distributed Personal Computing Environment," *CACM* **26**(3), pp. 184-201 (1986).
- [Ros88a] M. Rose and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-based Internets," *RFC 1065*, TWG (August 1988).
- [Ros88b] M. Rose and K. McCloghrie, "Management Information Base for Network Management of TCP/IP-based Internets," *RFC 1066*, TWG (August 1988).

StormCast – A Distributed Application

Dag Johansen Gunnar Hartvigsen

University of Tromsø, Norway

{ dag | gunnar }@cs.uit.no

Abstract

The objective of this paper is to present the architecture of the distributed application StormCast. StormCast has been designed and implemented to enable an evaluation of the proposed architecture as well as the underlying operating system mechanisms. StormCast consists of a set of modules monitoring weather data and a set of sub-applications using this data. StormCast is running in both local area and wide area networking environments.

1. Introduction

The last ten years have seen an increased interest in and evolution of distributed systems with a distributed system defined as a distributed operating system consisting of a replicated set of microkernels and a set of operating system services typically implemented as a set of servers outside the microkernels. This includes distributed systems as Amoeba [Mul86a], Chorus [Roz88a], Mach [Acc86a], Sprite [Ous88a], V [Che88a] and the x-kernel [Pet90a]. However, we do not face similar innovative changes in user applications resulted from enhancements to the operating system technology. Consequently, applications are still often constructed based on a monolithic approach not taking advantage of the distributed nature of the underlying distributed system. A common approach is basically to run traditional sequential applications on distributed systems to achieve objectives as increased performance at a lower price, utilization of present resources or increased fault-tolerance.

A distributed application is defined as a set of separate modules or processes cooperating to meet an overall application goal. We argue for an increased interest in distributed applications based on the assumption that several sectors exist lending themselves naturally to distributed computing. The reason for the lack of distributed applications is not of a technical art as long as current distributed and networking systems seem to provide a means of running distributed applications. The problem is more that application engineers still consider applications in the term of monolithic structures.

This paper describes the architecture of the StormCast distributed application, an application in the weather and environment sector. Our methodology is to design and implement a realistic distributed applica-

tion in full scale and run and validate it on different network based platforms.

The rest of this paper is organized as follows: Chapter 2 argues for the application sector chosen for distributed processing. Chapter 3 describes the architecture of StormCast. Chapter 4 discuss aspects of the architecture while chapter 5 concludes the paper.

2. The Weather Domain

Several distributed applications have been implemented, Grapevine [Bir82a] to mention one. Typically, a distributed system itself also contains a set of distributed applications as a distributed name server, a distributed file server or a distributed authentication server. This can also include monitoring mechanisms to capture the behaviour of distributed systems as [Hol90a].

The sector we have chosen for distributed processing purposes is outside the typical operating system sector, and it is based on the recognition that real-life monitoring is distributed in its nature. This includes monitoring for industrial purposes as a factory automation application monitoring the different stages in a production line or a defence application monitoring specific events occurring in a geographical area of interest.

We have investigated the weather and environment sector. As argued in [Joh88a], the weather sector is a proper candidate lending itself naturally to distributed computing. The reason for the choice of this sector includes the following:

1. Monitoring of weather data is distributed in its nature. Weather data is either monitored from different points on the ground or at upper air installations as weather balloons or satellites. Typically, this involves sensors measuring data as temperature, wind speed and wind directions, humidity, cloudiness, precipitation, brightness and visibility. We intend to investigate if both monitoring and transmission of weather data can be fully automated. Due to practical limitations, Version 1.8 of StormCast is based on ground sensors exclusively assuming that this type of equipment can be used to obtain sufficient weather data for a practical distributed application.
2. Usage of weather data is already heavily based on computers where monitored weather data are input to complex numerical computations. This involves heavy computations requiring hours of mainframe cpu time. We intend to investigate if numerical weather models can take advantage of the parallel processing potential in distributed systems. We also intend to investigate if alternative computational models can add to the process of predicting weather, either exclusively as separate computational models or together with existing numerical models.
3. The geographical area to monitor span areas so large that wide area networking is commonplace. Consequently, the distributed application must operate in local area networking environments as well as in wide area networking environments including mobile networks and satellites for communication. Research on distributed systems and distributed applications in this type of environment is rare. We intend to add to the knowledge in this particular field.

4. We assume that monitoring of data related to weather data can be done by small enhancements to a distributed application designed and implemented for monitoring of weather data. This will include monitoring of environmental changes as pollution by reuse of existing hardware and software built for the weather data monitoring.
5. Last, but not least is that there is a local interest in the weather domain facing the fact that StormCast is intended to operate in the Arctic regions of the world. This is a geographical location where human activity might depend heavily on the current and future weather conditions.

To summarize, we assume that the weather sector is a proper candidate for the construction of a realistic distributed application. The StormCast distributed application has been designed and implemented, and its architecture will be presented in the succeeding chapter.

3. The StormCast Architecture

The StormCast architecture basically consists of two functional layers, the data collection layer and the weather application layer. In short, the data collection layer obtains the weather data needed by the weather application layer.

3.1. The Hardware and Software Platform

Several StormCast versions have been developed based on different hardware and software platforms. Version 1.8 of StormCast is based on an initial design and implementation [Joh88a] running on the distributed system Amoeba [Mul86a] in a wide area networking environment [Ren88a]. A change in development platform has mainly been motivated by pragmatic concerns. As long as earlier Amoeba versions were not used as our production system and as long as few sites were running Amoeba, we decided to use a more common platform. Our current UNIX platform has shown to be more convenient when building a large distributed application as StormCast since this has enabled a speedier application development. Nevertheless, we have lost functionality that a distributed system normally provides.

StormCast is mainly running on 50 MHz Motorola 68030 workstations (Hewlett-Packard 9000/4xy s/t) connected through a 10 Mbit/s Ethernet. StormCast consists of a set of processes written in the programming language C augmented with library calls for inter process communication. Each module in StormCast is running as a UNIX[†] process using TCP/IP (Transmission Control Protocol/Internet Protocol) and the X Window System. X.25 and cisco Systems AGS Gateway servers are used for communication over wide area networks. Modems are also used for transmission over existing telephone lines.

3.2. The Data Collection Layer

The data collection layer is responsible for monitoring and transmission of weather data. The geographical area monitored are separated in different areas or domains, where each domain contains one synthesizing module and a set of monitoring modules. Figure 1 illustrates monitoring of a geographical domain containing a synthesizing module and

[†] HP-UX Release 7.03.

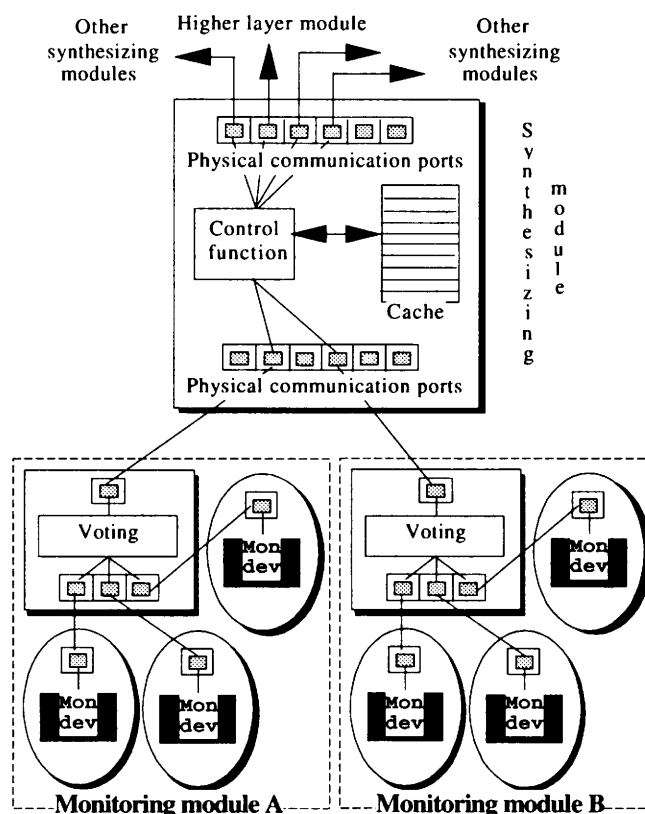


Figure 1: The data collection layer

two logical monitoring modules [Har90a]. The interaction schema is mainly based on a pure client-server model.

The synthesizing module is responsible for the collection of data in its domain by replying on requests from the weather application layer or from other synthesizing modules. A synthesizing module issues requests to other synthesizing modules on behalf of a module in the weather application layer. A reply is either based on timestamped weather data cached locally in each synthesizing module or based on nested requests to the monitoring modules in each domain. The default is to return current data involving the monitoring modules. Each logical monitoring module contains one or several physical modules and a voting function. The voting function manages the different replicas of the physical monitoring modules with replication completely hidden for modules outside the logical monitoring module. The voting function multicasts the same request to all physical modules in the same logical monitoring module, get replies from each of them, validates the received data by a voting function and returns the majority vote to the synthesizing module.

The amount of data transmitted between a monitoring module and a synthesizing module is about 40 byte, and the data is timestamped with the local time in the voting function. The amount of data returned from a synthesizing module is in the order of $40 \text{ byte} * n$ where $n \leq$ the amount of logical monitoring modules in the domain(s) of interest.

Figure 2 illustrates a typical sequence of data transmitted. For mobile monitoring modules, we also add location data. Such modules will typically be located on board on trawlers in the Barents Sea.

```
typedef struct {
    SOURCE Source;
    time_t ObservationTime;
    int BarometricPressure; /* 955 - 1075 */
    int Temperature; /* -60 - +40 degrees C */
    int WindSpeed; /* 0 - 100 m/s */
    int WindDirection; /* 0 - 360 degrees */
    int DewPoint /* 0 - 100 % */
    int Clouds; /* 0 - 8 n/8 */
} WEATHERINFORMATION;
```

Figure 2: Weather data transmitted

The data collection layer is used by several applications found in the weather application layer. Figure 3 illustrates three geographical domains being monitored and two workstations running weather applications. The next subsection describes the functionality of the present weather application layer.

3.3. The Weather Application Layer

The weather application layer contains a set of applications providing functions as weather maps, weather statistics, severe storm predictions and pollution monitoring and warning. Each application is based on data obtained from the same data collection layer.

3.3.1. Weather Map

A weather map was the first application built in the weather application layer [Joh88a]. The weather map shows the current weather situation

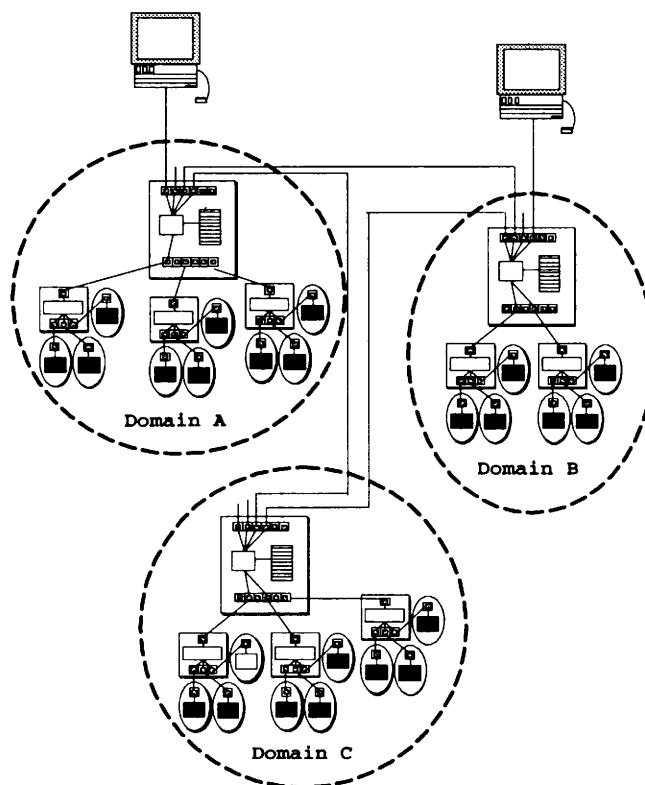


Figure 3: A small configuration of StormCast

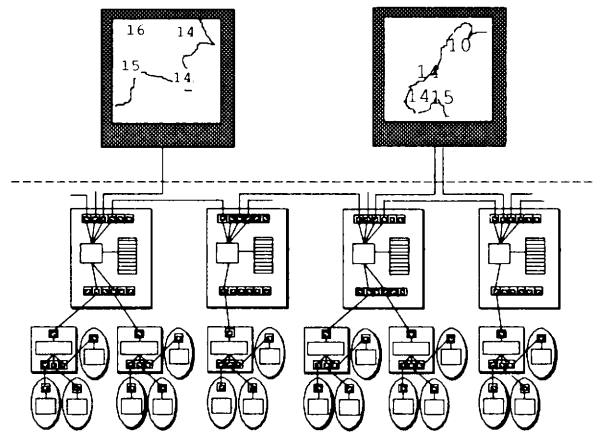


Figure 4: Weather maps displaying the current weather in two domains

in one or several domains based on data obtained directly or indirectly from a nearby synthesizing module. Each weather map module interacts with the same synthesizing module each time. This synthesizing module returns weather data locally cached or obtained from the logical monitoring modules in its domain. Alternatively, other synthesizing modules are requested similarly through this synthesizing module to obtain data from other domains. Figure 4 illustrates two weather maps displaying data from two domains. The weather map on the left covers a part of Norway based on data from the leftmost synthesizing module. The weather map on the right displays the whole Norway by data obtained from the entire set of synthesizing modules.

