EUUG

European UNIX® systems User Group

Spring '89

# CONFERENCE Proceedings

UNIX: European Challenges

3-7 April 1989

at

Palais des Congrès
Brussels

EUUG

European UNIX® systems User Group

# UNIX: European Challenges

Proceedings of the
Spring 1989 EUUG Conference

April 3-7, 1989
Palais des Congres,
Brussels, Belgium

ISBN 0 9513181 2 8

Further copies of the proceedings may be obtained from:

EUUG Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

These proceedings were typeset in Times Roman, Courier and Helvetica on a Linotronic L300 PostScript phototypesetter driven by a Sun Workstation. PostScript was generated using **refer**, **grap**, **pic**, **psfig**, **tbl**, **eqn** and **troff**. Laserprinters were used for some figures.

# PREFACE

This year, we celebrate UNIX's 20th birthday. The first releases have undergone great transformations, growing in size and functionality into the mature and sophisticated operating systems that are forcing the current breakthrough. Although recent UNIX versions are much larger and more complete than their ancestors in the late sixties, the fundamental mechanisms, invented, defined and implemented by Ritchie and Thompson have survived, unaltered, several generations of hardware families.

The high quality standard of the software, the elegance of the basic mechanisms and the intellectually appealing richness of ideas are still present, and well alive in UNIX.

The users community has drastically changed. Time has passed, that UNIX mainly served the purpose of UNIX developers; the technical experts are still there and actively for sure, but they are joined by "real end-users", software houses, multinationals, international institutions.

Recent history has proven undoubtedly, that UNIX is not dead, to answer the provocative question asked by Theo De Ridder during a previous European UNIX systems User Conference. But its face is rapidly changing, partly still by the contribution of individuals mostly now by large bodies.

In this context and with the present conference's theme in mind, consider following relevant questions:

- Can Europe keep up with the fast pace of this evolution?
- Can we – at least partly – control and direct its technical characteristics?
- Do technical innovators on our continent get the chance to work, to experiment and to elaborate their ideas?
- Are the mechanisms in place to allow migration from prototypes into successful products with a worldwide impact?

The European UNIX systems User Group and affiliated national groups face the challenge to adapt to the new types of users in the current context, also finding young people to carry on the work. All this while maintaining their essential reasons to exist:

- establishing an independent platform for technical aspects of UNIX
- setting up information channels through publications, the computer network, conferences and software distributions
- showing the "faces" of the people behind the programs
- encouraging, giving credit to and even sponsoring worthwhile realizations
- maintaining a few positive traditions.

And all this, without becoming a small club of UNIX freaks, isolated from the real world.

When reviewing the contributions to this conference, I think the potentialities are present: several "hot" technical topics will be explored in original contributions, by authors from different backgrounds.

At least some elements of the answers to the above questions will emanate from this conference.

Marc Nyssen

Medische Informatica
Vrije Universiteit Brussel
Laarbeeklaan 103
B-1090 Brussels
Belgium
*marc@minf.vub.ac.be*

# ACKNOWLEDGEMENTS

| | | |
|---|---|---|
| Sponsored by: | European UNIX systems User Group | |
| Hosted by: | Belgian UNIX systems Users Group | |
| Programme Chair: | Marc Nyssen | Vrije Universiteit, Brussels, Belgium |
| Programme Committee: | Eva Kühn | Technische Universität Wien |
| | Michel Gien | Chorus Systèmes, Paris |
| | Sunil K. Das | City University, London |
| Conference Organisation: | Neil Todd | GID Ltd, London, UK |
| Proceedings Production: | Jan-Simon Pendry | Imperial College, London, UK |
| | Dave Edmondson | Imperial College, London, UK |
| | Stuart McRobert | Imperial College, London, UK |
| | Mark Taylor | Imperial College, London, UK |
| Local committee: | Albert Wambecq | Bell Telephone Mfg. Co. |
| | Bruno Schröder | Matra Datasystème |
| | Danny Buyse | Data General Belgium |
| | Michel Rustanowicz | N.C.R. Belgium |

For a young and small group such as the BUUG, the involvment in the setting up of this conference was both stressing and rewarding; behind the persons named above, many others contributed directly or behind the scenes on a shear voluntary basis.

Special thanks go to Elie Milgrom, our former President; for his wise guidance; all board members contributed in brainstorming sessions, at the expense of their valuable time.

The "local committee" stepped forward and forged loose ideas into effective arrangements. Lobbying was part of their approach, to find sponsors and to make the conference known in the growing commercial UNIX community.

The EUUG Secretariat's constant involvment and concern, materialized in dozens of letters, faxes E-mails and phone calls, the data-flow, essential to all concerned by this conference.

EUnet carried an important load of the conference communications at the very moment major changes to domain addresses took place. We appreciate the many postmasters who did not take a bureaucratic attitude, but remembered the true spirit of UNIX solidarity, in this period of transition.

Our thanks to the authors who were approached by the programme committee or who spontaneously sent in their work, giving us a hard job to select between many excellent contributions. Several papers could not be included due to lack of time in the programme and their authors are encouraged to submit their work for the next EUUG conference, in Vienna.

We are also indebted to the institutions that contributed by giving access to their resources or by direct sponsoring: Vrije Universiteit Brussel, Katholieke Universiteit Leuven, Universitė Catholique de Louvain, Data General, Unisys, Matra Datasystème, AT Computing, Digital Equipment, N.C.R., Unisys Belgium.

Marc Nyssen

We would like to thank the Foundations Group in the Department of Computing at Imperial College for supplying the resources used during the production of these proceedings.

— jsp, dme, sm.

# UNIX Conferences in Europe 1977–1989

## UKUUG/NLUUG meetings

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

## EUUG Meetings

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

# Technical Programme

## Graphics

## UNIX "Standards" I

# Author Index

# UNIX and Load Balancing: a Survey

*C. Jacqmot and E. Milgrom*

Unite d'Informatique
Universite Catholique de Louvain
Place Sainte-Barbe, 2
B-1348 Louvain-la-Neuve
Belgium
*cj@lln-cs.uucp*

*W. Joosen and Y. Berbers*

Departement Computerwetenschappen
Katholieke Universiteit te Leuven
Celestijnenlaan, 200 C
B-3030 Heverlee-Leuven
Belgium
*wouter@kulcs.uucp*

*ABSTRACT*

The paper presents a survey of load balancing schemes and their application or applicability to distributed UNIX systems. It aims to provide a guided tour through some important concepts and ideas in this area, together with examples of complete or partial implementations. Relevance to UNIX is stressed, even though some of the systems under consideration have little relation to UNIX proper.

## Introduction

A true general-purpose *distributed system* appears to its users as a single, monolythic system, even though it is built out of discrete components (machines and software) interconnected by means of some kind of network. The physical distribution of the components becomes invisible because the software ensures full transparency. As pointed out in [Tane 85], ... *the user views the system as a "virtual uniprocessor", not as a collection of distinct machines.*

A *distributed operating system (d.o.s)* is the operating system component of a distributed system: it consists of various local software components which cooperate to achieve the global functionality. A *d.o.s* therefore attempts to manage, in a transparent way, a pool of resources physically distributed on a set of interconnected machines.

A *network operating system (n.o.s.)* consists of a number of systems which could operate independently of each other, but which are set up to cooperate to achieve some measure of global behaviour, with a much lower degree of transparency than in the case of a true *d.o.s.*

The difference between a *d.o.s.* and a *n.o.s.* thus lies in the degree of logical integration as perceived by the users. The hardware architecture which underlies both *n.o.s.* and *d.o.s.* belongs to one of several models [Tane 85] [Coulo 88].

- The *minicomputer model*: a number of standard multi-user systems are connected by means of a network; a user logs in on a specific machine and may have facilities for remote logins and remote executions; some systems even provide ways to achieve transparent access to remote files (e.g. RFS, NFS). Such an architecture is found mostly in *n.o.s.*, where a rather low level of transparency is reached; it can also be found in *d.o.s.* with a much higher degree of transparency.

- The *workstation/server model*: a variant of the minicomputer model, in which some of the machines are personal workstations, possibly without local disks. Servers are needed to provide access to resources which are not available locally.
- The *processor pool model*: a pool of generally identical CPUs, out of which some are temporarily allocated to users in function of their processing needs. The pool is managed transparently by a global server. A user is not logically associated in a permanent way with any of the CPUs.
- The *hybrid model*: a user is logged in on a given machine (mini computer or workstation) on which its programs are executed; a pool of CPUs is available to allocate additional computing power for specific tasks.
- The *general model*: a heterogeneous collection of machines (single and multi-user) and CPUs; the notion of logging in on a given machine has disappeared.

Both in *n.o.s.* and in *d.o.s.*, but mostly in the latter, one of the possible goals is to avoid overloading some machines, thus degrading the performance for users whose jobs are executed on these machines. Techniques used to avoid such local congestions belong to the class of *load sharing* or to that of *load balancing* schemes. Additional goals of these techniques are: executing parts of the same job on different CPUs to increase the degree of parallelism, minimize the global interactive response times to optimize overall system throughput.

We shall use the term *load distribution* to designate either of both approaches, which will be defined more precisely below.

It should be noted that another goal sometimes conflicts with the ones mentioned above: one might want to cluster subsets of cooperating processes on the same machine, to limit inter-processor communication costs.

The purpose of this paper is to present a number of criteria for comparing various load sharing and load balancing schemes. This will be achieved by refining and adding to the taxonomy proposed by Casavant and Kuhl for scheduling principles in general purpose distributed systems [Casa 88].

A number of systems are analyzed in the light of the proposed criteria. We have limited ourselves to systems which appear to have reached an advanced stage of implementation and about which enough information on their load distribution aspects is publicly available. We shall both address systems which are presented as being UNIX-based and systems which appear to have no direct relation to UNIX.

Finally, we shall discuss the relationship between load sharing and load balancing concepts and the "UNIX philosophy", i.e. how these concepts can be applied to UNIX-based distributed systems.

## Basic concepts

A number of previous studies have resulted in published surveys in the area of distributed operating systems [Casa 88], [Smit 88], [Tane 85], [Zhou 87], [Coulo 88], etc. In many cases, these surveys are either very general, or they fail to address those aspects we wish to present here in any depth.

Our goal is to present criteria which will allow one to effectively compare systems with respect to load sharing and load balancing capabilities.

Our starting point will be the taxonomy proposed by Casavant and Kuhl for scheduling principles in general purpose distributed systems [Casa 88], which presents a two-level taxonomy: a hierarchy for some criteria, and a flat list for other ones. In our opinion, the hierarchical part is not very convincing: little justification is given for the tree structure, which appears to be both incomplete and redundant. Indeed, some useful branches of the tree are not shown, while some subtrees appear twice.

We feel, therefore, that a "flat" list is more appropriate than one that appears to be artificially structured, even though some criteria obviously depend on other ones (e.g. *cooperative* strategies appear only in systems with *distributed* scheduling authorities, see below).

We found that some of the criteria of Casavant and Kuhl are not fine enough to allow us to describe the various aspects of interest in load distribution schemes. On the other hand, since their goals were different from ours, it is not surprising that not all their criteria are relevant here. We should recall that their taxonomy is about *process scheduling* in distributed systems, while our interest lies in processor assignment within load distribution schemes. Thus, some of their criteria are not significant for discriminating among the systems we wish to study. We shall be able to make effective use of five of their criteria (out of twelve). Nevertheless, three of the criteria which are non discriminating are useful for general background information: we present them briefly below as the first three in the list.
In order to compare systems from the point of view of load distribution, we have defined eight new criteria

to complete the taxonomy.