Based on X Windows, a weather map module accounts for almost a Megabyte of code. Typically, a request for 1 Kbyte cached data in a synthesizing module located on another node in a local area network takes about 30 milliseconds. A request for weather data located in another part of Norway involving X.25 traffic is in the range of one second. Similar transatlantic requests between two modules take 2.5 seconds on the average to carry out. Communication accounts for most of these figures, especially in a wide area network environment.

3.3.2. Weather Statistics

A second application in StormCast is a statistics application providing weather statistics for a user. Such a user also includes weather applications predicting weather forecasts. In functionality, the statistics application issues requests to the data collection layer on a regular basis storing the data. On demand, different weather statistics can be provided based on the stored data. Typically, this is average figures for a specific interval as average temperatures the last month.

We have used two different approaches, the first uses indexed UNIX files as a storage medium while the second approach uses a SQL based database. This is illustrated in Figure 5. Clearly, we prefer the first approach due to performance benefits and little loss in functionality.

3.3.3. Weather Predictions by Computers

A third application developed using the same data collection layer is motivated by the fact that weather forecasters already base much of their work on output from computerized weather models. This is often

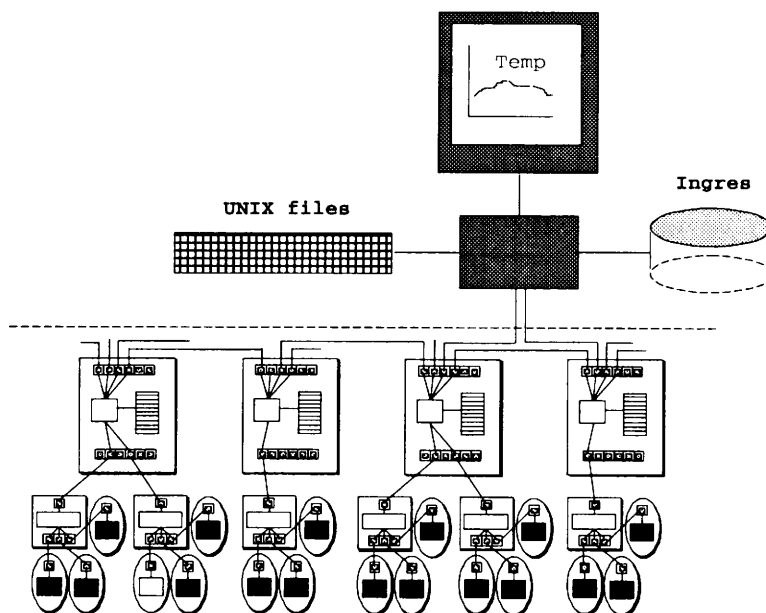


Figure 5: The statistics application

complex numerical models running as monolithic processes on mainframes.

We have taken two different approaches when building a weather prediction application as well. The first approach is to use a traditional numerical model and investigate if this can take advantage of a distributed system compared to the centralized approach running the application on a mainframe. This is illustrated in Figure 6, where a set of nodes is running the computational model in parallel.

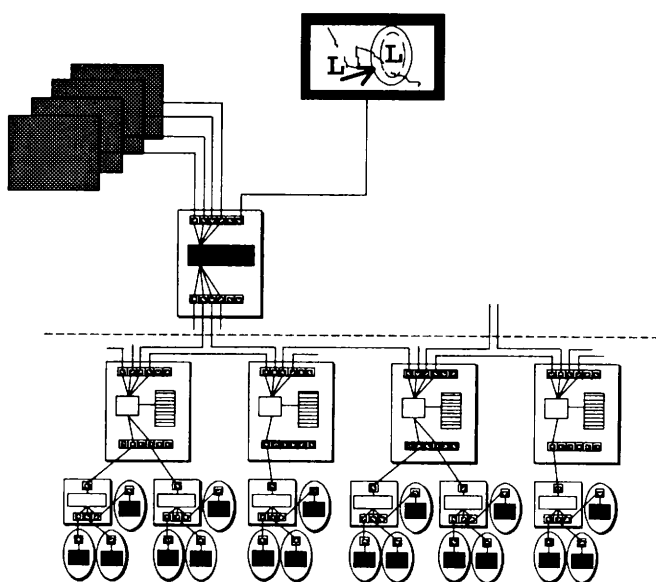


Figure 6: Running numerical computations in parallel

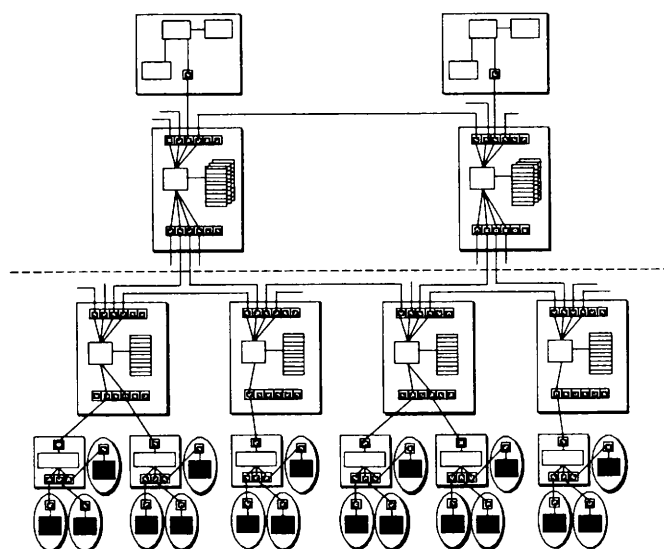


Figure 7: A set of expert systems predicting severe storms

A classical numerical model divides the atmosphere into six layers which is analysed and projected separately. The final output combines the output from the six layers. This might illustrate the potential for parallel processing in this particular application sector.

A second approach we have taken is to investigate if alternative computational models can be used in the process of predicting severe storms. Consequently, we have built an application consisting of a set of expert systems predicting severe storm forecasts [Har88a, Har90a]. To simplify the actual weather forecasting problem, we restrict the weather forecasting process to severe storm forecasting. We also see the practical expert system approach as a supplement to traditional numerical model based forecasts. This means that StormCast is intended to come with microforecasts, that is highly detailed forecasts of conditions over relatively limited areas. This is motivated by the fact that very local storms might appear in the Arctic within very short time intervals. Figure 7 illustrates the design of this application with a set of expert systems predicting microforecasts for the domain they receive weather data from. Implicit in this structure is communication of information, which is handled by a set of blackboards also located in the weather application layer. Typically, this communication requires wide area networks. We intend to use this StormCast application on a set of trawlers communicating through the Inmarsat.c satellite running X.25. Each trawler constitutes a weather domain with an expert system module, a blackboard module, a synthesizing module, and at least one logical monitoring module.

In functionality, the expert system in each domain predicts storm forecasts regularly based on locally monitored data. However, storm forecasts are multicasted to neighbouring domains aiding in the prediction process in the other domains. To illustrate the usage of this multicasted information, consider two domains North and South. The South domain might have a storm prediction indicating that the probability of a storm in this region is 0.2. However, if an upcoming storm centre is located in domain North heading south, the local storm prediction in domain North might be 0.9. By multicasting this prediction to its neigh-

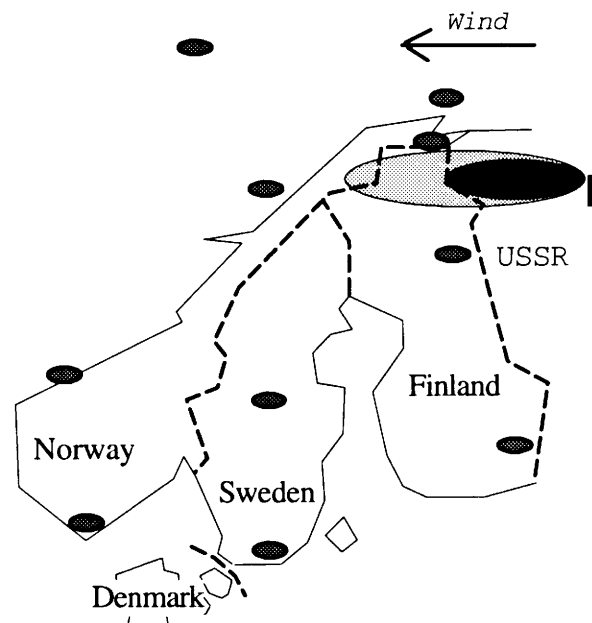


Figure 8: *The propagation effect of a serious accident*

bours including domain South, domain South can use this information to give a more accurate microforecast for itself.

3.3.4. Pollution Monitoring

A full implementation of StormCast might have a configuration where the data collection layer covers a wide geographical area as the Nordic Countries. This involves a large set of black boxes monitoring weather data replaying on requests from synthesizing modules, which again serve requests from the weather application layer including for instance the weather map and the expert systems predicting severe storms. Our objective is to add minimal software and hardware to this configuration achieving much more in functionality.

This extra functionality added is an application monitoring environmental changes. This is basically monitoring of pollution changes by adding some extra sensors to the black boxes already monitoring the weather. The software changes are minor; hardware changes is basically to add some extra sensors. Reuse of the rest of the software and hardware is then achieved.

Another interesting aspect by integrating weather and pollution monitoring is the possibility to predict how the consequences of an unfortunate accident are gradually felt further and further afield. Consider a scenario with an accident in a nuclear plant in a neighbouring country east of Norway. Figure 8 illustrates how pollution from such an accident might disperse based on the wind direction monitored.

Today, it might take less than an hour from an accident occurs in the nearest located power plant until the first Norwegian population is reached. However, the current warning system operates with a warning period of 24 hours. In our part of Norway, this is basically based on data from one monitoring module. We intend to show that this can be dramatically improved by use of StormCast.

Figure 9 illustrates the architecture of this application. The boxes between the interface modules and the data collection layer are warning modules multicasting pollution warnings.

4. Discussion

Few distributed applications exist in the weather domain, the existing ones either monitor data automatically or predict forecasts by expert systems or numerical computations. To our knowledge, no weather applications fully automate the tasks as described in the previous section. StormCast is also based on distributed problem solving by a set of expert systems, each expert system responsible for a small domain. Alternatives found in the weather domain are all based on monolithic structures.

The layering in StormCast has enabled software reuse since the bottom layer monitors and transmits raw data while the upper layer handles the more application specific aspects. The approach we have taken in StormCast is important to meet the software crisis generated by the problem of developing the same software over and over again. We have shown that different applications can be built by reuse of the same mechanisms, in StormCast this mechanism is a separate layer providing raw weather data. Maybe similar mechanism layers can be found in other application domains as well.

StormCast consists of a set of modules who communicate with each other through a well defined interface. This has the advantage that modifying one component does not require changing or recompilation of another. Reuse of software is also supported since it is easier to reuse specialized modules in stead of a large, monolithic application. Our separation of cpu intensive modules from I/O modules also eases process migration, a functionality we find important to guarantee availability of long lived processes in StormCast. Replication is also supported by the modularization we suggest. We replicate the modules acting as bottlenecks, which provides a service who should be made more fault-tolerant or which must meet functional requirements. An

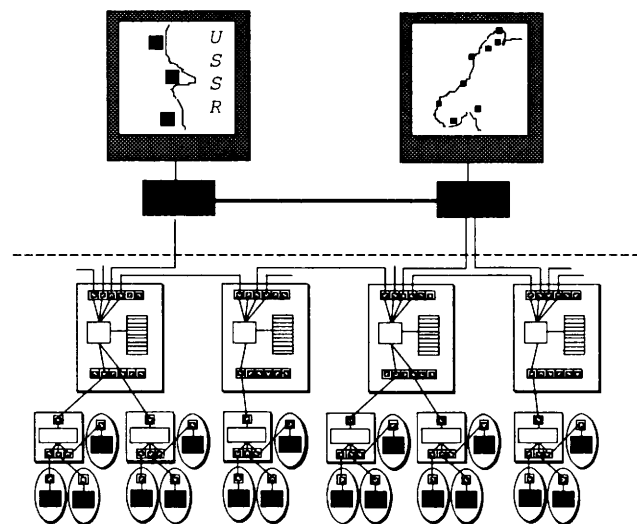


Figure 9: Pollution monitoring and warning in StormCast

example of this is adding a new user to StormCast where a customized interface module is all code that is added. Our experience is also that scaling is supported by a decomposition as shown in StormCast. We simply scale the part of the application which is necessary, for instance the interface modules when new users are added or a synthesizing module is added in a domain if this is heavily loaded by requests.

StormCast has been configured differently involving over 100 nodes both in local area and wide area networking environments. One extreme was to centralize as many modules as possible on one node, the other extreme was to allow only one module of Stormcast on each node. The first alternative had its drawbacks related to hardware fault-tolerance aspects, the distributed nature of the application as well as scaling problems.

The other extreme was more attractive since it meets fault-tolerance requirements. A distribution of modules also meets the functional requirements of StormCast where at least the monitoring modules must be distributed. The configuration we favour is to distribute the physical monitoring modules to the geographical area to monitor. The voting function and the synthesizing module are located on the same local area network as the interface modules. Especially in a wide area networking environment, we found it appropriate to have the synthesizing modules as close to the user as possible to ensure availability of cached data in case of network partitioning.

In StormCast, we are interested in a failure model where software failures are better dealt with. This is motivated by the recognition that software failures are difficult to discover during the construction process of an application. Even though formal methods and thorough testing are applied, several software failures are first discovered when the application is in operation. Application users in for instance the financial, medical or defence sectors might consider run time failures as something that should be avoided at almost any cost, even if the cost of redundancy techniques is generally higher since it entails extra processing and network traffic. Hence, the methodology we use in StormCast is labour intensive, but reduces the probability of run time errors.

This means that the replication in StormCast intends to cope with a failure model where logical faults might occur. This is done by involving n different programmers as well, each one responsible for one of the n replicas. Each replica implemented has a well-defined interface and some functional requirements to met. The internal design and implementation depends entirely on the different programmers. The n different replicas are then run in parallel as we do with the monitoring modules in StormCast. We consider this a convenient methodology to reduce traditional software errors, even if it is labour intensive.

To improve availability, replication of monitoring modules ensure that data is delivered from a domain even if one replica breaks down. Caching of data in the synthesizing modules also ensures availability. If a network partition occurs or all monitoring modules fails, cached timestamped data might still be useful for the client requesting the data.

The n -replication schema in StormCast includes configurations with $n = 1$. The idea is that redundancy is specified when configuring the application. Then, trade-offs between redundancy and added functionality must be done. Not all domains might need the same reliable data, it is for instance domains close to a potential pollution area who should be made more fault-tolerant. It is also convenient that a module issuing a pollution warning base this on very reliable data. No redun-



dancy in the pollution monitoring process means that somebody can hamper with the single monitoring module.

To summarize, we consider fault-tolerance as one of the most important objectives of StormCast. For a user of a distributed system, failure transparency [ANS87a] should be guaranteed as a rule. The exception here might be as exemplified in StormCast where timestamped data cached in the synthesizing modules are returned when the monitoring modules are unavailable. Then, the user must be notified that current weather data can not be obtained. However, sometimes it is better with old data than no data at all.

If software failures shall be masked as we do in StormCast, failure transparency can not be guaranteed at the implementation level. The programmer of the different replicas do not have to know about the other $n - 1$ programmers, but the implementor of a voting function as in StormCast must deal with this replication. Hence, we consider fault-tolerance as a task where both the distributed system and the application programmer must contribute. The distributed system might then be responsible for the distribution of modules which is replicated by the different programmers.

5. Conclusion

StormCast has been designed and implemented to obtain experience with distributed computing. In the light of this objective, StormCast meets its goal. StormCast has been running on Amoeba, it is running on several UNIX platforms and currently on Mach as well. We also intend to use it on distributed systems being developed in the department.

The objective of this paper has been to show the potential for distributed computing in one application domain. StormCast is now implemented in larger scale by Norwegian industry. This means that StormCast is developed to run in full scale both on- and offshore in the Arctic part of Norway.

In addition, tailorware based on StormCast is being developed. This involves StormCast versions monitoring weather conditions at a small domain as an airport to predict potential dramatic weather situations. Located near to the North Pole, small airports in the northern part of Norway might have turbulent weather conditions during the winter. Another tailorware is weather monitoring for cross country skiers. This involves a set of sensors located around the track providing weather data for the skiers in the start area.

Many distributed systems exist today, some of them intended to be the new virtual machine for user applications. A common problem is as always to convince end users of potential benefits of this new concept. Benefits like fault-tolerant computing, improved resource sharing or a better price/performance ratio are arguments used for distributed computing. However, an argument often omitted is the potential of running distributed applications on top of this virtual machine.

Consequently, distributed applications are more the exception rather than the rule in many application sectors. We have intended to contribute to the fundamental question of what to use distributed systems for. Our approach has been to show that the weather sector lends itself to distributed computing. As a result, StormCast has been designed, implemented and evaluated. StormCast might already foster development of industrial distributed applications which means that one of our

objectives has been met. We have shown that there are at least one application sector born for distributed computing.

The next fundamental question is whether a distributed system is necessary for distributed computing. We have run StormCast on both the distributed system Amoeba and on alternative virtual machine platforms including standards such as UNIX, TCP/IP and X Windows. Clearly, we experience the benefits of the first approach. This includes the ability to access local and remote operations transparently without explicit location knowledge, this includes the ability to improve fault-tolerance, this includes the ability to migrate processes when dynamically configuring the distributed system and this includes the ability to monitor and control the dynamic behaviour of distributed applications.

There is at least two approaches when building a virtual machine for distributed computing. One can either build a distributed system from scratch or one can build ad hoc mechanisms on top of existing virtual machines as UNIX filling the functional gap between them. For instance, we have used ISIS [Bir85a] as a means of filling this gap. We argue for the first approach as long as common virtual machines as UNIX were not at all designed for distributed computing. The distributed system might end up with the same functional interface as current virtual machines due to pragmatic decisions. However, we also end up with a virtual machine properly designed for the distributed environment to operate in.

6. Acknowledgements

The StormCast project has involved more than 50 students and staff from several departments at the university. Listing all contributors is not feasible. However, Arne Helme and Roar Steen have made invaluable contributions to the project. Discussion with Otto J. Anshus, Tore Larsen, Robbert van Renesse, Kenneth P. Birman and Michael D. Schroeder have been valuable.

References

- [Acc86a] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for Unix development," pp. 93-113 in *Proceedings of the Summer Usenix Conference*, Atlanta, GA (July 1986).
- [ANS87a] ANSA, *ANSA Reference Manual, Release 00.03.*, 1987.
- [Bir85a] K. P. Birman, "Replication and Fault-Tolerance in the ISIS System," *Operating Systems Review* **19**(5), pp. 79-86 (December 1985).
- [Bir82a] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder, "Grapevine: an exercise in distributed computing," *Comm. of the ACM* **25**(4), pp. 260-273 (April 1982).
- [Che88a] D. R. Cheriton, "The V distributed system," *Comm. of the ACM* **31**(3), pp. 314-333 (March 1988).
- [Har88a] G. Hartvigsen and D. Johansen, "StormCast – A distributed artificial intelligence application for severe storm forecasting," pp. 99-102 in *M. G. Rodd and T. L. d'Epinay (Eds.), Distributed Computer Control Systems 1988*,



Proceedings of the Eight IFAC Workshop, Pergamon Press, Oxford, England, 1989, Vitznau, Switzerland (13-15 September 1988).

- [Har90a] G. Hartvigsen and D. Johansen, "Cooperation in a Distributed Artificial Intelligence Environment the StormCast Application," *Engineering Applications of Artificial Intelligence* 3(3), pp. 229-237 (September 1990).
- [Hol90a] D. B. Holden, O. J. Anshus, T. Fallmyr, D. Johansen, P. Noordzij, H van Staveren, and J. Hall, "A Study on Control in the Distributed Systems Environment," in *IFIP TC6 WG6.4a Int. Symp. on Local Communications Systems Management (invited paper)*, North-Holland (To be published), Kent, UK (18-19 september, 1990).
- [Joh88a] D. Johansen, "Weather forecasting – distributed in nature," pp. 197-203 in *Network Information Processing Systems, Proceedings of the IFIP TC6/TC8 Open Symposium*, North-Holland, Amsterdam, Sofia, Bulgaria (9-13 May 1988).
- [Mul86a] S. Mullender and A. S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," *Computer Journal* 29, pp. 289-300 (March 1986).
- [Ous88a] J. K. Ousterhout, A. R. Cherenson, F. Douglass, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *IEEE Computer* 21, pp. 23-36 (February 1988).
- [Pet90a] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao, "The x-kernel: A Platform for Accessing Internet Resources," *IEEE Computer* 23(5), pp. 23-33 (May 1990).
- [Ren88a] R. van Renesse, H. van Staveren, J. Hall, M. Turnbull, B. Jansen, J. Jansen, S. Mullender, D. Holden, A. Bastable, T. Fallmyr, D. Johansen, S. Mullender, and W. Zimmer, "MANDIS/Amoeba: A widely dispersed object-oriented operating system," pp. 823-831 in R. Speth (Ed.), *Research into Networks and Distributed Applications*, Elsevier, Amsterdam (1988).
- [Roz88a] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "CHORUS distributed operating system," *Computer Systems* 1, pp. 299-328 (Fall 1988).

Location-Independent Object Invocation in Open Distributed Systems

Herman Moons Pierre Verbaeten

*Dept. of Computer Science,
Katholieke Universiteit Leuven, Belgium.
herman@cs.kuleuven.ac.be*

Abstract

Open distributed computing in an internetwork environment has gained considerable attention in the past few years. This paper presents COMET, a *Common Object Management Environment*, that serves as a testbed for investigating basic problems associated with open distributed computing in an internet environment with mobile objects.

Object naming and location schemes are of vital importance in an open distributed system, since they provide the basis for all interactions between objects in distributed applications. In this paper we present the COMET naming scheme, which is tailored to an open environment. This naming scheme is complemented by a location scheme, that ensures efficient mapping of location-independent names on object addresses.

Special care is taken to provide open-ended solutions. This permits a seamless integration of application specific naming and location strategies within a general framework.

1. Introduction

In the past few years world-wide connectivity of computing equipment has become a reality. Local computing facilities are being integrated with one another, providing users with access to computing services scattered throughout the world. Attention is now shifting to the issues of Open Distributed Computing (ODC). The goal of ODC is to support transparent distributed computing in an internet environment. Building such an *Open Distributed System* is a difficult task, due to the very heterogeneous nature of the underlying architecture.

The object-oriented approach holds much promise as a base for an Open Distributed System. Objects provide good data abstraction facilities. Furthermore, when objects are mobile, significant advantages can be realised [Bla90a]:

- Ability to share load between nodes
- Reduced communication cost
- Increased availability

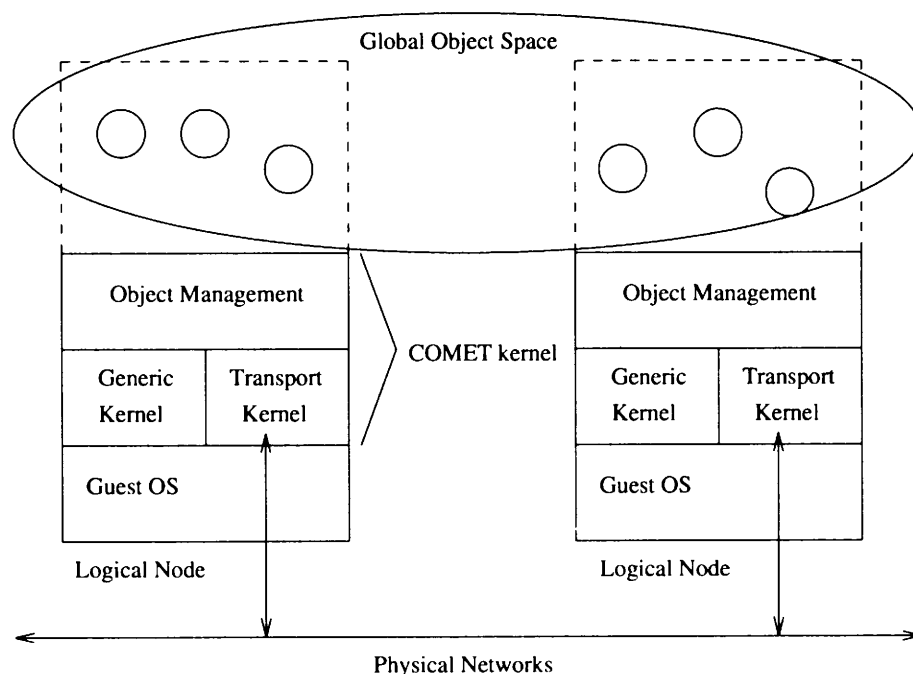


Figure 1: *COMET System Architecture*

- Reconfigurability
- Taking advantage of special hardware

This paper presents COMET, a *Common Object Management Environment*. COMET serves as a testbed for investigating basic problems associated with open distributed computing in an internet environment with mobile objects.

The paper starts with an overview of the COMET architecture. Section 3 presents the naming scheme used to identify objects in an open distributed environment. The following section shows how named objects can be located. Naming and location schemes are then brought together to realise location-independent object invocation. A final section discusses name space management issues.