*Criteria from Casavant and Kuhl*

- *Local vs. global scheduling*: Casavant and Kuhl distinguish between assignment of a time-slot of a processor on a single system (*local*) and assignment of a processor on a system based on many processors (*global*), where local scheduling is taken care of by some local operating system function. In our case, the systems we wish to analyze are all based on multiple CPUs: they belong to the latter class.
- *Static vs. dynamic scheduling*: this has to do with the *time* at which scheduling and CPU assignment decisions are made. In *static* scheduling, it is assumed that all decisions can be made at program linking time: an executable program is always associated with the same processor. In *dynamic* scheduling, processor assignment is based on execution-time information: this is the case for all our systems.
- *Optimal vs. suboptimal scheduling*: *optimal* scheduling is based on the availability of complete information; whenever a scheduling decision is made, all possible solutions are compared to allow choosing the best one. In *suboptimal* techniques, decisions may be based on partial knowledge of global system state: this is again the case in all systems we analyzed, because of the cost involved in trying to compute complete information is much too expensive.

To summarize, all distributed systems we have studied and which exhibit load distribution features are *global, dynamic* and *suboptimal* as defined in [Casa 88].

- *Distributed vs. nondistributed scheduling authority*: in systems with *nondistributed scheduling authority*, the global strategy is implemented on a single processor. In systems with *distributed scheduling authority*, the global strategy is implemented by means of partial decisions within the scheduling algorithms of some or all of the various machines of the whole system.
- *Cooperative vs. noncooperative scheduling*: for systems with *distributed scheduling authority*, Casavant and Kuhl distinguish between those which are *cooperative* and those which are not.
  We feel that the given definition is not fully operational, since two different concepts appear to be mixed together: firstly, whether the individual entities responsible for implementing part of the global scheduling indeed operate towards a common goal and, secondly, whether they take into account the effect of their decisions on the other components of the whole system. The two items are not necessarily linked together in all distributed systems. In systems with load distribution features, the two items are inherently linked together: systems with *distributed scheduling authority* are necessarily *cooperative*; systems with *nondistributed scheduling authority* are *noncooperative*.
- *Approximate vs. heuristic scheduling*: in *approximate scheduling*, the decisions are made on the basis of an estimation of the impact of the decision on the quantity one tries to optimize (e.g. load, communication overhead, response time, etc.). This assumes the existence of a *metric*.
  In *heuristic scheduling*, the effect of decisions on the quantity one tries to optimize cannot be computed directly; instead, one relies on assumptions about the *indirect* effect of actions on various parameters on system performance.
- *Adaptive vs. nonadaptive scheduling*: in the former, the parameters of the scheduling algorithms vary in function of the global system state; in the latter, they remain constant.
- *One-time assignment vs. dynamic reassignment*: in some systems, once a process is allocated a CPU, it remains in execution on the same CPU until its normal or abnormal completion (*one-time assignment*); in other systems, a process may be moved from one CPU to other ones in its life time (*dynamic reassignment*).
  For our purposes, we shall refine the latter criterion into two subclasses: *back and forth assignment*, where a process initiated on CPU **A** may be executed on CPU **B** until it must be shipped back to CPU **A** because of new requirements for CPU **B**, and *multiple assignment*, where a process may be moved any number of times between any number of CPUs.

*Definitions*

Before proceeding with additional criteria for comparing load distribution in *d.o.s.*, we must first define a number of additional concepts in the area of *load distribution*.

*Processor assignment* is that part of the global scheduling policy which is concerned with the allocation of CPU resources to processes. In centralized systems, a process is always assigned the single available processor; in distributed systems, whenever a new process is started, there may be more than one processor which could be allocated to the new process; furthermore, during the life time of a process, one might contemplate changing this assignment at various points in time.

The execution of a process requires the availability of code and data, but also the existence of an *environment*, set up by the operating system.

*Remote execution* is the technique whereby a process is created on a remote machine by having the environment created by the operating system of the remote machine and the code and data loaded from file.

*Process migration* is the technique whereby the environment of a process is *transferred* to a remote machine or whereby enough information is transferred to allow the operating system of the remote machine to reconstruct an adequate environment. The execution of the process then proceeds on this remote machine. The code and data may be either transferred or possibly loaded from file.

The *source processor* is the CPU on which the initiative is taken to start the execution of a new process or the CPU from which a process will migrate to another one. The *target processor* is the CPU on which a process will be executed next. When the source and the target are identical, we shall speak of *local processor assignment*; when they **may** be either identical or different, we speak of *remote processor assignment.*

We should stress here that our definitions of remote execution and process migration are different from the ones sometimes used in the literature. Our emphasis lies on the matter of what happens to the environment of a process. Others consider that the first processor assignment of a new process to a remote CPU is always to be considered as remote execution. In our definition, this is not the case, since we shall speak of remote execution only if no environment is propagated to the target machine. For example, in our terminology, the Berkeley *"rsh"* command is true remote execution, while the SunOS *"on"* command is process migration.

Some systems may thus combine remote execution for the first remote processor assignment with process migration for the next ones; others use either technique on an exclusive basis.

We consider that there are two main families of remote processor assignment strategies. The common goal is always to achieve some measure of load distribution among the available processors, i.e. avoiding imbalance in the individual processor loads, since this is expected to improve global performance. One should be aware of the fact that there exists some confusion in the literature about the terminology in this area; we have therefore chosen to distinguish between the strategies by using the following names:

- *Load sharing* allocates processes to CPUs which are considered to be *idle*, according to some definition of the concept of idleness. Idleness is sometimes understood to mean truly without any active user processes; in other cases, idle means: without any "guest" user processes. In other cases still, idle might mean: without interactive user activity. In some cases, a process may migrate from one processor to another one, because its host CPU ceases to be idle according to the working definition.
  The strategy is concerned with deciding which processor among several possible candidates should be assigned to a single process.
- "*Load balancing* attempts to balance the load on all processors in such a way as to allow progress by all processes on all nodes to proceed at approximately the same rate" [Casa 88]. The notion of *continuously* trying to *balance* the loads is central here. To reach the global objective (or to approximate it), it might be necessary to review the processor assignments of many processes at several points in time.
  Here, one doesn't look for idle processors, but for processors which are less loaded than other ones.

It should be noted that both *remote execution* and *process migration* may be used in the frame of load balancing or load sharing schemes. They may also be used in order to achieve completely different purposes: remote execution is often used because of the local unavailability of some resource(s); process migration may be invoked to move processes in order to share resources efficiently or to achieve a higher degree of fault tolerance.

*Load distribution criteria*

- *Load sharing vs. load balancing*: as explained above, the major difference between these two approaches lies in the fact that *load balancing* attempts to **maintain** some kind of balance between the loads on the various processors, thereby implying many processor assignment decisions for a given process, whereas *load sharing* only tries to avoid processor idleness.
- *Remote execution vs. process migration*: the distinction here resides in the fact that the environment for a process is either created on a remote machine (*remote execution*) or transferred or reconstructed on the remote processor (*process migration*).
- *Degree of transparency*: there are several classes of schemes, depending on where the responsibility lies for selecting candidate processes for load distribution: on some systems, the *user* specifies whether a command should result in a process that is a candidate; on other systems, this may be indicated *within processes* themselves; some systems achieve a higher degree of transparency by having this decision made automatically (in the command interpreter or within the operating system).
- *Nature of parameters for choosing the target*: the algorithms for choosing a target CPU for a process which is a candidate for load distribution are based on a variety of possible parameters which determine load indices.
  Load sharing systems look for an "idle" CPU, where "idle" may mean, for instance, a CPU without an interactive user, or a CPU which hasn't yet been assigned a remote process, and so on.
  Some systems take the load of possible target CPUs into account.
  Other systems base their decisions on a more general measurement of the usage of resources (CPU, I/O, communications, etc.).
  In some cases, the decision is based not only on measurements of the load on possible target CPUs, but also on some *global* measure of the overall system load.
- *Degree of distribution of parameters for choosing the target*: for systems in which these parameters are memorized in other machines than only on the candidate target CPUs, one distinguishes between systems with a single centralized server which is the source for the parameters for all target assignment decisions, systems with groups of such servers, and systems in which local copies of parameter values are kept in all source machines.
  In other systems, the target assignment parameters are only kept on the candidate target machines: source systems must broadcast requests to obtain current values of the parameters.
- *Global vs. partial knowledge of the state of the system*: in some systems, target assignment decisions are based on full knowledge of the states of all possible target CPUs; other systems rely on a subset of the global system state; this subset may vary in time. Note that global knowledge doesn't necessarily imply optimal scheduling.
- *Time frame of load distribution decisions*: this refers to the moments, in a process life cycle, when load distribution decisions are made (refinement of the *dynamic scheduling* criterion). Some systems make these decisions only at process creation time *(initial assignment)*; others react whenever some load imbalance is detected; others still may react on internal criteria (e.g. process table becoming full).
- *Static vs. dynamic selection of candidate processes*: static selection happens either when processes subject to load distribution are chosen by the user himself or because they appear in a table (e.g. a table of CPU-bound commands); dynamic selection implies some evaluation of resource consumption either at initial assignment or during execution. One could also imagine a combination of both approaches, even though none of the systems we studied attempts to do this.
  It should be noted that most authors indicate what the characteristics are of processes which are appropriate candidates for load distribution, but few explain clearly *how* these processes are identified in practice.

## Overview: selected systems

In this section, we present the major features of the various systems for which enough information about load distribution aspects is publicly available, in alphabetical order. A table summarizing the application of the load distribution criteria to these systems appears in appendix.

**Amoeba** [Tane 81], [Tane 85], [Tane 86], [Rene 86]

Amoeba is an "Object Oriented Distributed Operating System" developed at the Department of Mathematics and Computer Sciences of the Vrije Universiteit of Amsterdam under the direction of A.S. Tanenbaum; it is operational since 1984.

The hardware architecture belongs to the *hybrid model* and consists of personal workstations, a fixed pool of processors (CPUs) dynamically allocated and released according to users' needs, and specialized servers (file servers, database servers, ...) connected by a fast local area network.

Important issues are: communication and protection (achieved through the use of capabilities).

The Amoeba system has a UNIX-emulation library: UNIX systems calls are converted in message transactions with UNIX supporting services.

*Processor assignment*

A centralized administrator (*nondistributed authority*) manages the pool of processors; it has *global* knowledge of the state of *each* processor (free or not).

The *load sharing* strategy comes into play whenever a *process is started up*: the system on the *source processor* contacts the administrator, which tries to locate a *free CPU* in the processor pool. The algorithm is thus neither *adaptive* nor *cooperative*. Local processor assignment should be used for highly interactive processes, while the target processors for batchlike jobs might be remote (*dynamic selection of candidate processes*).

Processes are *one-time* assigned on the target processor using a *process migration* technique. These assignments have an effect on system performance which is not directly measurable: no metric is available (*heuristic scheduling algorithm*).