2. COMET System Architecture

The target environment we envision consists of hosts with widely differing hardware architectures, and networks based on different communication technologies. The hosts typically run different operating systems, and communication over networks uses a variety of communication protocols.

COMET imposes a logical node structure on this environment. Logical nodes are abstractions of physical machines. A COMET kernel on each logical node provides the necessary support for object management and interaction.

The COMET kernel consists of three major components:

- **Generic Kernel Layer**

This layer provides a uniform set of operating system services [Moo90a]. Its main purpose is to hide the idiosyncrasies of the underlying guest operating system. The Generic Kernel implements activities (light-weight processes), efficient inter-activity

communication, timer support and synchronisation primitives. Where possible, the mechanisms offered by the guest operating system are used.

- **Generic Transport Layer**

This layer attaches to the underlying networking facilities. It offers a uniform set of communication primitives for interaction between application components on different logical nodes in the network. The main function of this layer is to provide the illusion of a fully connected network of logical nodes. The Transport Kernel uses whatever networking facilities are available to implement its communication service (TCP/IP, SNA, OSI).

- **Object Management Layer**

This layer implements the concept of a COMET object. It offers primitives for object creation/destruction, object migration and location-independent object invocation.

The COMET kernels on different logical nodes cooperate to implement the notion of a global object space. Within this object space, COMET objects can interact with each other regardless of their physical location. The following sections investigate how objects interact. More particularly we examine how objects are identified, located and invoked.

3. A Naming Scheme for Open Distributed Systems

In order to invoke an object, we need a way to identify it. When objects would remain at the node where they are created, the node's transport address would be sufficient for purposes of naming and locating the object. When objects are mobile, i.e. they can move from node to node during their existence, we need a way to uniquely identify them independent of their physical location. This is realised by giving names to objects, in such a way that the name unambiguously identifies the object in question. A *naming scheme* describes the set of rules used to construct an object name.

3.1. Classification of Naming Schemes

Naming schemes can be classified according to location content, structure and range.

- **Location Content**

Some naming schemes embed location specific information in object names. The intent is to use this information to speed up the process of locating the object. We therefore distinguish between *location-sensitive* and *location-insensitive* names.

Locus [Pop85a] and Cronus [Gur86a, Sch86a] are examples of systems that use location-sensitive names. Most other systems use location-insensitive names [Sin89a, Mul85a, Das88a].

- **Structure**

Some naming schemes define a hierarchy, where an object name designates a path through the name tree. With such a scheme, names have a *hierarchical structure*. Other systems assign unique names with *flat structure*.

Galaxy [Sin89a] and the Domain Naming System [Moc81a] are examples of systems using hierarchical names. Flat names are

used in Amoeba [Mul85a, Mul86a], Clouds [Das88a] and Cronus [Sch86a].

- **Range**

Object names can be globally unique throughout the distributed system (*global*), or they can be specified according to some reference point (*relative*). In the first case identical names always refer to the same object, whereas in the second case the same name can refer to different objects, depending on the reference point.

Clouds, Cronus and Amoeba use global names to identify objects. An example of relative names can be found in the UNIX system, where filenames can be specified relative to the current directory of the process using them.

3.2. COMET Naming Scheme

Object naming in an open distributed system poses a number of problems, that are a direct consequence of the characteristics of an open environment. First of all, open environments are of a dynamic nature. As new users attach to the system, they want to combine their own namespace with the system's existing namespace. Furthermore, users want to assign meaningful names to objects, and need a way to express the relations between objects.

As a result not all naming schemes are suitable in an open environment:

- *Location sensitive names* are of little use in an environment with mobile objects. When an object moves, it has to leave an indication of its new location at the node referred to by the physical component of its name. When this node becomes unreachable (e.g. because of a node crash), the object can no longer be located using the location-sensitive part of its name. We will therefore focus on location-insensitive names, which are more suitable in an environment that supports object mobility.
- *Global names* introduce a number of complications. First of all, one must devise a mechanism that ensures the system-wide uniqueness of names. A more severe problem occurs when we want to merge name spaces. In that case we must guarantee that all names used in the merged system are unique. This is a complex task, that often requires name translations when crossing the boundary between the old name spaces.
- *Flat names* are very low-level, and require a higher-level naming scheme to make them more meaningful to applications. It seems therefore appropriate to immediately support names that are of direct use to the application.

The COMET system uses a location-insensitive, relative and hierarchical naming scheme. In this scheme, each object has a unique primitive name within a specific naming context. Naming contexts themselves are assigned primitive names within their enclosing context. An object name specifies both the primitive name of the object, and a hierarchy of naming contexts.

When an object is created, it is assigned a unique primitive name in some naming context. The context where the object is created will be referred to as its *home context* (designated with an @-sign). Initially, the object's home context is also its *current context* (designated with a dot). References to other objects are interpreted relative to the object's

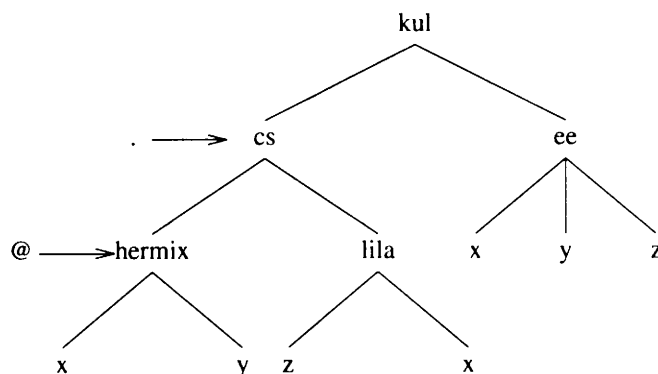


Figure 2: Example of Comet Name Space

current context, or relative to its home context (when starting with @/). An object may change its current context.

Figure 2 illustrates the name tree of a university, as seen by object *x* with home context *hermix*, and current context *cs*. Object *x* refers to object *z* within context *lila* with the names *lila/z* or *@/./lila/z*. Note the use of UNIX-like pathnames to traverse the namespace. Relative hierarchical names are well-suited to identify objects in an open distributed system:

- Hierarchical names directly reflect the relations between the objects in the distributed system, and introduce structure to the namespace. This makes them immediately useful to applications.
- Relative names have no need for a global reference point in the hierarchy (no *root context*). The hierarchy of naming contexts is always interpreted relative to a per-object reference point. The absence of a root context makes it easy to extend the name space, which is important in a dynamic environment like an internet.

3.3. Name Transformations

Because names are relative to the home or current context of the object that uses them, it makes no sense to pass them *as is* to other objects. When passing names between objects, the COMET system therefore transforms the names so that they are again relative to the object that receives them.

Name transformations are carried out as follows. All names used by the sender object (*S*) are made relative to the sender's home context. When passing them to the receiving object (*R*), they are transformed to make them relative to the receiver's home context. This transformation is performed by the COMET system with the help of the Δ_{RS} -*path*, which is the path through the name tree from the receiver's home context to the sender's home context. The Δ_{RS} -*path* is obtained when locating the receiver object.

Figure 3 illustrates the name transformation process. The name *@/./lila/z*, as used by object *x*, is transformed to make it relative to the home context of object *z* within context *ee*. The transformed name is obtained through concatenation of the Δ_{zx} -*path* with the name originally used by object *x*. The concatenated name is furthermore reduced to eliminate redundant components in the context path. Object *z* thus receives the name

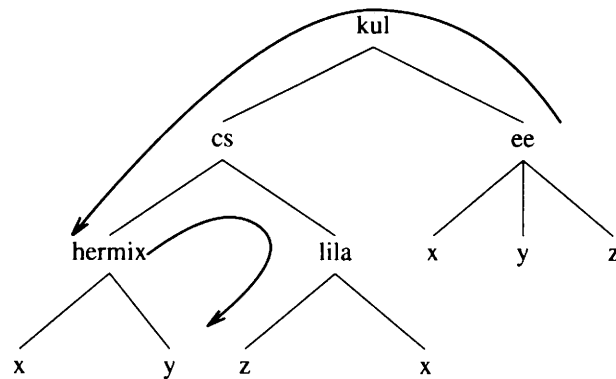


Figure 3: Name Transformation Mechanism

```

reduce ( concat (  $\Delta_u$ , @/../../lila/z ) )
=      reduce ( concat ( @/../../cs/hermix, @/../../lila/z ) )
=      reduce ( @/../../cs/hermix/../../lila/z )
=      @/../../cs/lila/z

```

4. Locating COMET Objects

When we want to invoke an object, we need a mechanism to map the object's name to its current address.[†] Mapping object names to addresses is complicated by the fact that objects are mobile, i.e. object addresses can change during the location process. A *location scheme* describes how the mapping of names to addresses is performed.

4.1. Classification of Location Schemes

Location schemes can be classified according to the basic algorithm used to map object names into addresses. The following basic mechanisms can be distinguished:

- **Direct Mapping**

This is the case where the object name fully specifies the actual address of the object referred to, i.e. we have a location-sensitive name. This scheme only makes sense in environments with immobile objects.

- **Table-based Mapping**

Every object in the distributed system maintains a table with correct <name,address> bindings for every referenced object. Locating an object consists of a simple lookup operation in this table. When an object moves to a new location, all tables have to be updated to reflect the address change.

- **Chaining**

Every object in the system maintains a table with <name,address> bindings for every referenced object. When an object migrates to another node, it leaves a *forwarding address* at the source node, specifying the object's new address. Locating an object starts at the initial address specified in the table, and follows the chain of forwarding addresses, until the object has been found. A drawback of this scheme is its vulnerability to

[†] An object address is a <node, objectid> pair, that specifies the node's transport address and an identifier that specifies the object on that node.

node crashes, which may break the chain of forwarding addresses.

- **Broadcasting**

This location scheme broadcasts the object name to all nodes in the system. The node that contains the object responds to the broadcast, and provides the object's current address. This technique works well for small local area networks, but has serious scaling problems in large-scale environments.

- **Name Server**

All <name,address> bindings are stored in a name server that resides at a well-known address. Locating an object now means querying the name server for the object's address. This location scheme depends on high availability of the name server.

Each basic location strategy has advantages and drawbacks, but none of them is suitable (in its pure form) in an open distributed environment. Therefore actual location schemes use hybrid techniques, that try to combine the advantages of several approaches.

4.2. The COMET Location Algorithm

4.2.1. Basic Algorithm

In COMET we associate a *Context Manager* with each naming context. A context manager is an object that maintains <name,address> mappings for all objects with a primitive name in this context. The entire set of context managers in the distributed system functions as a distributed name server. They are the glue that keeps the name space together. Every object in the distributed system knows the address of its home and current context managers.

When an object wants to resolve a name, it passes the name to the *location manager* component of the local COMET kernel. The location manager contacts the object's home or current context manager to obtain the address corresponding to this name. If the name is a primitive name in this context, the object's address can be immediately obtained. Otherwise the request is passed to a neighbouring context manager for further resolution.

The major advantage of this approach is that it scales well. For most mappings only a few context managers need to be contacted, even if the name space becomes very large. This is due to the fact that we use relative names (relative to an object's current or home context). Since most computations are clustered (i.e. the interacting objects are located in neighbouring contexts), relative names will generally be short, and few context managers are involved in the location phase.

The basic location algorithm finds the requested mapping as long as the involved context managers are available. High availability of context managers is thus essential. This is realised by replicating the context managers. Replicas are updated using a lazy updating scheme, i.e. their information may be temporarily inconsistent, but will evolve to a consistent state.

4.2.2. Optimising the Location Algorithm

The basic location scheme provides us with a correct <name,address> mapping, but at a considerable cost. Contacting context managers on every object reference is an expensive operation, especially in an inter-

net environment. To make things even worse, there is no guarantee that the obtained address remains correct when we finally use it. If the involved object migrates the address used will be invalid, and invocation will fail. In that case the basic algorithm must be applied again to obtain a new <name, address> mapping. For frequently moving objects this means that several cycles through the basic algorithm may be needed before invocation finally succeeds.

We therefore need additional mechanisms to speed up the basic algorithm, starting from the following observations:

- Object are referenced more often than they move,
- Context managers move infrequently.

We distinguish between mechanisms that optimise the localisation phase, and those that speed up the actual invocation process.

- **Name caches in location manager**

The first observation immediately leads to the following optimisation technique: once an object has been located, use the obtained address directly for future references (*name cache*). Since objects are referenced more often than they move, the address obtained through name resolution will remain valid for some time. When referencing an object, the name cache is now used to obtain the requested binding directly. When the obtained address turns out to be incorrect, the basic algorithm is used to obtain a new mapping, that is used to update the cache information.

- **Prefix tables in context manager**

The second observation means that it is possible to introduce shortcuts in the basic location algorithm. Context managers could optimise the name resolution scheme by directing locate requests to managers that are not their immediate neighbours. Context managers therefore maintain *prefix tables*, containing mappings from context path to corresponding manager address. When the basic strategy is used to locate an object, a context manager first checks whether a prefix of the object name matches an entry in its prefix table. When such an entry is present, name resolution continues at the context manager specified in the prefix table, skipping any intermediary context managers. Prefix tables are updated during name resolution.

- **Invocation speedup**

Whenever an object migrates, the information in the name caches becomes invalid. This is only detected during invocation. To avoid the overhead of reverting to the basic algorithm to obtain a new valid address, we can introduce the following optimisation. Whenever an object moves to a new location, we keep a *forwarding address* at the old node. The chain of forwarding addresses is traversed during invocation when the name cache refers to an out-of-date address. Note that the use of forwarding addresses introduces residual dependencies. When the chain is broken, we have no alternative but to revert to the basic location algorithm. The impact of residual dependencies can be reduced by updating the name cache when a more recent address is obtained.[†]

[†] To determine whether one address is more recent than another, additional information is needed. We therefore extend the address format with a migration stamp, that is incremented each time the object migrates.

The above-mentioned optimisations ensure that the basic location algorithm (which is expensive) is rarely used. In the majority of cases the optimisations result in inexpensive object localisation and invocation.

4.3. Organisation of the Location Manager

The location manager is responsible for resolving object names to addresses, possibly with the help of context managers. It maintains the name cache and forwarding address chain to speed up handling of locate requests and invocations. Internally, the location manager uses two types of tables to maintain its information base: per-object *name tables* and a node-wide *address table*.

- **Address Table**

The address table maintains the location of objects local to this node, or that once resided at this node, but have now migrated (forwarding chain). Furthermore it maintains the cache entries for objects referenced from this node. Entries in the address table are indexed by an object identifier, that is part of each object's address. For local objects, the stored location is a pointer to an object descriptor. For migrated and cached objects, their remote address is maintained.

Each entry in the address table contains the following information:

- ◆ Object identifier,
- ◆ Object name, relative to the node context,[†]
- ◆ $\Delta_{RS} - path$ from object's home context to node context,
- ◆ Object location.

The address table thus contains the <name, address> bindings for all objects known at the node, with names relative to the node's context.

- **Name Table**

Every object on a node has its own *name table*, which maintains information about all referenced objects.

When an object is first invoked, the location manager searches the node's address table to see whether an entry for this object is present. If so, the referenced object's address is immediately available. Otherwise, the object's home or current context manager is contacted to obtain the requested <name, address> binding. The obtained information is entered into the address table and the object's name table.

Name table entries maintain the following information:

- ◆ Referenced object name, relative to the invoking object's home context,
- ◆ $\Delta_{RS} - path$ from referenced object's home context to invoking object's home context,
- ◆ Index to corresponding entry in the node-wide address table.

The use of separate name and address tables serves two purposes. First, when one object on the node obtains a <name, address> binding, the information is entered into the address table. This makes the bind-

[†] Associated with every node is a node context, that appears in the name space. This makes it possible to refer to nodes with a logical name (see section 5).

ing immediately available to all other objects on that node. Second, the name table makes it easy to determine which references are used by a particular object on the node. When migrating the object, we can use this information to update the address table on the destination node.

4.4. Customising Context Managers

Context managers are ordinary objects that implement the name resolution protocol. Users can thus implement their own context managers, and adapt them to their needs. The only requirement imposed is that these user-defined context managers implement the name resolution protocol. A similar technique is used in the V-system [Che84a].

Customised context managers can be very useful to realise naming gateways and object clusters.

- A *naming gateway* functions as an entry point into a different namespace. A good example is the interaction with a mail system, that uses its own naming conventions. In that case the naming gateway will manage that part of the object name that corresponds to the foreign naming conventions.

When locating such a hybrid name, the naming gateway will provide its own address. Subsequent invocations will thus arrive at the naming gateway, which can pass them on to the mailing system.

- *Object clusters* can be used to implement very small objects, without the overhead associated with normal COMET objects. An object cluster is essentially a group of objects that is considered as an indivisible entity by the general mechanisms. The object cluster functions as a context manager for all its members, and provides its own address when members are located. Invocations directed to a cluster member will thus arrive at the object cluster, which can then pass them on to the corresponding member, using internal invocation mechanisms.

Since the address obtained through name resolution now refers to the naming gateway or object cluster, the address does not suffice to identify the ultimate destination. The solution used in the COMET system is to extend the object address format with an opaque field. This opaque field can be set by the naming gateway or object cluster. When an invocation arrives, the naming gateway or object cluster uses the information in the opaque field to identify the ultimate destination.

5. Location Independent Object Invocation

When a source object invokes an operation of a destination object, it uses the services of the *invocation subsystem* of the local COMET kernel. The invocation subsystem is responsible for delivering the operation parameters to the destination object, and for returning the results of the operation to the source object. This is realised by *invocation managers* that reside on every node in the distributed system.

The invocation manager interacts with the location manager to obtain the address of the destination object. There are two cases to consider:

- **Local invocation**

The situation where both source and destination object reside at the same node poses no problems. The invocation manager obtains a reference to the destination object from the location

- 1 initiate invocation
- 2 get address of invoked object
- 3 pass parameters to invoked object
- 4 operation completed
- 5 get address of invoking object
- 6 pass results to invoking object

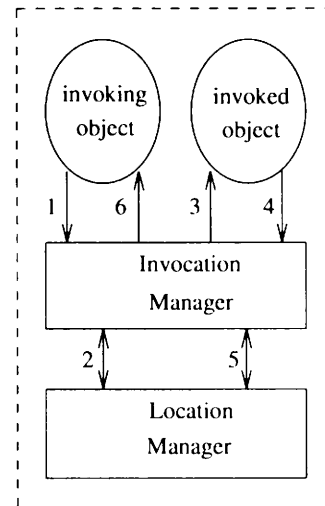


Figure 4: Local Invocation

manager. This reference is used to directly deliver the operation parameters to the invoked object.

When the operation has been carried out, the invoked object returns its results to the invocation manager. The latter again uses the location manager, this time to obtain a reference to the source object. Finally the results are delivered to the invoking object.

The invocation manager's use of the location subsystem during both phases reflects the possibility of an object migrating to another node while an invocation is in progress.

- **Remote invocation**

When the destination is remote, the request is forwarded to the invocation manager on the remote node. This process continues until the request reaches the node where the destination object resides. Here the invocation manager performs a local invocation of the destination object.

The results of the invoked operation are transferred directly to the source invocation manager, which will pass them to the source object. The invocation scenario is illustrated in Figure 5.

6. Name Space Management

The previous sections discussed how objects are named, located and invoked within an operational open distributed system. This section provides more detailed information about the management of name spaces. It discusses the following points:

- Integration of new nodes into a working system,
- Combination of name spaces.

6.1. Node Creation

The exact mechanism of starting a new node depends on the underlying guest operating system. In a UNIX environment, we implement a node as a normal process, linked with the COMET library. Starting a new node is then equivalent with starting the UNIX process.

Every node contains a *boot manager* that is activated when execution starts in that node. The boot manager is responsible for node initialisation. The first step is to initialise the COMET kernel data structures. Next the boot manager sets up a name space for the new node. The initial name space contains three objects:

- **Node context manager**

The boot manager creates a context manager on the node with the node's name. The node name is obtained from the *boot parameters*, which are typically available in a file on the underlying guest system.

- **Kernel object**

Within the node context, the boot manager creates a *kernel* object. This object represents the node's kernel. The kernel object provides operations to interact with the COMET kernel (e.g. to obtain information on available resources).

- **Init object**

Finally the boot manager creates the *init* object within the node context. This object is responsible for finalising the initialisation phase. Its executable image is part of the boot parameters.

The *kernel* and *init* objects are created with home and current context set to the node context. Once the *init* object is activated, the new node is up and running. The *init* object will then typically perform the following actions:

- Merge this node's name space with the name space of the open distributed system,
- Create other objects, s.a. a shell to interact with the user.

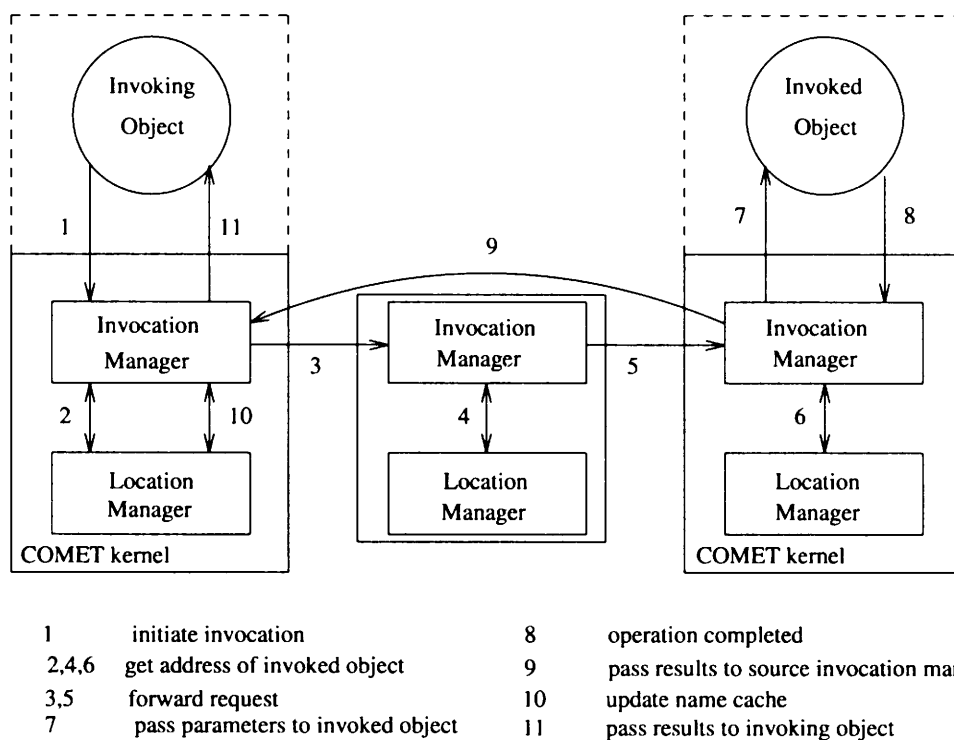


Figure 5: Remote Invocation

6.2. Merging and Splitting of Name Spaces

Name space merging combines disjoint name spaces into one naming hierarchy. A typical example is the integration of a new node into the open distributed system. Another example is the merging of the name spaces of two different organisations. Name space merging is accomplished by attaching a context as a sub-context of some other context.

Name space splitting partitions the name space into two disjoint name spaces. It is accomplished by breaking the connection between a sub-context and its enclosing context.

Context managers provide the following operations to support name space merging/splitting:

- The *create-binding* operation takes a primitive name and address as parameters. It adds the <name,address> binding to the context manager. Name space merging is accomplished by binding the name ".." in the sub-context manager to the address of the enclosing context manager, and by adding the <name,address> binding for the sub-context manager in the enclosing context manager.
- The *delete-binding* operation takes a name as a parameter. It removes the binding for the specified name from the context manager. Name space splitting is accomplished by removing the binding for ".." in the sub-context manager, and the binding for the sub-context in the enclosing context manager.

When performing name space merging, the *create-binding* operation is invoked using location-dependent invocation. This is necessary since the context managers themselves are needed to implement location-independent invocation.

7. Conclusion

Object naming and location schemes are of vital importance in an Open Distributed System with mobile objects. They form the basis for all interactions between objects in distributed applications.

COMET's hierarchical naming scheme reflects the hierarchical structure of the real world. It provides names that are meaningful to applications. The absence of a root context makes it easy to dynamically adjust the name space. Merging or splitting of namespaces poses no problems.

The associated location scheme reflects the fact that communication overhead is a dominant factor in an internet environment. It is based on a distributed <name,address> database, enhanced with alternative strategies, thus providing very efficient name resolution in a majority of cases. The proposed location scheme scales very well, because we use relative names. Since most computations are clustered, these names will generally be short, and few context managers will be involved.

The COMET naming scheme is designed to be open. Customised context managers can be added that implement different naming conventions, or alternative location strategies.

The COMET naming and location schemes are specifically designed for use in a large-scale internet environment. As such, they are well-suited to support the multitude of objects that appear in open distributed applications.