**Butler** [Dann 82], [Nich 87]

The Butler system was developed as part of a PhD thesis at the Computer Science Department of Carnegie-Mellon University in the early eighties. Dannenberg's Butler system consists of a set of programs running on stations which must provide facilities similar to those offered by the Andrew Computing Environment [Morr 86] (a shared file system such as Sun's Network File System and a network window manager such as MIT's X Windows).

At CMU, the environment is composed of a large number of single-user workstations running under standard Berkeley 4.2BSD UNIX; it is thus built according to the *workstation/server model.*

Butler's major goal is to make idle workstations available to users who might benefit from additional computing power.

*Processor assignment*

Butler's goal is a clear example of a system with a *load sharing* strategy.

In the first version, a list of *all* available workstations was maintained in a *centralized* way. In order to avoid a possible bottleneck, recent versions use a *number* of servers, which each maintain a list of idle machines. Because of the transmission delay, this list is sometimes inaccurate: it therefore leads to a *partial knowledge* of the state of the pool.

*Users* who wish to take advantage of the Butler system must prefix their commands with "rem" (*static selection of candidate* processes). The *local* "rem" program (*distributed scheduling authority*) consults one of the servers which owns a list of free workstations in order to find a *free CPU*; it then contacts the Butler program running on the target processor to receive a confirmation of the idleness of the machine. An environment is then transmitted to the target node and, afterwards, the execution of the *new process* may start on that machine. According to our definition, this is a case of *process migration* (initial assignment), even though the authors claim to use remote execution. Load distribution occurs thus at the *shell* level.

Whenever a user reclaims a workstation, "guest" processes are killed. The assignment is thus typically a *one-time assignment.*

As was the case in Amoeba, the scheduling algorithm is *heuristic* and *nonadaptive*.

**Charlotte** [Arts 84], [Arts 86a], [Arts86b], [Berb 87]

Charlotte is a distributed operating system developed at the Computer Sciences Department of the University of Wisconsin-Madison, by Y. Artsy, H-Y. Chang and R. Finkel.

This *d.o.s.* is designed for the loosely-coupled Crystal multicomputer; it has several goals: to explore operating system design for multi-process applications, to study communication paths and primitives, to maximize utilization of computational resources while minimizing overhead, and to serve as a testbed for distributed algorithm design.

Process migration was implemented in mid-1985.

*Processor assignment*

*Process migration* in Charlotte is used to achieve *load distribution* in a *completely transparent* way.

One of the main goals is to provide a testbed for the study of load distribution policies. The scheme allows *dynamic reassignment* of processes, which are *selected dynamically* as candidates for migration.

Machines are grouped into clusters whose load information is maintained by a single manager process per cluster.

A large variety of strategies can be implemented; some of these are based on the one-per-cluster managers. For instance, the authors have successfully realized the load exchange algorithm used in MOS.

In order to support several policies, each kernel on each node of the system collects a large set of statistics on process and machine activities:

- machine load: the number of processes and communication channels to other processes, the average CPU load and the average network load;
- process resource usage: average and total CPU and network utilization of each process and the number of packets send to local and remote processes;
- communication channel statistics: the total number of packets sent and received on the most active communication channels.

The managers also play the role of the scheduling authority (*distributed authority*). It should be noted that, in view of the flexible nature of the testbed approach, none of the other criteria are frozen: it is possible to implement many different strategies on top of the basic mechanisms (see appendix).

**MOS** [Bara85a], [Bara 85b], [Bara 86a], [Bara 86b], [Berb 87]

The MOS Multicomputer Operating System, developed at the Department of Computer Science of the Hebrew University of Jerusalem, is a distributed system which truly emulates UNIX (the development started from UNIX Version 7); it makes a cluster of loosely connected independent homogeneous computers appear as a single UNIX system: it is therefore based on the *minicomputer model.* It has been operational since 1984.

Main properties are: network transparency, decentralized control, site autonomy and dynamic process migration. An additional goal of the design is to reduce the complexity of the whole system while preserving a high performance.

*Processor assignment*

MOS provides *dynamic load balancing* through *process migration.*

A local manager process is responsible for gathering load information (*local server*) and for marking candidate processes for migration (*distributed scheduling authority*).

A marked process will determine, the next time it enters a system call, whether it really is useful for it to migrate by computing cost/benefit ratios for possible target systems. This may result in the selection of a target machine and migration of the marked process (*dynamic reassignment*).

Another migration situation occurs when a local process table is full and when a local process *forks* to create a child: the child must then necessarily be moved to another machine.

The nature of processes which are typically good candidates for migration is linked to UNIX characteristics: CPU bound processes, process which forks frequently, processes who are not about to terminate, processes with small amount of data. This is a typical case of *dynamic selection.*

The scheduling authority is *decentralized*: each site assumes responsibility to *approximately* balance the global load. The scheduling algorithm is *nonadaptive*.

The load information is recorded in a *decentralized* way; load data is collected by means of a load exchange algorithm that we shall now briefly describe.

Each system stores load information about a fixed number of machines (*partial view*) and transmits the most recent half part of that information periodically to another system which is selected randomly. Its own load always belongs to the more recent information part; this is the only up-to-date load data known by the local system. Data used on each node to compute the local load consists of the local CPU load and the local number of processes.
On the destination node, the contents of the message replaces the older information.

**NEST** [Agra 85], [Ezza 86]

NEST (NEtwork workSTations) is a project undertaken at the Computing Systems Technology Research Department of AT&T Bell Laboratories. The environment consists of a network of autonomous cooperating personal computer workstations.

The architectural model adopted in NEST is the *hybrid model*. Processors are divided into two classes: a pool of *compute servers* may be used by the workstations to supplement their computational needs. Some processors are permanently marked to be compute servers, while any workstation may temporarily join the shared pool of compute servers by advertising its availability.

Processor sharing is an important aspect of the project: NEST developed an augmented UNIX system with an execution capability which allows processes to be offloaded to compute servers.

*Processor assignment*

In NEST, load distribution is not completely transparent at the *user level*: a user may indicate that he wants to offload his programs to a compute server by using the "rexec" prefix when he submits a command. This indicates that the resulting processes are candidates for *migration*. A similar facility may be used in *C programs*. The authors presents this mechanism as a remote execution mechanism. We prefer to use the term "process migration", since a not insignificant chunk of information is transmitted to the target node in order to build the remote process environment.

The *system* may also decide to *migrate* a newly created process whenever its process table is full. (*dynamic selection*) This is a *one-time assignment* scheme, but the authors expect to realize a dynamic reassignment scheme soon.

Each *source system* keeps information about the various processors which belong to the pool of compute servers and, of course, about itself (*global view*). This information is used to compute a Normalized Response Time (which is another name for the load) and which consists of the average number of processes (user processes, systems processes, network server processes and so on) and the average number of processes during the busy period over the most recent window in time. In other words, the CPU utilization is used as load index.

The aim of processor assignment is to minimize the Normalized Response Time. We shall now describe the algorithm which is used to achieve this goal and which, in fact, tries to *balance* (*approximate scheduling*) the load among the set formed by the source processor and the compute servers. NEST is the only system in our list which uses an *adaptive* algorithm.

For each machine in the set {pool − source}, an assignment processor cost is derived from the corresponding NRT and an average cost is deduced for the set. If the local assignment cost is less than or equal to the average value, the source processor is allocated. Otherwise, a multiple of the local assignment cost, called "bias against process movement" is estimated. The target processor will be different from the source processor if the sum of the remote assignment cost and the bias remains lower than the local assignment cost. A list of predefined factors used in the computation of the bias is available to the algorithm, which uses a greater bias if the current load difference is important in the set and a lower one if the system is nearly balanced. Assignment processor decisions are made on each source node (*distributed authority*).

## **Sprite** [Doug 87], [Oust 88]

Sprite is an experimental *d.o.s.* under development at the Computer Science Division of the University of California at Berkeley. It is part of the SPUR project, whose goal is the design and construction of a high performance multiprocessor workstation with special hardware support for Lisp applications. One of Sprite's primary goals is to support applications running on SPUR workstations; it should facilitate the development of multiprocessor applications and take advantage of the multiple processors in providing system services.

Sprite is *currently* used on Sun-2 and Sun-3 workstations (*workstation/server model*). It implements a superset of the UNIX 4.3BSD interface.

*Processor assignment*

Little information is publicly available in the literature about process scheduling in Sprite: process migration is the only mechanism currently implemented. Modules which will decide which processes must or may migrate and where they have to go are not yet ready.

Nevertheless, *process migration* was originally designed in order to allow the use of *idle stations*: it will therefore be used in the framework of a *load sharing* scheme. The scheduling algorithm is *heursitic* for the same reasons as was the case for Amoeba and for Butler.

Processor assignment is not necessarily transparent from the *user* point of view: manual migration will be permitted through the use of shell commands. On the other hand, a new version of the "make" utility uses process migration to offload compilations to idle stations. Sprite allows guest processes to be migrated away when a user reclaims his own workstation. This is a case of *back and forth assignment.*

[Doug 87] gives additional criteria according to which processes should be migrated: when there are two or more CPU-intensive processes on the same node, when a process is detected whose life expectancy is "long", ...

## **V** [Cher 83], [Cher 84], [Cher 85], [Thei 85], [Cher 88], [Berb 87]

The V Distributed System was developed by D. Cheriton and his team at the Computer Science Department of Stanford University. It has been running since September 1982.

V is used in an environment of diskless workstations and file servers (*workstation/server model*).

As V is an experimental system for research in distributed systems protocols, special attention was paid to design general purpose protocols. A lot of attention was also paid to performance. The various service protocols define uniform interfaces and access to services. Any machine supporting those protocols can be included in a V distributed system [Cher 84]: the group's UNIX systems run a server program which implements the V protocols, in order to provide access to UNIX files and services.

V offers a number of UNIX related facilities: the V file server implements a UNIX-like file system; various run-time libraries implement conventional language or application-to-operating system interfaces such as C standard I/O or Pascal I/O, a pipe server implements UNIX-like pipes, etc...

*Processor assignment*

V belongs to the family of *load sharing* systems. It provides both true *remote execution* (there is no transfer of information about the environment) and true *process migration* (with transfer of environments) in order to allow the sharing of *idle workstations.*

Remote execution is only used for initial assignments.

Processor assignment is not totally transparent for the *user*, who may decide at the *command level* to allow the process generated by his command to be executed on a remote processor. The same decision may be made inside a *program* and, finally, the migration decision may be taken at the *system level.*

The V system is the only system among the distributed systems presented here which does *not* use a cache mechanism to record load information about the remote processors. Whenever a process must be allocated a new CPU, the source node *broadcasts* a request and receives, in return, an answer from every idle machine. The return message which is received first is considered to originate from the lesser loaded workstation.

A workstation is reclaimed when a user issues a command that requires the removal from the workstation of a specified program or of all guest programs. No limits are put on the number of successive migrations of a given process (*dynamic reassignment*).

**F&Z** [Ferr 87]

D. Ferrari and S. Zhou have developed a load balancer within the Computer Science Division (EECS) of the University of California at Berkeley.

This load balancer may be used on components of a distributed system built on top of a *minicomputer model*: the system is based on a modified C-shell for Berkeley 4.3BSD UNIX.