References

- [Bla90a] A. P. Black and Y. Artsy, "Implementing Location Independent Invocation," *IEEE Transactions on Parallel and Distributed Systems* 1(1), pp. 107-119 (January 1990).
- [Che84a] David R. Cheriton and Timothy P. Mann, "Uniform Access to Distributed Name Interpretation in the V-System," Research Report 4/045, Computer Science Department, Stanford University (December, 1984).
- [Das88a] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe, "The Clouds Distributed Operating System," *Proceedings 8th International Conference on Distributed Computing Systems*, San Jose, California, pp. 2-9 (June 13-17, 1988).
- [Gur86a] Robert F. Gurwitz, Michael A. Dean, and Richard E. Schantz, "Programming Support in the Cronus Distributed Operating System," *Proceedings 6th International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, pp. 486-493 (May 19-23, 1986).
- [Moc81a] P. Mockapetris, "The Domain Name System," *Proceedings IFIP 6.5 International Symposium on Computer Messaging*, Paris, France, pp. 31-40 (April 1981).
- [Moo90a] H. Moons and P. Verbaeten, "A Portability Environment for Distributed Application Programming," *ISMM International Symposium on Mini- and Microcomputers and their Applications*, Lugano, Switzerland (June 19-21, 1990).
- [Mul85a] S. Mullender, "Principles of Distributed Operating System Design," PhD Thesis, CWI, Amsterdam (October 1985).
- [Mul86a] S. J. Mullender and A. S. Tanenbaum, "The Design of a Capability Based Distributed Operating System," *The Computer Journal* 29(4), pp. 289-299 (August 1986).
- [Pop85a] G. J. Popek and J. W. Walker, "The Locus Distributed System Architecture," in *The M.I.T. Press* (1985).
- [Sch86a] Richard E. Schantz, Robert H. Thomas, and Girome Bono, "The Architecture of the Cronus Distributed Operating System," *Proceedings 6th International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, pp. 250-259 (May 19-23, 1986).
- [Sin89a] Pradeep K. Sinha, Kentaro Shimizu, Naoki Utsunomiya, Hirohiko Nakano, and Mamoru Maekawa, "Network-Transparent Naming and Locating in Distributed Operating Systems," Technical Report 89-033, Department of Information Science, University of Tokyo (November 1989).

Communicating Database Objects

Agnes Hernadi Elod Knuth

Ferenc Jamrik Gabor Janek

Computer and Automation Institute

Hungarian Academy of Sciences,

Budapest

h792her@ella.hu

Abstract

We address the problem of establishing communication between databases created independently with no preliminary agreement concerning the adaption to an appropriate convention. An interactive algorithm is provided for databases with a simple data scheme in aid of accomplishing a partial mapping and based on this, data communication protocols automatically.

1. Introduction

In reality databases are mostly created independently of each other. Usually it turns out only afterwards that they contain information which is of value for some other databases even though those (may) use different concepts for the same thing.

Using a simplified data model we found it was possible to provide an interactive algorithm for the transparent use of such databases by a later mapping. We connected databases created by our experimental database system called Data Gallery [Her89a, Knua] to solve the problem addressed above.

By nature Data Gallery is an unusual, non-language-oriented approach to manage loosely structured ad hoc information like personal, office or management data using visual aids. Unlike other approaches of the same kind interacting with databases [Shia, Kan88a, Rus88a, Roh88a] and [Rog88a, Dud89a, Ang89a, Tsu89a, Cze89a], Data Gallery introduces a single new concept called Data Picture to cover all the traditional database functions and as we shall conclude in the paper it turned out that the same concept is rather suitable to be a communication media between independent databases. What is more, in case of an environment supporting distributed object management [Rym90a], the Data Pictures could serve as media in sessions ensuring shared access.

2. Database objects

2.1. The basic data model

Since cognitive research did not draw strong conclusions that inexperienced users can analogically import the understanding of formal models being fairly complex in nature [Pol84a] we intentionally limited the model underlying the experimental system implemented.

A version of the binary-relationship model was chosen for our experiments consisting of a set of atoms and binary connections over them. An atom is considered a type-value pair while a connection a relation name and an unordered pair of atoms. All the objects mentioned (types, values and relation names) are strings.

2.2. Data Pictures

Data Pictures serve as the basic unifying objects of interaction. They give a relatively simple general framework capable of

- Performing all traditional database functions formerly associated with particular forms separately (like entry forms, query specifications, reports, pieces of the schema etc.); and
- Learning the schema from examples.

As all user actions are associated with Data Pictures exclusively this approach results in a kind of a multifunction database editor still possessing all conventional database functions, though in a different manner. The introduction of editing functions adds some unusual features to the conventional spectrum for instance moving, copying, composing and decomposing. These prove especially useful when manipulating individual data objects.

During dialogues arbitrary number of Data Pictures – each presenting a self-contained interactive working context – may simultaneously be present on the screen. As Data Pictures are sensible to changes of the database caused by operations performed on another Data Picture we are able to observe the effects of those operations from various inverted viewpoints.

Technically Data Pictures show textual information in a hierarchic arrangement of data lines. Semantically they can represent entry forms, query specifications, reports, pieces of the schema etc.

To each Data Picture three partly independent properties can be assigned. These are called as Valid, Filled and Saturated. Validity expresses that the Data Picture is in all respect consistent with the actual database contents. The other two properties express completeness marks on data and scheme level, respectively.

The Picture Mode is perhaps the most important concept our approach suggests. It could also be considered a far more general idea when models and viewpoints, facts and reflections are distinguished in any area of interactive computer applications. Any Data Picture can be put in each of the following three modes determining its way of interaction with the database. In the Free Mode a Data Picture can be freely transformed without any consequence. It may lose even its validity. When manipulating a Data Picture in the Check Mode any modifying action which contradicts its validity is refused. The Force Mode also

preserves the actual Data Picture's validity but by enforcing a database update whenever any action in it contradicts the Picture's validity.

2.3. Operations on Data Pictures

As Data Pictures are hierarchies of data lines three classes of operations are provided, namely:

- Token Operations acting on elements of individual data lines;
- Line Operations acting on a whole data line;
- Picture Operations acting on arbitrary subhierarchy of the one constituting that Data Picture.

Individual operations can be found in [Her89a]. However, we cannot avoid mentioning the following operations:

- *Unfold*: expounds the interconnections of an individual object at a given point of the hierarchy one more level of detail and helps to learn for possible orientations to be followed. This is especially useful on browsing.
- *Cut/Copy/Paste*: using these operations Data Pictures can be composed and decomposed.
- *Evaluate*: considers the Data Picture a query specification and fills it recursively in all possible valid ways.

An operation always acts according to the Picture Mode selected. Invocation of an operation should always be preceded by a selection.

2.4. The Data Gallery

As it was hinted at, the information base splits into two parts: the database, the conventional part, and the repository of Data Pictures called the Data Gallery.

The Data Gallery is divided into two regions namely the so-called Exhibition where all Data Pictures are maintained on all changes of the information base, and the Archive storing Data Pictures which are still valuable in a sense but which are not maintained regularly any longer.

In the Exhibition you can find at least two unremovable read-only Data Pictures. The Data Picture named Types contains all the existing types while the Data Picture named Schema contains all the existing relationships. These Data Pictures or any of their parts can be freely copied however.

For a particular database any number of Data Galleries can be created. (People may use their own Data Pictures to handle the same data.) Moving and copying Data Pictures between the regions of different Galleries associated with the same database should fulfil the same requirements as within a Gallery, concerning the validity of a Data Picture. Although Data Pictures can be moved and copied between the Data Galleries of different databases, the transferred Data Picture is usually invalid and it is left to the user's discretion how to exploit the contents of such a Data Picture.



3. Communicating Data Galleries

3.1. Data Channel

The concepts introduced, however, afford us an opportunity to post-link independent databases in a virtual manner and to use them in a transparent way. The communication of databases created independently with no preliminary agreement concerning the adaption to an appropriate convention can be realized by so-termed Data Channels.

The interactive algorithm described later takes two database schemas and results in a partial mapping between them. This mapping is converted then to a data communication protocol called Data Channel connecting the two databases through any of their Data Galleries. Two databases can be connected by any number of Data Channels.

A Data Channel can be used in the following two ways:

- *Sending/Receiving Data Pictures.* This works like an ordinary electronic mail system with the exception that Data Pictures are not simply sent to the Exhibition region of the receiver Gallery but even converted into the terms of the receiver database. This conversion is carried out according to the mapping defined on creating that Data Channel.
- *Augmenting the scope.* Using a Data Channel one can directly access another database (if permission is granted by the permission system). In this case the scope of data access is extended to the other database in a transparent way.

3.2. Building a Data Channel

The location where the Data Channel object is to be created can be identified in the usual way.

A menu helps to select the databases to be connected. Let A and B denote these databases, while ATypes and BTypes denote the Data Pictures containing all the types defined in A and B respectively. (These Data Pictures are exhibited in each Gallery of the corresponding database under the name Types.)

3.2.1. Mapping algorithm

Initialization

ATypes and BTypes are opened simultaneously and assigned to be source objects to the algorithm.

Initial Designation (user action)

Select one of the types in both Data Pictures to be mapped to each other.

Initial Reorganization (system response)

Those types not mapped disappear from the Data Pictures. The ones mapped are marked (appear in a distinguished way) and are Unfolded. (The Unfold operation in this case displays all the relations interpreted on the selected type as a subhierarchy of the latter.)

General Designation (user action)

Select a pair of the types (one in both Data Pictures) from the subhierarchy to be mapped to each other. By this a pair of relations to be mapped to each other is selected as well.

Constraints:

- If the pair of types has already been mapped to each other, the selection will be accepted;
- If neither of them have been mapped to any other type, the selection will be accepted;
- Otherwise the selection will be refused.

General Reorganization (system response)

The order in the subhierarchy changes: the designated types appear marked on the top (placed under the last selected).

Cycle

Repeat *General Designation/General Reorganization* until "End Cycle" is chosen to indicate that no further types are intended to be mapped at the given level of hierarchy.

Reorganization after End Cycle (system response)

Types not mapped at the given level of hierarchy disappear. The first pair of corresponding types will be Unfolded in a somewhat modified way: relations already mapped are not displayed.

Do *Cycle* for each node.

It may happen that in one of the Data Pictures this modified version of Unfold results in no change. In this case the result of the operation is cancelled and the next line of the same level should be Unfolded. If there is no such line, the algorithm returns to the next lower level to Unfold the next corresponding pair of types at that level.

3.2.2. Characteristic Data Pictures

The above algorithm always results in a pair of Data Pictures corresponding to each other line-by-line. If these isomorphic Data Pictures are not empty, the algorithm is considered to be successful and the Data Channel is created and stored for later use.

So we can say that a Data Channel is defined by a pair of isomorphic Data Pictures each belonging to the different databases to be post-linked virtually. This pair of Data Pictures, called Characteristic Pictures, appoints at once both the database subschema of the database to which they belong separately, and a one-to-one correspondence between the elements of these subschemas. As a result, on "peering through" the Data Channel from one of the databases into the other, we get a partial view of the latter. In this manner we can look at a part of a database implemented in a different database by defining an appropriate Data Channel, or rather a pair of Characteristic Pictures. (The other part of that database not mapped by the Data Channel remains, of course, invisible on peering through the Data Channel.)

Creating the Characteristic Pictures is a computer-aided interactive activity, intentionally not completely automated.



4. Conclusion

Connecting independent databases is a widely examined problem nowadays. Most of the solutions available restrict themselves to mapping various heterogeneous database schemas to each other. However, there is another problem in practice, namely providing semantic mapping between the meaning of the concepts represented in the databases physically connected.

Our project addressed the latter problem area. In nature, creation of a semantic mapping cannot be fully automated. This is why our fundamental algorithm is an interactive one.

Naturally, the algorithm and the prototype system we built is limited by the data model used. The principle, however, can be easily generalized for the data models of most of the available commercial databases, though they would result in much more complicated interfaces. Our subsequent work is to extend the communication capabilities of our experimental system towards popular database management systems.

The project reported here has been implemented in UNIX System V environment under the UA window manager on interconnected MC68010 based computers.

References

- [Ang89a] M. Angelaccio, T. Catarci, and G. Santucci, "QDB: A Fully Visual System for E-R Oriented Databases," pp. 56-61 in *Proceedings of 1989 IEEE Workshop on Visual Languages*, Rome, Italy (October 4-6, 1989).
- [Cze89a] Bogdan Czejdo, David Embley, and Venugopal Reddy, "A Visual Query Language for an E-R Data Model," pp. 165-170 in *Proceedings of 1989 IEEE Workshop on Visual Languages*, Rome, Italy (October 4-6, 1989).
- [Dud89a] Tim Dudley, "A Visual Interface to a Conceptual Data Modelling Tool," pp. 30-37 in *Proceedings of 1989 IEEE Workshop on Visual Languages*, Rome, Italy (October 4-6, 1989).
- [Her89a] Agnes Hernadi, Zalan Bodo, and Elod Knuth, "Context-Reflecting Pictures of a Database," pp. 273-282 in *Proceedings of the EUUG Spring 1989 Conference*, Brussels (April 1989).
- [Kan88a] Hannu Kangassalo, "Concept D: A Graphical Language for Conceptual Modelling and Data Base Use," pp. 2-11 in *Proceedings of 1988 IEEE Workshop on Visual Languages*, Pittsburgh, Pennsylvania, USA (October 10-12, 1988).
- [Knua] Elod Knuth, Agnes Hernadi, and Zalan Bodo, "Pictures at a Data Exhibition," pp. 303-315 in *Visual Languages and Visual Programming*, Shi-Kuo Chang, Ed., Plenum Press, New York, 1990.
- [Pol84a] P. G. Polson and D. E. Kieras, "A Formal Description of Users' Knowledge of How to Operate a Device and User Complexity," pp. 249-255 in *Behav. Res. Methods Instrum. (USA)* (1984).

- [Rog88a] Greg Rogers, "Visual Programming with Objects and Relations," pp. 29-36 in *Proceedings of 1988 IEEE Workshop on Visual Languages*, Pittsburgh, Pennsylvania, USA (October 10-12, 1988).
- [Roh88a] Gabriele Rohr, "Graphical User Languages for Querying Information: Where to Look for Criteria?," pp. 21-28 in *Proceedings of 1988 IEEE Workshop on Visual Languages*, Pittsburgh, Pennsylvania, USA (October 10-12, 1988).
- [Rus88a] Bogdan Czejdo, Venugopal Reddy, Marek Rusinkiewicz, "Design and Implementation of an Interactive Graphical Query Interface for a Relational Database Management System," pp. 14-20 in *Proceedings of 1988 IEEE Workshop on Visual Languages*, Pittsburgh, Pennsylvania, USA (October 10-12, 1988).
- [Rym90a] John R. Rymer et al, "PANEL: OOPSLA Distributed Object Management," pp. 331-345 in *Proceedings of ECOOP/OOPSLA '90* (October 21-25, 1990).
- [Shia] Yukari Shirota, Yasuto Shirai, and Tosiya L. Kunii, "Sophisticated Form-Oriented Database Interface for Non-Programmers," pp. 127-155 in *Visual Database Systems*, T. L. Kunii, Ed., North-Holland, Amsterdam - New York - Oxford - Tokyo, 1989.
- [Tsu89a] Kazuyuki Tsuda, Masahito Hirakawa, Minoru Tanaka, and Tadao Ichikawa, "Iconic Browser: An Iconic Retrieval System for Object-Oriented Databases," pp. 130-137 in *Proceedings of 1989 IEEE Workshop on Visual Languages*, Rome, Italy (October 4-6, 1989).

UNIX in Novell Environment

Gabriella Ivanka Gyorgy Leporisz

*Computer Research and Innovation Centre
Hungary*

Abstract

Through the new concept of their Open Network Architecture with NetWare 386 as well as the agreement made with IBM to make a closer communication between SNA, OS/2, UNIX and the NetWare, Novell network operating systems go on strengthening their leading position on the world market.

The lecture intends to present the transparent integration facilities of the NetWare and UNIX systems (through NetWare 3.11 and NetWare NFS) and will also outline the declared plans of Novell to deepen the communication between the two systems.

An International Hotel Reservations System Using Loosely Coupled UNIX Systems

Gary M Bilkus

BLiX Limited

gary@utell.UUCP

Abstract

This paper describes the design and implementation of the latest version of an international hotel reservations system.

The system allows reservations staff in offices around the world to make reservations as though they were permanently on-line to a central information database, with up-to-date information. In fact, each office runs independently, and the International Packet Switching Service is used to communicate with other locations on a batched up basis.

After a brief discussion of the business needs and the hardware environment, we will look at the resulting data and transactional requirements. This leads to a discussion of the in-house infrastructure, written in C++ under UNIX. Finally, an analysis of performance will be given.

1. Introduction

This paper describes the implementation of a new computer system for a large independent Hotel Representation company. Travel Agents or members of the public can call its offices around the world to make immediately confirmed hotel reservations at almost 10,000 properties. In addition, the company operates a sophisticated prepayment scheme which allows payment in local currency by the client, and offers immediate commission payment to the travel agent.

As well as handling telephone bookings, the system is connected to most of the world's major electronic booking systems, including all of the world's largest airline reservations systems, Prestel in the UK, and similar systems elsewhere. Indeed, anyone who books a hotel through a travel agent with airline connections may well have been a customer without knowing it.

Before 1983, the company used the GEISCO timesharing service to handle its DP requirements. Information about hotels was disseminated to offices on a weekly basis in the form of a manually produced microfiche. Staff would use the fiche to check hotel information, availability etc, and fill in manual forms which were batched, and processed overnight by GEISCO. Confirmations for hotels were printed in

the office nearest the hotel, and the head office in London obtained daily and monthly accounting and management reports.

In 1983, development began of a fully computerised system. The first stage was to automate the microfiche production and the management and accounting functions. Then, a reservations program was written to allow the automation of the hotel selection and booking functions. The entire system was written in C on PDP11s running UNIX System III and later System V.0

Offices around the world ran independently of each others, but once a day each office would perform a file transfer to the London head office. At that time, London would receive all bookings made, and would send all updates to hotel information. This of course meant that bookings were made on the basis of information which was up to 2 days out of date, and it could be even longer before the hotel knew of the booking. Nevertheless, it was a considerable improvement over the weekly fiche, and compared favourably with the alternatives available from competitors.

During the course of the next few years, the market became much more sophisticated. Major airlines started offering hotel reservations services to their on-line agents, and there was pressure to improve the frequency of updates and the amount of data available. The existing system was modified, adapted, and hard-coded, but it became clear that it was only a matter of time before it ceased to be viable, and in late 1988 the decision was made to re-implement it. The result of that reimplementation is the subject of the rest of this paper.

2. The Business Needs for which the System was Designed

A small (3 person) design team, headed by the author, spent 3 months devising a strategy for the new system. The new system had to remain broadly upward compatible with the existing system described above. In addition, it needed to satisfy a number of other requirements.

- **The system must be able to run essentially stand-alone on cheap hardware in isolated or badly served areas.**

One of the features which distinguishes the company from its competitors is its ability to offer its services in small markets and in developing countries. This put a great strain on the design. It must be possible to obtain the necessary equipment in places like Bombay, Sao Paulo, and indeed Eastern Europe while remaining cost effective in small markets like New Zealand. This requirement alone ruled out any possibility of a mainframe and leased lines.

- **No pre-set limit on the amount of information held about a hotel.**

The old system had fixed length limits on the amount and format of the information held about hotels. Only full-price rooms were available, and no more than six categories of accommodation could be offered. These limits originated from what could be put on a microfiche frame, and were hopelessly out of date. We felt that simply to increase those limits to a higher number was asking for trouble in the future, and instead we decided to allow arbitrary amounts of all types of data per hotel. This decision probably had a greater effect on the design of the user interface than any other, since it meant that The application had to support displays of potentially huge lists in a sensible way.

- **Operational flexibility in controlling the frequency of update to remote offices.**

Ideally, any change of rates, availability or whatever notified to the company should take immediate effect in all offices around the world. However, in practice it is still not necessary to be quite that efficient, especially when the destination office is small. It is vital to be able to control costs by batching up several changes and sending them all at once. On the other hand, the traffic in some areas may justify a dedicated line, in which case data should be transferred immediately.

- **Offices should not be dependent on working comms to function.**

This requirement relates to the earlier one of being able to function in difficult locations. The company needs to continue to be able to take bookings even if all communications failed for an extended period. Also, each location needs to be able to restore itself to full working order after a hardware failure without other systems being involved.

- **Except in major offices, no computer literate operations staff should be required.**

This requirement had a significant effect on a number of areas. The old system had reached the point at which it was often necessary to have operators in London or Omaha connect to other office machines in order to solve local problems. Often, the integrity of the system itself was at risk. Our new design was required to be able to function reliably, even after hardware problems, with minimal danger of unskilled operators causing irremediable damage.

Other Factors

In addition to the above, there were a number of other important factors we wanted to take into account. Wherever possible, we wanted to be able to use existing hardware, and we certainly wanted to preserve our hard-won expertise in running UNIX based systems. At the time, almost all of the offices were running Motorola Delta series systems and UNIX System V.2 or V.3. However, the largest locations – London and Omaha Nebraska had recently moved to Sequent Symmetry running Dynix. The Sequents are really BSD machines, so it was important that our software should run properly under BSD and System V kernels. Furthermore, we were keen to assure ourselves that our eventual implementation would not seriously restrict our choice of UNIX hardware, which needed to be cost effective in locations ranging from the one-person Zurich office, to the 200 person Omaha office.

From a software development point of view, we were keen to ensure that our approach was easy for new recruits to learn, so that they could be productive quickly. One major problem with the existing system was that it was almost impossible to obtain a reasonable idea of how it worked without spending months learning by osmosis.

In particular, we had become increasingly disillusioned with C as a programming language for our purposes. The limitations of C as an application development language are well known, and need not be addressed here. Suffice it to say that we felt that it was appropriate, at the very least, to examine alternatives to the continued use of C as a sole development language.

3. Third Party Package Evaluation, and its Conclusions

By this time, our outline specification, in the form of a user document, had been discussed and approved by the company, and we had a few more months to finalise our software development strategy. We decided to approach various vendors of database and/or communications software to see whether we could use their products effectively.

It became clear quickly that one particular area was going to cause major problems. For every hotel, and every valid room/rate combination in that hotel, it is necessary to store the availability of the accommodation for every date from now, typically up to a year in the future. Availability is usually either open (yes) or closed (no), but sometimes other values are required – there are about 10 possibilities in all. In addition it is necessary to distinguish dates which are known to be open from dates for which no information is yet available.

To store that information in a relational table would therefore require typically records of length 10 + bytes (4 for the hotel, 2 for the room, 2 for the date and 1 for the availability itself) allowing 1 overhead. The number of records would be 365 for a year * 10 roomtypes per hotel * 10,000 hotels. Thus, the storage requirement for this table alone would be at least $365 * 10 * 10000 * 10$ or about 400 megabytes. Allowing for indexes, and other bookkeeping and we were looking at about 1GB for this table alone. Furthermore, that table is the most volatile in the system, and would require frequent backing-up.

Against that number, we knew that if we stored the availability in a packed binary record, using sensible compression techniques, the same information per room would occupy about 50 bytes for a year, resulting in a total table size of $50 * 10000 * 10$ or 5MB – say 20MB including indexes etc. Furthermore, our knowledge of the requirements of the application made it certain that availability would need to be accessed only within the context of a preselected hotel record. Other parts of the system would have been subject to similar overheads although not quite as dramatically.

When we broached this problem with the vendors, we were offered the chance of storing the information as a BLOB of binary data, and packing/unpacking it ourselves. This seemed like a viable solution, but it would have meant losing the ability to write our code in their database access languages, SQL or whatever, and would force us to revert to C for what was likely to be a large part of the code.

Another problem was that of the distributed nature of our application. It was already clear that none of the database vendors could offer us a means of allowing distributed transactions where most of the systems were not permanently on-line or available in real-time. Fortunately, as described later, we had identified a way in which individual transactions could be safely processed on one machine and transmitted subsequently to others without compromising the integrity of the system as a whole. This would allow us to use a standalone database package on each machine, with well defined hooks for distributing needed data to other machines. We hoped to find a system which would integrate with our chosen database to allow reliable message passing between systems which were not permanently on-line to each other.

Despite going as far as advertising our requirements, we were unable to find what we wanted in that area. There were a number of packages available which offered what amounted to sophisticated mail and which could perhaps have been kludged into shape. Indeed, we pur-

chased evaluation copies in one case to see what we could do. However, none of the available alternatives offered us much over what we felt we could do ourselves, and all were rather pricy.

It had become clear that what we really wanted wasn't available from anybody yet, and the cost of licenses for what we could get would be hard to justify in terms of reduced development time or better running. In the meantime, we had become very interested in the possibility that C++, now at last becoming widely available, might allow us to escape the worst problems of C, while maintaining the degree of control we required. After much discussion it was decided to adopt C++ as a base and to develop in-house an infrastructure on top of which the application could be built. Accordingly, the project team split for a while into two groups. The author headed an infrastructure team, while a colleague headed an application design team. Communications between the two teams was maintained by all members of both teams sharing an open-plan office.

4. The Methodology

At the same time as we decided to adopt C++ and our own infrastructure, we had finalised the details of our methodology for the application development. Each member of the team would be involved in several stages of the process – nobody would be programming exclusively, but everybody would do some coding. Following a successful in-house course on structured analysis and design, we began the process of turning our requirements into entity-relationship diagrams for the data, and higher-level dataflow diagrams for the processing. The eventual result of these was a set of tables (about 200 in all) and a set of program module outline specifications.

Meanwhile, the infrastructure development had resulted in a series of class libraries, and proforma examples of how to use them to implement each of the different categories of program module which the applications team had identified. In many cases, it turned out that the entire module could be specified as a simple parametrized form, which could then be coded by rote, resulting in a lot of boring but very productive work. In other cases, the module coding was more complicated, but was seldom more than a day or so's work.

It would be nice to claim that this hybrid approach had been fully worked out in advance, but it was not the case. During the year which the design process took, there was a constant to and fro between the application and infrastructure teams trying to delineate the exact boundaries. Little was specified in advance, and there were a few awkward problems when important areas were in danger of slipping through the gaps.

Nevertheless, the combination of bottom-up infrastructure and top-down design worked well in practice, and probably resulted in a better eventual interface boundary than could have been planned in advance.

5. The Way Data is Distributed

The data tables which resulted from our analysis fall into several categories, depending on how they are distributed among the various machines worldwide. We have:

“Static” global tables: e.g. lists of currencies, countries etc

Hotel related tables

Reservation related tables

Local administration tables (passwords)

Internal control tables (routing, machine PSS addresses etc)

Temporary tables (pending reports)

Copies of the static global tables are held on each machine. These tables are updated only rarely, and updates almost always consist of additions rather than changes.