*Processor assignment*

The work of Ferrari and Zhou consists in an experimental environment that may be used as testbed for studies of load indices. The *load balancer* is transparent for the user, even though it operates at the *C-shell level*. A table which belongs to the user's context contains a list of commands which will generate processes eligible for *remote execution*: this is an instance (the only one in our overview) of a *static* technique for selection of candidate processes. When a user submits such a command, the *local* C-shell acts as a processor assignment manager: it contacts a local *load information manager* in order to select a target processor for that job and to *approximately* balance the load.

Load information is collected periodically by a *central node* which propagates the new *global* load information to all nodes. The load metric is based on the average value of the *main resources queue length*; it uses coefficients which reflect the relative importance of the resources. According to the authors, this type of load indices performs better than load indices based on resource utilization.

When remote processor assignment occurs, the new process is *remotely executed* on the target node and remains on that CPU for the rest of its life time. This is the only case in our study where processor assignment is implemented solely by means of true remote execution. The absence of process migration limits the scope of the mechanism.

## Load distribution and UNIX

Studies of actual UNIX systems [Cabr 86], [Lela 86], [Bara 86a], [Casta 86] show that processes tend to exhibit common characteristics. Some of these characteristics may have a direct bearing on load distribution strategies for UNIX systems.
We shall review here a number of behavioral patterns of UNIX processes and we shall indicate the consequences on load distribution.

- *A majority of UNIX processes have a very short life time.* [Cabr 86] indicates an average life time of 0.4 sec; no more than 8 % of all processes have a life time longer than 8 sec.
  We see thus that it is probably not very useful to assign short-lived processes to remote CPUs, since the load balancing overhead may cancel the possible benefits. This line of reasoning is followed in MOS, in Sprite and in F&Z; it would be possible to do so in Charlotte too, thanks to the available information.
- *A few processes consume the most important part of CPU resources*: the 0.1 % largest processes account for 50 % of all CPU used [Lela 86]
  It is thus also not very useful to apply load distribution strategies to processes which consume little CPU or, more generally, which consume few resources: the overhead may again cancel possible benefits.

The two points mentioned above lead to the question of determining the future behaviour of a process (life span, usage of CPU and other resources).

- *It is usually difficult to predict future behaviour of a process.* In particular, since it is usually impossible to predict future behaviour *at process creation time*, systems which attempt to apply load distribution strategies whenever a new process is created can be assumed to waste a lot of effort with little achievement! On the other hand, if one waits until a process is running to extrapolate future behaviour from the past, one is compelled to use process *migration*, thereby inducing the higher cost of moving the whole process environment.
  It should be noted that none of the systems we studied attempts to distinguish between *classes* of processes (e.g. CPU-bound, I/O bound, communication-bound) with respect to the computation of load factors: the same formula is used for all types of processes.
- *In UNIX, it is often possible to identify a large class of standard programs (commands) with predictable behaviour.*
  Instead of looking at processes in an abstract way, one can take advantage of the knowledge that one has about specific programs in order to choose an adequate target CPU. Programs such as *date* are so short lived that they should probably always be executed locally. Editors such as *vi* may require a lot

of user interaction, which may also indicate a preference for local execution. CPU-bound programs such as *troff* are good candidates for remote execution.

MOS, Sprite, F&Z all try, in their own way, to take advantage of these typical UNIX characteristics when selecting possible candidate processes.

- *The command line contains information that is useful for characterizing some aspects of process behaviour.*
  There is another exception to the rule which states that future behaviour is hard to predict: at the command interpreter level, some information is readily available about access to (special) files and use of pipes. This may be used to influence the choice of a suitable target machine.
- *Processes are created in short bursts.* For instance, [Cabr 86] states that in 94 % of observed one-second intervals, at most 1 process was created.
  In those systems where load distribution occurs when new processes are created, it is thus wasteful to gather load information by means of cyclic algorithms: broadcast techniques are more efficient.

In our opinion, the systems we have studied explore but part of the spectrum of possible combinations of the various techniques mentioned earlier. For instance, no system combines a priori knowledge about "standard" programs and dynamic evaluation of candidates for load balancing. This shows that many more variants are possible and should be examined !

## Conclusions

We have presented a number of criteria to characterize load distribution mechanisms in distributed systems. We have then used these criteria to discuss various systems described in the literature; it should be noted that this presentation doesn't aim at *comparing* these systems from the point of view of their respective performances, but only from that of their load distribution approaches.

The systems we studied belong to three classes:

- Distributed systems built on top of non modified (or weakly modified) UNIX kernels: Butler, NEST, F&Z.
- Distributed systems offering a UNIX-like interface: Amoeba, MOS, Sprite.
- Distributed systems without direct relation to UNIX: Charlotte, V.

We note that the documentation about many of the systems is much more explicit about the selection of target machines (how, when) than about the selection of potential candidate processes. Could this indicate that their authors have built a mechanism which they believe to be useful, but whose effective application is not always obvious ? Indeed, one could also argue that load distribution is useless both on heavily loaded systems (because of added overhead) and on lightly loaded systems (because of the low gains to be expected).

It is not always obvious how one should precisely evaluate the cost/benefit ratio of load distribution: the expected improvements should be balanced against increased overhead (CPU, communications, etc.) and possibly costly residual dependencies (e.g. how to propagate *signals* after a process has moved several times).

We also note that, even though improved response times are frequently mentioned as primary motivations for load balancing, none of the systems attempts to measure response times to influence the load distribution strategies: the metrics are always based either on resource utilization or on queue lengths.

A number of problems have not yet been addressed here (but are currently being studied), e.g. finding out to what degree the load distribution mechanisms are related to the three classes mentioned above; techniques to avoid *thrashing* because of processes chasing lightly loaded target machines, issues related to protection and security, ...

Further study should also be devoted to issues more specifically related to *process migration* itself: even though some of our general criteria may be reused in this context (for instance: one-time assignment vs. dynamic reassignment, static vs. dynamic selection of candidate processes, time frame for process migration decisions, etc.), a number of other issues must be addressed and are currently being studied (e.g. how to actually move the environment of a process, how many processes are moved together, etc.).

## References

[Agra 85] R. Agrawal, A. K. Ezzat, "Processor Sharing in NEST: a Network of Computer Workstations", *1st International Conference on Computer Workstations*, November 1985, pp. 198-208.

[Arts 84] Y. Artsy, H. Chang, R. Finkel, "Charlotte: Design and Implementation of a Distributed Kernel", Computer Sciences Technical Report 554, Computer Sciences Department, University of Wisconsin-Madison, August 1984.

[Arts 86a] Y. Artsy, H. Chang, R. Finkel, "Process Migrate in Charlotte", Computer Sciences Technical Report 655, Computer Sciences Department, University of Wisconsin-Madison, August 1986.

[Arts 86b] Y. Artsy, H. Chang, R. Finkel, "Interprocess Communication in Charlotte", Computer Sciences Technical Report 632, Computer Sciences Department, University of Wisconsin-Madison, February 1986.

[Bara 85a] A. Barak, A. Shiloh, "A Distributed Load-balancing Policy for a Multicomputer", *Software-Pratice and Experience*, **15**(9), September 1985, pp. 901-913.

[Bara 85b] A. Barak, A. Litman, "MOS: A Multicomputer Distributed Operating system", *Software-Pratice and Experience*, **15**(8), August 1985, pp. 725-737.

[Bara 86a] A. Barak, O. G. Paradise, "MOS - A load-Balancing UNIX", *Proceedings of the EUUG Autumn 1986 Conference*, September 1986, pp. 273-280.

[Bara 86b] A. Barak, O. G. Paradise, "MOS - A load-Balancing UNIX", *Proceedings, Usenix Technical Conference - Summer 1986*, 1986, pp. 414-418.

[Berb 87] Y. Berbers, "Design of the HERMIX Distributed Operating System", PhD Th., Departement Computerwetenschappen, K.U. Leuven (Belgium), December 1987.

[Cabr 86] L. Cabrera, "The Influence of Workload on Load Balancing Strategies", *Proceedings, Usenix Technical Conference - Summer 1986*, 1986, pp. 446-458.

[Casa 88] T. L. Casavant, J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", *IEEE Transactions on Software Engineering*, **14**(2), February 1988, pp. 141-153.

[Cast 86] C. Castagnoli, "Load Balancing Computational Servers in a UNIX Environment", *Proceedings of the EUUG Autumn 1986 Conference*, September 1986, pp. 267-272.

[Cher 83] D. R. Cheriton, W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", *ACM Operating Systems Review: 9th ACM Symposium on O.S. Principles*, **17**(5), October 1983, pp. 129-140.

[Cher 84] D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, **1**(2), April 1984, pp. 19-42.

[Cher 85] D. R. Cheriton, W. Zwaenepoel, "Distributed Process Groups in the V Kernel", *ACM Transactions on Computers Systems*, **3**(2), May 1985, pp. 77-107.

[Cher 88] D. R. Cheriton, "The V Distributed System" *Communications of the ACM*, **31**(3), March 1988, pp. 314-333.

[Coul 88] G. F. Coulouris, J. Dollimore, "Distributed Systems-Concepts and Design", Addison-Wesley Publishing Company, Inc., 1988.

[Dann 82] R. B. Dannenberg, "Resource Sharing in a Network of Personal Computer", PhD Th., Carnegie-Mellon University, December 1982.

[Doug 87] F. Douglis, "Process Migration in the Sprite Operating System", Report No. UCB/CSD 87/343, Computer Science Division, University of California at Berkeley, February 1987.

[Ezza 86] A. K. Ezzat, "Load Balancing in NEST: a Network of Computer Workstations", *Proceedings AFIPS Fall Joint Computer Conference*, November 1986, pp. 1138-1148.

[Ferr 87] D. Ferrari, S. Zhou, "An Empirical Investigation of Load Indices for Load Balancing Applications", Report No. UCB/CSD 87/353, Computer Science Division, University of California at Berkeley, May 1987.

[Joos 88] W. Joosen, P. Verbaeten, "Load Sharing: Current Research in Distributed Systems", Report CW 80, Department of Computer Science, Kathollieke Universiteit Leuven, May 1988.

[Lela 86] W. E. Lelandt, T. J. Ott, "Load-bWarning: mis-matched quotes alancing Heuristics and Process Behavior", *Proceedings of performance '86 and ACM sigmetrics 1986*, **14**(1), May 1986, pp. 54-69.

[Morr 86] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, , F. D. Smith, "Andrew; A Distributed Personal Computing Environment", *Communications of the ACM*, **29**,(3), March 1986, pp. 184-201.

[Nich 87] D. A. Nichols, "Using Idle Workstations in a Shared Computing Environment", *ACM Operating Rystems Review: 11th ACM Symposium on O.S. Principles* , **21**,(5), November 1987, pp. 6-12.

[Oust 88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, B. B. Welch, "The Sprite Network Operating System", *Computer*, **21**(2), February 1988, pp. 23-36.

[Rene 86] R. van Renesse, "From UNIX to a Usable Distributed Operating System", *Proceedings of the EUUG Autumn 1986 Conference*, September 1986, pp 15-21.

[Smit 88] J. M. Smith, "A Survey of Process Migration Mechanisms", *ACM Operating Systems Review*, **22**(3), July 1988, pp. 28-40.

[Tane 81] A. S. Tanenbaum, S. J. Mullender, "An Overview of the Amoeba Distributed Operating System", *ACM Operating Rystems Review*, **15**(3), July 1982, pp. 51-64.