Copies of the hotel related tables are kept on each machine too. Every record in these tables relates to a specific single hotel, and only those machines in offices which have booking agreements with that hotel keep that information.

The reservation related tables on each machine similarly only contain the reservations relevant to that office. However, the London office has a master machine which contains a copy of every reservation made, and uses this for management reporting etc.

Local administration tables are used by each office to set up their configuration. These tables are not visible from elsewhere.

Internal control tables are used by the parts of the infrastructure responsible for sending data to/from machines. They are like static global tables in that they are held everywhere, but they require special care when updating.

Finally, each machine will have temporary tables, containing, for example lists of telex messages which have been generated but not yet sent.

6. Transactions

All updates to the tables above are defined in terms of transactions.

One of the most important factors about our application was that it did not require multi-system access in order to guarantee the correctness of transactions. Since this is a key feature of the system, it is worth describing in more detail.

In the general case, a distributed database may require access to some or all of the systems over which it is distributed before a transaction can be checked for validity and allowed. This requirement places a heavy burden on the level of connectivity between systems, and if it applied to us would have made it impossible to operate quasi-independent systems.

Fortunately, we were able to find a way of ensuring that a transaction would be valid without access to more than one machine, by restricting the possible transactions. The idea we adopted was that of ownership. Almost all transactions on the system fall into one of two categories – hotel update or reservation. By designating a single system as the “owner” of each hotel, or reservation, we insist that all updates to that entity must be processed first on the owning machine, and only then can subsequently be sent to any other machines which need to know about the transaction.

Of course, some transactions cannot be treated in the above way. Changes to certain global tables, changes of ownership, or transactions whose legitimate originator does not own the data etc, need to be han-

dled more carefully. However, these exceptional transactions are rare, and seldom urgent, and a clumsy but workable solution was found in the form of a several stage process, which mimics two-phase commit.

One factor which simplified a lot of the analysis of transactions was the decision to handle consistency problems by using a singly-threaded transaction process. One of the consequences of not buying into a proprietary DBMS was that we lost those systems' sophisticated locking/rollback mechanisms. Instead, we created several server processes, each handling certain category of transaction which were always mutually compatible, and each processing one transaction at a time.

As part of its processing cycle, a transaction server may decide that one or more remote machines must receive transactions (either a direct copy of the current one or one or more different ones). The transaction server will generate these remote transactions, and send them via the infrastructure to the destination machine. The comms-in-out feature of the infrastructure acknowledges transactions as soon as they are queued, and guarantees that all transactions sent between any given pair of machines will be processed in the same order as they were generated. Thus, two hotel updates generated on a single machine for the same hotel, in which the second partially contradicts the first, will be transmitted in the correct order to all interested other machines.

The exact time taken for those transactions to reach each other machine depends on the frequency of communication between those machines, and on the load at the destination.

7. Audit and Backup

In any system it is important to have a properly thought-out strategy for backup of data, and for the provision of an audit record which can be used to investigate any mysterious events. Our single-threaded transaction server on each machine made it easy to incorporate a simple transaction logging feature. Every transaction is logged to a separate device (in some cases to optical disk). The transaction log also notes the times and media on which full backups are done. A special option to the transaction processor allows a transaction log to roll forward from the last available backup.

The same strategy is used to log data as it is sent from or received by the machine by the comms-in-out system described below. Care is taken to ensure that when a transaction is passed from one part to another that agreement exists about what has happened.

The most important consequence of this way of working is that an office which breaks down does not need its comms to be up in order to restore its state. This contrasts with the old system, where a lot of information could only be restored from London if a local office had problems.

8. The Infrastructure

We are now ready to describe the infrastructure from the point of view of what it offers the programmer. Unfortunately, space does not permit a full discussion of the internals.

The infrastructure can be divided into several distinct areas.

- The uimake program
- The types system and tycpp
- The file access method
- The message protocol
- The transaction processor
- Comms in and out
- Windows

The Uimake Program

The first problem we faced when moving to C++ was that C++ imposes a much more rigid requirement than C on such things as declaring functions before use and including class definitions in the right order. It became clear that the traditional include file and makefile approach was going to cause problems, and we therefore wrote a program called uimake. uimake brings to C++ something similar to the Pascal UNIT concept. Every module contains a program file and a header file, together with a unit file ending in .u . The unit file specifies which other modules are required by the current one, and whether the current module is a main program. Once modules are set up, a call of

```
uimake modulename
```

will make the module, including all necessary headers in the right order, and if appropriate linking the objects to form an executable. While not perfect, uimake has proved invaluable in keeping track of the interrelationships between the different code elements.

Another benefit of uimake is that it guarantees that modules are loaded into a final executable in a consistent order vis-a-vis their relative dependencies. This allows safely the automatic calling of static constructors in the correct order.

The Types System and tycpp

One of the major objectives of the infrastructure was to allow the storage of complex data structures as records in tables, and to allow general purpose access programs to understand those structures and display them. Unfortunately, C++ provides no way of interrogating the compiler to find out the internal structure of objects. We got round this problem by introducing a class called type, which describes a type in an accessible way. In order to generate the correct type information for the structures we were using, we wrote a simple preprocessor, tycpp, which accepts definitions of a large subset of C++ structures, and generates a module which defines the structure and a type describing it.

We also defined a class object, which consists of an arbitrary data value and a type, and wrote routines to convert objects into binary and ascii representations of their values.

The File Access Method

Built on top of the types system, we introduced a generic class called table. A table(type) inherits a value of that type together with traditional imperative style functions for updating and fetching the value from an underlying table of that value. By default, tables are held entirely in memory, but a derived class reltable allows them to be associated with disk files. The tycpp facility was upgraded to allow easy creation and definition of these tables. At this point it may help to look

```
#include "example.x"
// This file is created by uimake with all needed headers
#define definitions
// This section is interpreted by tycpp
STRUCTURE hotel
int code;
string name 30;
ENDSTRUCTURE
KEYINFO hotel_keys;
KEY 1:
code;
KEY 2:
name;
ENDKEYINFO;
TABLE hotel USING hotel_keys;
#enddefinitions
/* the rest of the file is passed unchanged to C++ */

declare(retable,hotel);

main()
{
retable_create("hotel.dbf",hotel_type,hotel_keys);
/* hotel_type is created automatically by tycpp */
/* This has created an empty table */
myhot.open("hotel.dbf");
/* myhot is now associated with the empty table file */
myhot.code = 23;
myhot.name = "TWENTY THIRD HOTEL"
if (myhot.insert() != TRUE) cerr << "Error inserting hotel 0;
....
myhot.close();
return(0);
}
```

Program 1: Example table program

at Program 1. Obviously, the system does not provide a full set of relational-style features, but in practice this was not a problem. The programming side seemed to be well served by an "old-fashioned" way of getting at the data, and the transaction server handled both concurrency control and distribution in a comprehensible way.

The fundamental benefit of what we did is that we allow fields of a table structure to themselves be tables and so on.

Thus, if the type `hotel_tab` has a field `rate` of type `table(rate_tab)` and `h` is a `table(hotel_tab)`, `h.rate` is a fully fledged table, which can be used just as though it were standalone. The way in which such a subtable is stored within its parent record can be easily customised to allow end-user invisible storage optimisation. This is of course how we solved our problem with the storage of availability.

The Message Protocol

One of the most annoying "features" of UNIX is that it provides many different ways for processes to communicate, but no standardised, efficient, reliable message passing scheme. Shared memory, semaphores, message queues, pipes, streams drivers, and sockets are all contenders, but none of these is universally available. Accordingly, we built a layer in the form of a reliable message class, which allows processes to communicate directly and reliably with each other. The underlying implementation has been coded in several ways.

There are three classes of `rpm` message: immediate, reply, and mail. All three are ways of sending arbitrarily long data from one process to


```
#include "rmp_test.x"
// as before, this file is created by uimake
main()
{
    rmp_out myrmp(rmp_reply);
    myrmp.sendto("dev", "service1");
    /* names looked up as machine and service */
    myrmp.start();
    /* at this point myrmp is a c++ ostream, logically connected
       to the other side */
    myrmp << "This is a request";
    myrmp.end();
    int i; /* We assume the reply is an integer */
    myrmp.reply() >> i;
    myrmp.reply().end();
    cout << "The reply was " << i << "0";
    return(0);
}
```

Program 2: rmp example program

another. The differences are that immediate messages must be destined for a process on the same machine and are unidirectional, while reply messages allow the receipt of a reply from the destination process. Mail messages may be destined for a process on any other known machine. They are acknowledged by the comms-in-out system described below, and delivery is guaranteed provided that the destination machine knows about the destination process. Undelivered mail mounts up at the destination, and eventually triggers operator intervention. Mail messages also guarantee that they will arrive in order between given source and destination processes.

The message protocol is used extensively in the transaction server. The server waits for a message which may either be a local reply message or a remote mail message. It processes that message, possibly sending other messages as it does so. Finally, if the inbound message was a reply message, the success or failure is reported back. Since it is a serious error for a remote mail message to fail, no reply need be sent back.

As an example of the user interface, a code fragment (Program 2) may be of interest.

Comms-in-out

This feature of the infrastructure allows messages to be sent between processes on different machines, without a permanent connection between those machines. The user interface has already been described, being the mail option to the message protocol. What actually happens is that each message is sent as a local message with reply to a transaction server for the comms-in-out system. This server puts the message into a local data file of outgoing messages. Under user selectable conditions, all messages destined for any particular system are batched up, converted into a file, and transmitted via any suitable file transfer program.

Received messages are put into an inbound message file, and are forwarded to the ultimate destination program by a forwarding daemon. The system includes logic to allow delivered messages to be purged from the originating system, and copes with the file transfer method not preserving the order of files transferred, or losing the occasional file.

The actual program used to transfer the files is called *ftasync*. It was written to allow uucp style operation with a protocol which is optimised for X.25 environments where a PAD is involved at one or both

ends. Top level acknowledgements are minimised, but total reliability is not assumed, and partially complete transfers can be restarted without loss.

Windows

Finally, it is worth mentioning the windowing software we have written. Although it doesn't directly relate to the distributed application, it is the basis of what the user of the system actually sees.

The software provides text based windows on which forms, including scrolling areas, can be defined, and a fairly nice way of associating a scrolling area with a table. The usual ability to define specific actions on key or field events is provided as well. To date, no attempt has been made to provide a graphical front end for code generation.

9. Performance

The performance of a complex software system is always hard to judge, except in the sense that it is or is not good enough given the available hardware. What seems clear so far is that we have achieved our basic objective of gaining acceptable performance without sacrificing good structure. The disk space required to store tables compares extremely favourably with other schemes, and in fact on expected volumes the space required will actually be less than in the former system. Also, the CPU load on the system is fairly low, mainly because knowledge of the data has allowed sensibly defined access paths for data to be implemented explicitly.

On the other side of the coin, the performance of the message passing system is not as great as might be hoped, and in particular, the common problems of excessive context switching might come to the fore as the system is extended. Probably the most serious difficulty is that the applications are extremely memory hungry. This is primarily due to the sheer number of screens coded into certain key programs, and the total number of tables. As a result it does not seem reasonable to run the full system on a machine with less than 8MB of memory, which is a problem because the existing Motorola kit has only 4MB and no room for expansion. On the other hand, the application fits happily onto a 386 PC.

10. Conclusion

Anyone who chooses to implement facilities at the level we did must be prepared to defend themselves from a charge of suffering from the NIH (not invented here) syndrome. It is quite likely that before too long, well written products will be available which would have made our own development unnecessary. However, the total cost of the infrastructure development has amounted to less per machine than a basic proprietary database system license would have been. Furthermore, we now have a well-understood basis from which we can develop and enhance the application, without fear of falling foul of the limitations of a proprietary package. If there is any lesson here, it is that extensibility is more important than a fixed set of features, however rich.

As telecommunications costs fall, there may come a time when commercial organisations no longer need to worry about the costs of remaining permanently connected across international distances. Until

then, there will always be a desire to reduce connectivity without sacrificing functionality.

In this paper, we have shown how it is possible to distribute a suitable application over multiple locations in a way which allows independent operation of each location, but which still allows the system to function as a coherent whole.

Bibliography

Ellis & Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990. ISBN 0-201-51459-1

J. Ullman, *Principles of Database Systems*, second edition, Computer Science Press, 1989. ISBN 0-273-085948

D. Knuth, *The Art of Computer Programming volume 3 – sorting and searching*, Addison-Wesley, 1973. ISBN 0-201-03803-X

R. Rock-Evans, *Analysis within the Systems Development Life-Cycle*, Pergamon Press. ISBN 0-08-034100-4



European Forum for Open Systems

The Secretariat
Owles Hall
Buntingford
Hertfordshire SG9 9PL
United Kingdom

Telephone +44 763 73039
Facsimile +44 763 73255
Email europen@EU.net
Pátria Nyomda