[Tane 85] A. S. Tanenbaum, R. van Renesse, "Distributed Operating Systems", *ACM Computing Surveys*, **17**(4), December 1985, pp. 419-470.

[Tane 86] A. S. Tanenbaum, S. J. Mullender, R. van Renesse, "Using Sparse Capabilities in a Distributed Operating", *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, pp. 558-563.

[Thei 85] M. M. Theimer, A. L. Keith, D. R. Cheriton, "Preemptable Remote Execution Facilities for the V-System", *ACM Operating Systems Review: 10th ACM Symposium on O.S. Principles* **19**(5), December 1985, pp 2-12.

[Zhou 87] S. Zhou, "Performance Studies of Dynamic Load Balancing in Distributed Systems", Report No. UCB/CSD 87/376, Computer Science Division, University of California at Berkeley, October 1987.

## Appendix: system overview

| | Amoeba | Butler | Charlotte | MOS |
|---|---|---|---|---|
| Scheduling | | | | |
| Distributed vs. non distributed authority | non distributed | distributed | distributed | distributed |
| Cooperative vs. noncooperative | non cooperative | cooperative | cooperative | cooperative |
| Approximate vs. heuristic | heuristic | heuristic | * | approximate |
| Adaptive vs. nonadaptive | nonadaptive | nonadaptive | * | nonadaptive |
| Processor assignment | | | | |
| One-time assignment vs. dynamic reassignment | one-time | one-time | dynamic ‡ | dynamic |
| Load sharing vs. load balancing | load sharing | load sharing | * | load balancing |
| Remote execution vs. process migration | process migration | process migration | process migration | process migration |
| Degree of transparency | complete | not transparent for the users | complete | complete |
| Nature of parameters for choosing the target | free CPU | free CPU | * | target load: CPU utilization |
| Degree of distribution of parameters for choosing target | centralized server | group of servers | group of servers ‡ | local servers |
| Global vs. partial knowledge of the state of the system | global knowledge | partial knowledge | global knowledge ‡ | partial knowledge |
| Time frame of load distribution decisions | process creation | process creation in shell | * | periodically or when a system call is issued |
| Static vs. dynamic selection of candidate processes | dynamic selection | user selection | dynamic selection ‡ | dynamic selection |

One of the motivations for incorporating a process migration facility in Charlotte was to provide a *testbed* for the study of load sharing policies and migration algorithms. Therefore, nothing can be stated in a rigid way for the criteria marked with an "*". Similarly, "‡" indicates that load sharing policies implemented within Charlotte *may* use these facilities.

| | NEST | Sprite | V | F&Z |
|---|---|---|---|---|
| Scheduling | | | | |
| Distributed vs. non distributed authority | distributed | | distributed | distributed |
| Cooperative vs. noncooperative | cooperative | cooperative | cooperative | cooperative |
| Approximate vs. heuristic | approximate | heuristic | approximate | approximate |
| Adaptive vs. nonadaptive | adaptive | | nonadaptive | nonadaptive |
| Processor assignment | | | | |
| One-time assignment vs. dynamic reassignment | one-time | back and forth | dynamic | one-time |
| Load sharing vs. load balancing | load balancing | load sharing | load sharing | load balancing |
| Remote execution vs. process migration | process migration | process migration | remote execution, process migration | remote execution |
| Degree of transparency | transparent but can also be decided by users or in processes | | transparent but can also be decided by users or in processes | transparent |
| Nature of parameters for choosing the target | target load and average system load | | target load | target load: resources queue length |
| Degree of distribution of parameters for choosing target | local servers | | broadcast | centralized and local servers |
| Global vs. partial knowledge of the state of the system | global knowledge | | global knowledge | global knowledge |
| Time frame of load distribution decisions | process creation or system decision | process creation, machine's user return | process creation or system decision | process creation in shell |
| Static vs. dynamic selection of candidate processes | user selection, dynamic selection | user selection, dynamic selection | user selection, dynamic selection | table-driven selection |

As stated in the text, the description of the load sharing policy in Sprite is still imprecise and the implementation is incomplete: this explains the empty spaces.

# Amoeba – High Performance Distributed Computing

*Sape J. Mullender*

Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam,
The Netherlands
*sape@cwi.nl*

*ABSTRACT*

The Amoeba Project is a distributed project on distributed operating systems. The project, which started in 1980 is now a joint project of CWI and Vrije Universiteit in Amsterdam and Cambridge University in the UK. About a dozen people are working on the project, led by Prof. Dr. A. S. Tanenbaum (VU), Prof. R. M. Needham (Cambridge) and the author (CWI).

## 1. Introduction

Distributed information processing has long been practised by living organisms. The human brain, one of the most complicated living organs, functions in a highly distributed manner; different parts of the brain have specialised to perform different functions, such as speech and vision. Yet there does not seem to be any central control in the brain, 'consciousness' cannot be pinpointed to one specific group of brain cells. Not a single function of the brain seems to be impaired when any cell in the brain dies. The individual cells in living organisms die, are replaced by others, and yet, the organism as a whole continues to function uninterruptedly. Most life forms use distributed control of some form or another. Even simple life forms, such as the one-celled *amoebæ*–which have no single 'command centre' to decide where to go and how to get there–are somehow capable of co-ordinated action.

Imitating nature in all aspects, man has finally begun to incorporate the principles of distributed information processing in his most complicated artifacts, computers. In their desire to construct better, faster and more reliable information processing systems, researchers are building networks of many computers which co-operate to preform their task more quickly and more reliably.

The technology for connecting computers is available; many varieties of local-area networks are on the market, and most are fast and reliable. However, the infrastructure which is necessary to manage and control distributed information has hardly been developed. The subject of our research at CWI is the design of such an infrastructure, a model that allows people to understand distributed computer systems and describe their actions.

Distributed computing is a new research area, one that introduces a whole range of new problems to be solved, problems of managing information systems without global and up-to-date information of their state, of finding ways to prevent inconsistencies in large bodies of data caused by unsynchronised simultaneous changes. Mechanisms must be found for protecting information against unauthorised access. The potential in distributed systems of much greater reliability must be used by designing services that can survive failures of individual components of the system. For some of these problems, solutions had already been found in traditional, centralised operating systems; other problems did not even exist before the advent of distributed computing.

Take the exploitation of parallelism, for instance: If there are two different programs to be run, two processors are evidently more powerful than one; the work can be divided. But this is not so evident if there is only one program to be run. It is then much harder to put the available parallelism to use. Traditional system design methods and software engineering principles do not provide adequate methods of splitting up algorithms in independent parts which can be executed in parallel. Building distributed systems is easy. Using them is hard.

Potentially, systems built up of many processors are more reliable than traditional computers with a single cpu. If the single processor of a centralised system fails, the system comes to a halt. In a distributed system, this does not have to be the case. Every single component of the system can be replicated, so that, no matter what component fails, a subsystem is left behind that can be made to work. If one processing element fails, others can take over the work. If a disk fails, a copy of the information can still be available on another disk.

As it turns out, designing software that exploits this fault-tolerant property of such a configuration is surprisingly difficult. Standard techniques for software development are all based on the assumption that the underlying hardware is infallible. This is a perfectly proper assumption in traditional systems, where, if part of the system fails, the whole system stops working, but it is no longer true in a distributed system.

Distributed systems research concentrates on the problem of structuring the hardware and designing the operating system software in such a way that we can profit from the architecture's two most important potentials, parallelism and fault tolerance.

Distributed control plays a central role in 'avoiding single points of failure.' Specialisation and control cannot be obtained through a simple hierarchical structure as exemplified by most armies. Again, the analogy with nature teaches us that extensive hierarchical systems can exist with a control structure that provides enough redundancy to survive 'simple' failures.

The realisation of distributed control in all parts of the system is a key goal of the research: Any centralised part will be a potential bottleneck when the system grows, and a liability in the face of crashes. It is because of the importance of distributed control that we have named the distributed operating system emerging from our research **Amoeba**, after that one-celled creature using distributed control to move about.

## 2. The Amoeba Distributed Operating System

The price of processors and memory is decreasing at an incredible rate. Extrapolating from the current trend, it is likely that a single board containing a powerful cpu, several tens of megabytes of memory, and a fast network interface will be available for a manufacturing cost of a few hundred dollars in 1995. Our intention, therefore, has been to do research on the architecture and software of machines built up of a large number of such modules.

In particular, we envision three classes of machines: (1) personal computers consisting of a high-quality bit-mapped display and a few processor-memory modules; (2) departmental machines consisting of hundreds of such modules; and (3) large mainframes consisting of thousands of them. The primary difference between these machines is the number of modules, rather than the type of the modules. In principle, any of these machines can be gracefully increased in size to improve performance by adding new modules or decreased in size to allow removal and repair of defective modules. The software running on the various machines should be in essence identical. Furthermore, it should be possible to connect different machines together to form even larger machines and to partition existing machines into disjoint pieces when necessary, all in a way transparent to the user level software.

This model is superior to the oft-proposed 'Personal Computer Model' (as exemplified by Xerox PARC [Lam79a]) in a number of ways. In the personal computer model, each user has a dedicated minicomputer, complete with disks, in the office, or at home. Unfortunately, when people work together on large projects, having numerous local file systems can lead to multiple, inconsistent copies of many files. Also, the noise generated by disks in every office, and the maintenance problems generated by having machines spread all over many buildings can be annoying.

Furthermore, computer usage is very 'bursty': most of the time the user does not need any computing power, but once in a while he may need a very large amount of computing power for a short time (*e.g.*, when recompiling a program consisting of 100 files after changing a basic shared declaration). The fifth-generation computer we propose is especially well suited to bursty computation. When a user has a heavy computation to do, an appropriate number of processor-memory modules are temporarily assigned to him. When the computation is completed, they are returned to the idle pool for use by other users. This contrasts with the Cambridge Distributed Operating System, [Nee82a] which also has a 'processor bank,' but assigns a processor to a user for the duration of a login session.

A machine of the type described above requires radically different system software than existing machines. Not only must the operating system effectively use and manage a very large number of processors, but the communication and protection aspects are very different from those of existing systems.

Traditional networks and distributed systems are based on the concept of two processes or processors communicating via connections. The connections are typically managed by a hierarchy of complex protocols, usually leading to complex software and extreme inefficiency. (An effective transfer rate of 1 megabit/sec over a 10 megabit/sec local network, which is only 10% utilisation, is frequently barely achievable.)

We reject this traditional approach of viewing a distributed system as a collection of discrete processes communicating via multilayer (*e.g.*, iso) protocols, not only because it is inefficient, but because it puts too much emphasis on specific processes, and by inference, on processors. Instead we propose to base the software design on a different conceptual model–the object model. In this model, the system deals with abstract objects, each of which has some set of abstract operations that can be performed on it.

Associated with each object are one or more *'capabilities'* [Den66a] which are used to control access to the object, both in terms of who may use the object and what operations he may perform on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections. For example, a typical object is the file, with operations to read and write portions of it.

The object model is well-known in the programming languages community under the name of *'abstract data type'*. [Lis74a] This model is especially well-suited to a distributed system, because in many cases an abstract data type can be implemented on one of the processor-memory modules described above. When a user process executes one of the visible functions in an abstract data type, the system arranges for the necessary underlying message transport from the user's machine to that of the abstract data type and back. The header of the message can specify which operation is to be performed on which object. This arrangement gives a very clear separation between users and objects, and makes it impossible for a user to directly inspect the representation of an abstract data type by bypassing the functional interface.

A major advantage of the object or abstract data type model is that the semantics are inherently location independent. The concept of performing an operation on an object does not require the user to be aware of where objects are located or how the communication is actually implemented. This property gives the system the possibility of moving objects around to position them close to where they are frequently used. Furthermore, the issue of how many processes are involved in carrying out an operation, and where they are located is also hidden from the user.

It is convenient to *implement* the object model in terms of clients (users) who send messages to services. [Che83b, Nee82b, Bal79a] A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies. The design of these servers and the design of the protocols they use form an important part of the system software of our proposed fifth-generation computers.

As an example of the problems that must be solved, consider a file server. Among other design issues that must be dealt with are how and where information is stored, how and when it is moved, how it is backed up, how concurrent reads and writes are controlled, how local caches are maintained, how information is named, and how accounting and protection are accomplished. Furthermore, the internal structure of the service must be designed: how many server processes are there, where are they located, how and when do they communicate, what happens when one of them fails, how is a server process organised internally for both reliability and high performance, and so on. Analogous questions arise for all the other servers that comprise the basic system software.

## 3. Communication Primitives and Protocols

In the literature about computer networks, one finds much discussion of the iso Reference Model for Open Systems Interconnection (OSI) [Zim80a] these days. It is our belief that the price that must be paid in terms of complexity and performance in order to achieve an 'open' system in the iso sense is much too high, so we have developed a much simpler set of communication primitives, which we will now describe.

Instead of a 7-layer protocol, we effectively have a 4-layer protocol. The bottom layer is the Physical Layer, and deals with the electrical, mechanical and similar aspects of the network hardware. The next layer is the Port Layer, and deals with the location of services, the transport of large (up to 1 gigabyte) datagrams (messages whose delivery is not guaranteed) from source to destination and enforces the protection mechanism, which will be discussed in the next section. On top of this we have a layer that deals with the reliable transport of bounded length (1 gigabyte) requests and replies between client and server. We have called this layer the Transaction Layer. The final layer has to do with the semantics of the requests and replies, for example, given that one can talk to the file server, what commands does it understand. The bottom three layers (Physical, Port, and Transaction) are implemented by the kernel and hardware; only the Transaction Layer interface is visible to users. User programs execute in the fourth layer, the Application Layer.

The main function of the Transaction Layer is to provide an end-to-end *message* service built on top of the underlying *datagram* service, the main difference being that the former uses timers and acknowledgements to guarantee delivery whereas the latter does not.

The Transaction Layer protocol is straightforward. A *server process* makes a call to *getreq* (an abbreviation of *get request*) to tell the Transaction Layer it is ready to receive a request from a client. The client sends a request by calling *trans* (for *transaction*), which makes the Transaction Layer send a request and wait until a reply comes back from the server. The client is blocked until this reply arrives. The server, after carrying out the request, returns a reply by a call to *putrep*.

When the client does a *trans*, a packet, or sequence of packets, containing the request is sent to the server, the client is blocked, and a timer is started (inside the Transaction Layer). If the server does not acknowledge receipt of the request packet before the timer expires (usually by sending the reply, but in some special cases by sending a separate acknowledgement packet), the Transaction Layer retransmits the packet again and restarts the timer. When the reply finally comes in, the client sends back an acknowledgement (possibly piggybacked onto the next request packet) to allow the server to release any resources, such as buffers, that were acquired for this transaction. Under normal circumstances, reading a long file, for example, consists of the sequence

| | |
|---|---|
| From client: | request for block 0 |
| From server: | here is block 0 |
| From client: | acknowledgement for block 0 and request for block 1 |
| From server: | here is block 1 |
| | *etc.* |

The protocol can handle the situation of a server crashing and being rebooted quite easily since each request contains the identity of the file to be read and the position in the file to start reading. Between requests, the server has no 'activation record' or other table entry whose loss during a crash causes the server to forget which files were open, *etc.*, because no concept of an open file or a current position in a file exists on the server's side. Each new request is completely self-contained. Of course for efficiency reasons, a server may keep a cache of frequently accessed *i-nodes*, file blocks *etc.*, but these are not essential and their loss during a crash will merely slow the server down slightly while they are being dynamically refreshed after a reboot.

The Port Layer is responsible for the speedy transmission of datagrams. The Port Layer need only do this reasonably reliably, and does not have to make an effort to guarantee the correct delivery of every datagram. This is the responsibility of the Transaction Layer. Our results show that, compared to other approaches, our's leads to significantly higher transmission speeds, due to simpler protocols.

Theoretically, very high speeds are achievable in modern local-area networks. In practice, however, speeds of only a few kilobytes per second between user processes on a megabyte per second network have rarely been achieved. Obviously, to achieve higher transmission rates, the overhead of the protocol must be kept very low indeed. To do this, a large datagram size was chosen for the Port Layer, which has to split up the datagrams into small packets that the network hardware can cope with. This approach allows the implementor of the Port Layer to exploit the possibilities that the hardware has to offer to obtain an efficient stream of packets.

Two versions of the algorithm have now been implemented. The one described above has been implemented on native Amoeba and achieves over 650,000 bytes a second from user process to user process (using SUN 3/50s over Ethernet). A second implementation runs under UNIX, and gets 225,000 bytes/sec across the Ethernet between two SUN 3/50s running SUN UNIX, without causing a significant load on the system itself.

## 4. Ports

Every service has one or more *ports* [Mul84a] to which client processes can send messages to contact the service. Ports consist of large numbers, typically 48 bits, which are known only to the server processes that comprise the service, and to the service's clients. For a public service, such as the system file service, the port will be generally made known to all users. The ports used by an ordinary user process will, in general, be kept secret. Knowledge of a port is taken by the system as *prima facie* evidence that the sender has a right to communicate with the service. Of course the service is not required to carry out work for clients just because they know the port, for example, the public file service may refuse to read or write files for clients lacking account numbers, appropriate authorisation, *etc.*

Communication using ports basically works as follows. One, or several server processes make a call to

*getreq*(*serverport*, · · · );

a client, wishing to have some service rendered, calls

*trans*(*serverport*, · · · );

The client's Transaction Layer generates a (unique) *reply port* for the client, finds an active server process on *serverport* and sends a message containing

{*serverport*, *replyport*, · · · }

to the server. The server processes the request, and returns a reply in a message

{*replyport*, *serverport*, · · · }

The two ports provide a unique identification of the transaction.

The difference between this approach and that taken by conventional interprocess communication protocols is that, in principle, messages are addressed to a service name or *port*, not to a machine, or a process. Obviously, in Amoeba, clients never need to know where a service is implemented, or how many server processes there are. This is none of their business; it is part of the *implementation* of the 'abstract data type' that is the service.

The transaction mechanisms, however, must deliver requests and replies to specific processes on specific machines. Obviously, inside the Port Layer–which is, after all, responsible for message delivery–ports must be mapped onto networks addresses. So, when a client calls *trans*, the client's Port Layer must *find* a network node where a *getreq* on the matching port is outstanding.

In the local-area network, the technique for *locating* a port is the following. The client broadcasts a tiny message, saying *'anyone listening on port* x*?'*, to which servers listening on that port reply *'port* x *is at machine* y*!'* A similar technique is used for locating the client for the reply message.

Locating ports is inefficient. It can be sped up, however, by a simple technique which finds many useful applications in distributed systems: the *hint*. In this particular case, hints are stored in a little table of (*port, network address*) pairs in every host and they say in effect: *'If you're looking for port* x, *why don't you try network address* y*?'* If the hint works, a port has been located without sending any extra messages; if not, a message is returned saying that the port is not known at that address. In the latter case, the hint is scratched out, and a broadcast locate is done. Incoming packets contain a source address, so they provide a free hint for their source port.

Although the port mechanism provides a convenient way to provide partial authentication of clients ('if you know the port, you may at least talk to the service'), it does not deal with the authentication of servers. The primitive operations offered by the system are *trans*, *putreq* and *getrep*. Since everyone knows the port of the file server, as an example, how does one ensure that malicious users do not execute *getreq*s on the file server's port, in effect impersonating the file server to the rest of the system?

One approach is to have all ports manipulated by kernels that are presumed trustworthy and are supposed to know who may *getreq* from which port. [Che83a, Ras81a] We reject this strategy because some machines, *e.g.*, personal computers connected to larger multimodule systems, may not be trustworthy, and also because we believe that by making the kernel as small as possible, we can enhance the reliability of the system as a whole. Instead, we have chosen a different solution that can be implemented in either hardware or software.

In the hardware solution, we need to place a small interface box, which we call an F-box (Function-box) between each processor module and the network. The most logical place to put it is on the VLSI chip that is used to interface to the network. Alternatively, it can be put on a small printed circuit board inside the wall socket through which personal computers attach to the network. In those cases where the processors have user mode and kernel mode and a trusted operating system running in kernel mode, it can also be put into operating system software. In any event, we assume that somehow or other all packets entering and leaving every processor undergo a simple transformation that users cannot bypass.

The transformation works like this. Each port is really a pair of ports, $P$, and $G$, related by: $P = F(G)$, where $F$ is a (publicly-known) one-way function [Pur74a, Eva74a] The one-way function has the property that given $G$ it is a straightforward computation to find $P$, but that given $P$, finding $G$ is so difficult that the only approach is to try every possible $G$ to see which one produces $P$. If $P$ and $G$ contain sufficient bits, this approach can be made to take millions of years on the world's largest supercomputer, thus making it effectively impossible to find $G$ given only $P$. Note that a one-way function differs from a cryptographic transformation in that the latter must have an inverse to be useful, but the former has been carefully chosen so that no inverse can be found.

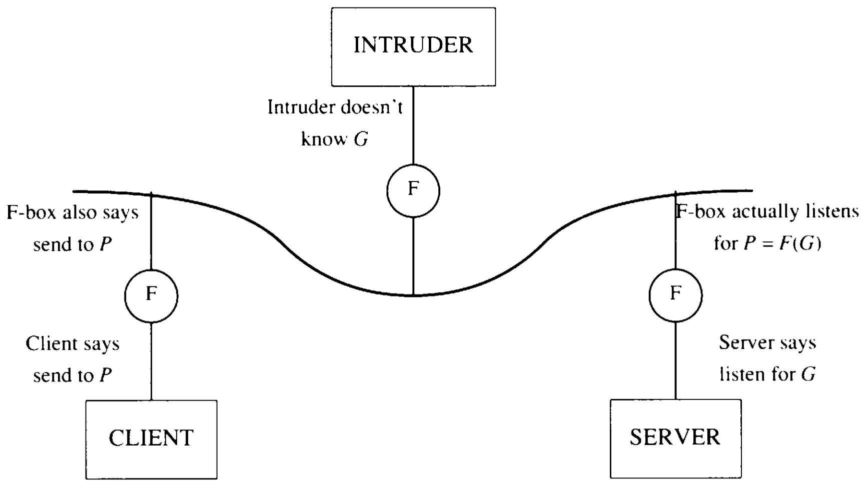

INTRUDER
Intruder doesn't know *G*
F
F-box also says send to *P*
F-box actually listens for *P* = *F*(*G*)
F
F
Client says send to *P*
Server says listen for *G*
CLIENT
SERVER

**Figure 1**

Using the one-way F-box, the server authentication can be handled in a simple way, illustrated in Figure 1. Each server chooses a get-port, $G$, and computes the corresponding put-port, $P$. The get-port is kept secret; the put-port is distributed to potential clients, or, in the case of public servers, is published. When the server is ready to accept client requests, it does a $getreq(G, \cap, req)$. The F-box then computes $P = F(G)$ and waits for packets containing $P$ to arrive. When one arrives, it is given to the appropriate process. To send a packet to the server, the client merely does $trans(cap, req, rep)$. *Cap* is a *capability**, giving the identity of the object the user wants to access, which contains the *port* field, $P$, of the service managing the object. This will cause a datagram to be sent by the local F-box with $P$ in the destination-port field of the header. The F-box on the sender's side does not perform any transformation on the $P$ field of the outgoing packet.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do $getreq(G, \cdots)$. However, $G$ is a well-kept secret, and is never transmitted on the network. Since we have assumed that $G$ cannot be deduced from $P$ (the one-way property of $F$) and that the intruder cannot circumvent the F-box, he cannot intercept packets not intended for him. Replies from the server to the client are protected the same way: The client's Transaction Layer picks a get-port for the reply, say, $G'$, and the client's F-box transforms $G'$ into $P' = F(G')$ in the request packet for the server to use as the put-port to send the reply to.

The presence of the F-box makes it easy to implement digital signatures for still further authentication, if that is desired. To do so, each client chooses a random signature, $S$, and publishes $F(S)$. The F-box must be designed to work as follows. Each packet presented to the F-box contains three special header fields: destination ($P$), reply ($G'$), and signature ($S$). The F-box applies the one-way function to the second and third of these, transmitting the three ports as: $P$, $F(G')$, and $F(S)$, respectively. The first is used by the

Capabilities are explained in the next section.

receiver's F-box to admit only packets for which the corresponding *getreq* has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the signature will know what number to put in the third field to ensure that the publicly-known $F(S)$ comes out.

It is important to note that the F-box arrangement merely provides a simple *mechanism* for implementing security and protection, but gives operating system designers considerable latitude for choosing various *policies*. The mechanism is sufficiently flexible and general that it should be possible to put it into hardware without precluding many as-yet-unthought-of operating systems to be designed in the future.

## 5. Capabilities

In any object-based system, a mechanism is needed to keep track of which processes may access which objects and in what way. The normal way is to associate a capability with each object, with bits in the capability indicating which operations the holder of the capability may perform. In a distributed system this mechanism should itself be distributed, that is, not centralised in a single monolithic 'capability manager'. In our proposed scheme, each object is managed by some service, which is a user (as opposed to kernel) program, and which understands the capabilities for its objects.

| | | | |
|---|---|---|---|
| SERVER | OBJECT | RIGHTS | CHECK |

**Figure 2**

A capability typically consists of four fields, as illustrated in Figure 2:

1. The put-port of the service that manages the object
2. An Object Number meaningful only to the service managing the object
3. A Rights Field, which contains a 1 bit for each permitted operation
4. A Check Field for protecting each object

The basic model of how capabilities are used can be illustrated by a simple example: a client wishes to create a file using the file service, write some data into the file, and then give another client permission to read (but not modify) the file just written. To start with, the client sends a message to the file service's put-port specifying that a file is to be created. The request might contain a file name, account number and similar attributes, depending on the exact nature of the file service. The server would then pick a random number, store this number in its object table, and insert it into the newly-formed object capability. The reply would contain this capability for the newly created (empty) file.

To write the file, the client would send a message containing the capability and some data. When the *write* request arrived at the file server process, the server would normally use the object number contained in the capability as as index into its tables to find the object. For a file server, the object number would be the i-node number, which could be used to locate the i-node.

Several object protection systems are possible using this framework. In the simplest one, the server merely compares the random number in the file table (put there by the server when the object was created) to the one contained in the capability. If they agree, the capability is assumed to be genuine, and all operations on the file are allowed. This system is easy to implement, but does not distinguish between *read*, *write*, *delete*, and other operations that may be performed on objects.

However, it can easily be modified to provide that distinction. In the modified version, when a file (object) is created, the check field is computed by applying a one-way function to Object Number, Rights Field, and the Random Number stored with the object. When the capability is returned for use, the server uses the object number to find the file table and hence the random number. If the result of recomputing the Check Field leads to the Check Field in the capability, it is almost assuredly valid, and the Rights Field can be believed. Clearly, an encryption function that mixes the bits thoroughly is required to ensure that tampering with the Rights Field also affects the Check Field.

When this modified protection system is used, the owner of the object can easily give an exact copy of the capability to another process by just sending it the bit pattern, but passing, say, read-only access, is harder. To accomplish this task, the process must send the capability back to the server with a bit-map saying which bits to strip off the Rights Field. By choosing the bit mask carefully, the capability owner can mask out any operations that the recipient is not permitted to carry out.

## 6. Process Service

Managing processes on Amoeba is a task that is carried out by a number of co-operating services. The central one is the service provided by the Amoeba operating system kernel. The kernel assigns processes to the processor so they can do their work. But there's more to process management: processes have to be assigned to the right processor; when a process crashes, or does something irregular, such as attempting to make a Unix system call on an Amoeba kernel, something has to be done about it; programs have to be fetched from a file before they can be run.

The design of the system is such that the Amoeba kernel implements a minimum of basic process management *mechanisms*, on top of which various *policies* can be implemented: If we ever decide to do process management in a different way, we want to run as little risk as possible that we have to change the kernel.

In Section 3, we've explained that client processes block when they do transactions, and that server processes block when they wait for a client's request. It is very difficult to write programs using non-blocking transactions. It's much simpler using blocking ones. Additionally, blocking transactions can be implemented much more efficiently than non-blocking ones. To achieve parallelism, one uses parallel processes.

In traditional operating systems, each process runs in its own address space. In distributed systems, processes are created at such an enormous rate, that the cost of making each one run in its own address space is prohibitive. Many distributed systems, therefore, provide *light-weight processes*, processes that share a single address space. Processes then have very little context, and process switching can be done very efficiently.

We call a light-weight process a *thread*, and a group of threads, sharing an address space is a *process*. The address space is divided up into *segments*. A process can have a number of read-only segments (useful to hold the program's code), and a number of read-write segments, write-only segments (they can be useful, believe it or not), segments that can grow (for the stack), and so on.

An executable file on Amoeba consists of a *process descriptor*, followed by several (usually two) segments containing the code and initialized data, and optionally a segment containing the symbol table. The process descriptor describes the initial state of the process: the number of threads, their program counters and stack pointers, contents of registers, the memory map and a number of other things.

To run a program, one sends the process descriptor of its executable file to a machine. The kernel then allocates memory for the new process using information from the memory map, fills the initialized memory by reading the segment contents from files whose capabilities are also in the memory map, zeroes all other segments, creates the required number of threads and starts the process.

When a running process is stopped, the Amoeba Kernel hands over a process descriptor for it, describing the state of the process at the moment it was stopped.

A process can easily be migrated to another machine by stopping it, giving its process descriptor to another machine, starting the new process and killing the old one.

Cluster descriptors also play an important role in exception handling. When an exception occurs, the kernel sends the process descriptor to the specified server for handling. The handler can examine and manipulate the process using the information provided by the process descriptor, and access the process' memory through the segment capabilities in the process descriptor.

Operating system emulation can be viewed as a special case of debugging. A program, native to another operating system, can be run on Amoeba as if it ran on the operating system it was written for. Before the process is run, its environment is set up so that all system calls it does, all actions that cause exceptions are trapped to an operating system emulator server. The emulator can examine the state of the excepted process, determine what its original operating system would have done when this exception occurred and simulate that with the same mechanisms that the debugger uses.

The Amoeba process service mechanism thus provides a basic mechanism in the Amoeba kernel for process management (segments and process descriptors), which is used by user-space servers to augment this service with services for remote execution, load balancing through migration, local and remote debugging, checkpointing and operating system emulation.

## 7. Files and Directories

The Amoeba file system and directory systems are realised as separate services. This allows the directory system to be used as a general object-naming facility rather than just a file-naming system. Directories map ascii path names into capabilities. The file system stores immutable byte arrays and uses capabilities for naming them.

A directory contains ascii string / capability pairs. A lookup operation has two input parameters, a *capability* of a directory and a *string*, and one output parameter, the *capability* associated with the input string in the directory referred to by the input capability. A path name is resolved by doing this iteratively for each path-name component.

When a user logs in to Amoeba (a proper authentication server still needs to be developed for this) the capability for the user's *home directory* is given to the command interpreter, which passes it on to the processes started up by it. All paths are interpreted from this home directory. Thus, the Amoeba file tree is not singly rooted, but has one root per user.

The directory server is replicated for reliability and also has the capability to assist the users in replicating any objects whose capabilities are stored by it.

The file server is primarily made for speed: files are stored consecutively on disk and are usually read and written as a whole. Files are immutable so the way to modify a file is to create a modified copy and to replace the capability of the old version by the new one in the directory server so that the file name refers to the new version instead of the old one.

The directory server has support for atomic update of many directory entries so that applications can maintain any desired degree of consistency.

Since files are immutable they can be cached without fear of cache entries becoming stale. Caching is used extensively; the file servers maintain 32 megabyte caches which have extremely high hit rates. Thus, reading a one kilobyte file from the file server (over the network) takes only 3 milliseconds on average, even though the rotational and disk arm delays of the disks are typically 30 milliseconds or more.

Storing files consecutively on disk allows the file servers to do very large disk writes, thus obtaining very good throughput to and from disk. A client program can maintain a file write rate of 650 kilobytes per second indefinitely (until the disk fills up), high enough, for instance, for storing live video.

## 8. Bank service

The bank service is the heart of the resource management mechanism. It implements an object called a "bank account" with operations to transfer virtual money between accounts and to inspect the status of accounts. Bank accounts come in two varieties: individual and business. Most users of the system will just have one individual account containing all their virtual money. This money is used to pay for cpu time, disk blocks, typesetter pages, and all other resources for which the service owning the resource decides to levy a charge.

Business accounts are used by services to keep track of who has paid them and how much. Each business account has a subaccount for each registered client. When a client transfers money from his individual account to the service's business account, the money transferred is kept in the subaccount for that client, so the service can later ascertain each client's balance. As an example of how this mechanism works, a file service could charge for each disk block written, deducting some amount from the client's balance. When the balance reached zero, no more blocks could be written. Large advance payments and simple caching strategies can reduce the number of messages sent to a small number.

Another aspect of the bank service is its maintenance of multiple currencies. It can keep track of say, virtual dollars, virtual yen, virtual guilders and other virtual currencies, with or without the possibility of conversion among them. This feature makes it easy for subsystem designers to create new currencies and control how they are allocated among the subsystems users.

The bank service described above allows different subsystems to have different accounting policies. For example, a file or block service could decide to use either a buy-sell or a rental model for accounting. In the former, whenever a block was allocated to a client, the client's account with the service would be debited by the cost of one block. When the block was freed, the account would be credited. This scheme provides a way to implement absolute limits (quotas) on resource use. In the latter model, the client is charged for rental of blocks at a rate of X units per kiloblock-second or block-month or something else. In this model, virtual money is constantly flowing from the clients to the servers, in which case clients need some form of income to keep them going. The policy about how income is generated and dispensed is determined by the owner of the currency in question, and is outside the scope of the bank server.

## References

Bal79a. J. E. Ball, E. J. Burke, I. Gertner, K. A. Lantz, and R. F. Rashid, ''Perspectives on Message-Based Distributed Computing,'' *Proc. IEEE*, 1979.

Che83a. D. R. Cheriton and W. Zwaenepoel, ''The Distributed V Kernel and its Performance for Diskless Workstations,'' *Proc. Ninth ACM Symp. on Operating Systems Principles*, pp. 128-140, New York, October 1983.

Che83b. D. R. Cheriton and W. Zwaenepoel, ''The Distributed V Kernel and its Performance for Diskless Workstations,'' *Proc. Ninth ACM Symp. on Operating Systems Principles*, pp. 128-140, New York, October 1983.

Den66a. J. B. Dennis and E. C. Van Horn, ''Programming Semantics for Multiprogrammed Computations,'' *CACM*, vol. 9, no. 3, pp. 143-155, March 1966.

Eva74a. A. Evans, W. Kantrowitz, and E. Weiss, ''A User Authentication Scheme Not Requiring Secrecy in the Computer,'' *Comm. ACM*, vol. 17, no. 8, pp. 437-442, August 1974.

Lam79a. B. W. Lampson and R. F. Sproull, ''An Open Operating System For A Single User Machine,'' *Proc. Seventh Symp. on Oper. Syst. Prin.*, pp. 98-105, 1979.

Lis74a. B. Liskov and S. N. Zilles, ''Programming with abstract data types,'' *Proceedings of the ACM SIGPLAN Conference on Very High Level Languages*, vol. 9, no. 4, pp. 50-59, Sigplan Notices, 1974.

Mul84a. S. J. Mullender and A. S. Tanenbaum, ''Protection and Resource Control in Distributed Operating Systems,'' *Computer Networks*, vol. 8, no. 5,6, pp. 421-432, 1984.

Nee82a. R. M. Needham and A. J. Herbert, *The Cambridge Distributed Computer System*, Addison-Wesley, Reading, Ma., 1982.

Nee82b. R. M. Needham and A. J. Herbert, *The Cambridge Distributed Computer System*, Addison-Wesley, Reading, Ma., 1982.

Pur74a. G. B. Purdy, ''A High Security Log-in Procedure,'' *Comm. ACM*, vol. 17, no. 8, pp. 442-445, August 1974.

Ras81a. R. F. Rashid and G. Robertson, ''Accent: A Communication Oriented Network Operating System Kernel,'' *Proc. Eighth Symp. Operating Syst. Prin.*, pp. 64-75, ACM, 1981.

Zim80a. H. Zimmermann, ''OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection,'' *IEEE Trans. Comm.*, vol. COM-28, pp. 425-432, April 1980.

# Implementation of an Event Distribution Mechanism (EDM) on a Network of Workstations

*Martin Zellweger*

Institut für Informatik
University of Zurich and
ACU Systems AG, Switzerland

*Peter Gloor, Rudolf Marty*

Institut für Informatik
University of Zurich, Switzerland
*gloor@ifi.unizh.ch*
*gloor@unizh.uucp*

*ABSTRACT*

This paper describes an event handling mechanism for distributed systems. Processes, which are distributed over a network of nodes, can then communicate easily by means of events which may be sent over the network. The notion of a "blocking event" serves as a means to synchronize the distributed processes. An implementation of the event distribution mechanism on a network of Sun workstations using Apollo's NCS is described. Finally some applications of the EDM are briefly mentioned.

## 1. Motivation

Operating systems like Quicksilver [Has88], which is based on a transaction man ager, or distributed software development systems like ARGUS [Lis87] are centred around the transaction concept. We think that this concept is too coarse a basic building block in distributed fault tolerant applications. Therefore, we propose a lower level mechanism which can be used for the construction of higher level synchronization mechanisms. This lower level synchronization mechanism is based upon the fundamental notion of the *event*.

We came to this result considering object oriented design practices: One of its basic concept is the "active object" [Int86]. In "programming with active objects" there must not necessarily be an "active" object, i.e. an object which is ultimately represented by its own process. Rather, this idea has to be interpreted on an abstract level by modeling the various objects as independent entities which are responsible for their own integrity and data consistency. To communicate, the objects are sending events to each other. Therefore, we suggest an **E**vent **D**istribution **M**echanism (EDM) which enables the objects (i.e. in a conventional operating system: processes) to communicate by exchanging events and by reacting to the reception of a particular event by the execution of eventhandler functions.

## 2. Description of the Event Distribution Mechanism (EDM)

The mechanism presented here (the EDM) allows event driven programming on the operating system level. It will be used for the transparent distribution of events over a network. All actions executed in the environment of the EDM can be subdivided into two categories:

- *Declaring an interest:*
  In order to be able to react to the reception of an event belonging to a particular event class by the execution of a sequence of actions, a process first has to declare its interest for this event class. This is done by passing the interest to the local scheduler process which in under certain circumstances, also forward the interest to the other schedulers.

- *Triggering an event:*
  If a process wishes to trigger an event, it has to pass the event to the local scheduler which will send the event to all schedulers on the network (and, through a shortcut, also to itself). The schedulers then will activate all processes which have declared an interest for this event class.

On every network node there has to be a dedicated scheduler process which has

- to distribute events passed by local processes (i.e. processes running on the same machine) to remote schedulers (i.e. schedulers on remote machines).
- to receive events from remote schedulers and to initiate the appropriate action for a particular event.

Every process which wishes to trigger an event passes this event to the local scheduler. The local scheduler keeps the events in a FIFO queue and invokes the actions be longing to the first event in its queue. As soon as the scheduler has terminated the invocation of the actions belonging to the previous event, it takes the next event from the queue, etc. In order to know which process is waiting on what event(s), the scheduler keeps an interest list. The interest list holds a description of the event classes upon which a process is waiting for, i.e. for which a process has declared an interest. Every event received by the scheduler will be compared with all interests in its interest list. If the scheduler detects a match between interest and event, it initiates the actions specified by the process which declared an interest for this event class. If more than one process (possibly on different machines) have declared an interest for this event class, they all will be notified in order to be able to react to the event.

This mechanism may by distributed transparently over a network of machines. In this case, the scheduler has to be replicated on every machine. All interprocess communication and synchronization is handled transparently by the schedulers. A process passes an event to the local scheduler, which not only compares the event with its own interest list, but also forwards the event to all other (remote) schedulers on the network. The remote schedulers compare the event with their own local interest lists too, and initiate the necessary actions.

In order to make EDM practicable as a synchronization mechanism, the additional notion of the "*blocking event*" is introduced. The declaration for a blocking event delays the invocation of the next event-handler-function of the same event class, until the execution of the function for the last event of this class has been terminated on all machines on the network, i.e. there can only be *one event of the same blocking event class* be triggered and the execution of the eventhandler-function be invoked simultaneously. This turns EDM into a networkwide synchronization mechanism, because access to critical resources can be serialized this way in a very comfortable manner.

## 3. Implementation Environment

The EDM prototype runs on a network of Sun workstations. It is implemented in C and uses the Network Computing System (NCS, OSF standard) from Apollo. NCS offers quite an easy way to distribute information on replicated servers by means of reliable broadcast, meaning that, if a server is up, it will get the broadcast message. Therefore, the schedulers can be kept stateless and all state problems be solved by the NCS kernel.

### 3.1. NCS – Network Computing System

NCS was originally developed for the operating system Domain/IX of the Apollo workstations. At the time there are versions available for UNIX, VAX/VMS and MS-DOS. NCS works with UNIX BSD4.2 sockets for interprocess communication and uses remote procedure calls. The following section gives a short overview of the system.

### Construction of NCS

NCS consists of three main components:

- Remote Procedure Call (RPC) runtime library
- Network Interface Definition Language (NIDL) Compiler
- Location Broker

The Location Broker and the RPC runtime library build upon the Network Computing Kernel. The kernel and the NIDL Compiler allow the implementation of distributed applications.

The RPC runtime library contains routines, tables and data that support local programs to execute procedures on remote hosts. The routines submit requests of clients (programs that call procedures) and return answers of servers (programs which execute the procedures).

The interface between server and client can be specified by means of the Network Interface Definition Language (NIDL). The NIDL compiler translates the NIDL interface into server and client stubs, which are linked into client and server programs. Each server is identified by an object Universal Unique Identifier (UUID), an objecttype UUID and an interface address UUID.

An additional process, a Local Location Broker Daemon (LLBD) runs on every network client (workstation). It provides information about locally active objects and interfa ces of NCS servers. One Global Location Broker Daemon (GLBD) anywhere on the network manages informations about resources available to all clients. Several library calls al low servers and clients to insert and extract inquiries about active objects.

## Usage of NCS / Client - Server Model

Using NCS a client can execute procedures implemented in a server program. The system does not make any difference between local and remote servers. When a server process is activated it listens to a socket. If more than one RPC arrives, the server deals with the first incoming request. The rest is entered in a FIFO waiting queue. Thus the server acts as sequentialization process for parallel requests. The client always waits for the server to finish the execution of a calling procedure. The communication between server and client works by exchange of parameters (in, out).

## 4. Components of the EDM

An application programmer uses the EDM by means of four library calls:

- *edm_declare()*
- *edm_delete()*
- *edm_await()*
- *edm_send()*

The first three of this calls are used by the processes which are waiting for an event. *edm_declare* and *edm_delete* notify resp. delete an interest of a process in an event in the waiting queue of the local scheduler. With *edm_await* a process waits until a specified event is triggered by another process. With *edm_send* a process can trigger an event by sending the eventname to the local scheduler.

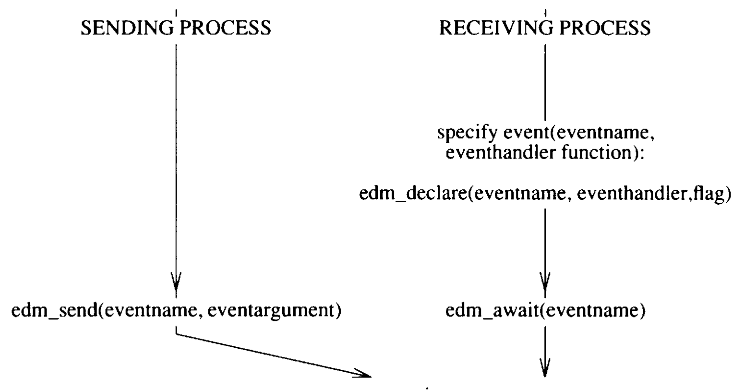

SENDING PROCESS
RECEIVING PROCESS
specify event(eventname, eventhandler function):
edm_declare(eventname, eventhandler,flag)
edm_send(eventname, eventargument)
edm_await(eventname)

**Figure 1**: *The usage of the EDM library calls*

On each network client a scheduler, which is responsible for the synchronization of the EDM is running. For this purpose it manages an interest list (INTEREST_LIST) and a token list (TOKEN_LIST). The scheduler executes remote procedure calls. Therefore it makes no differences between messages coming from local or from remote clients or other schedulers.
The actual implementation of EDM includes an additional server process named edm_transmitter. It transmits messages between different schedulers. (cp. section 5.4)

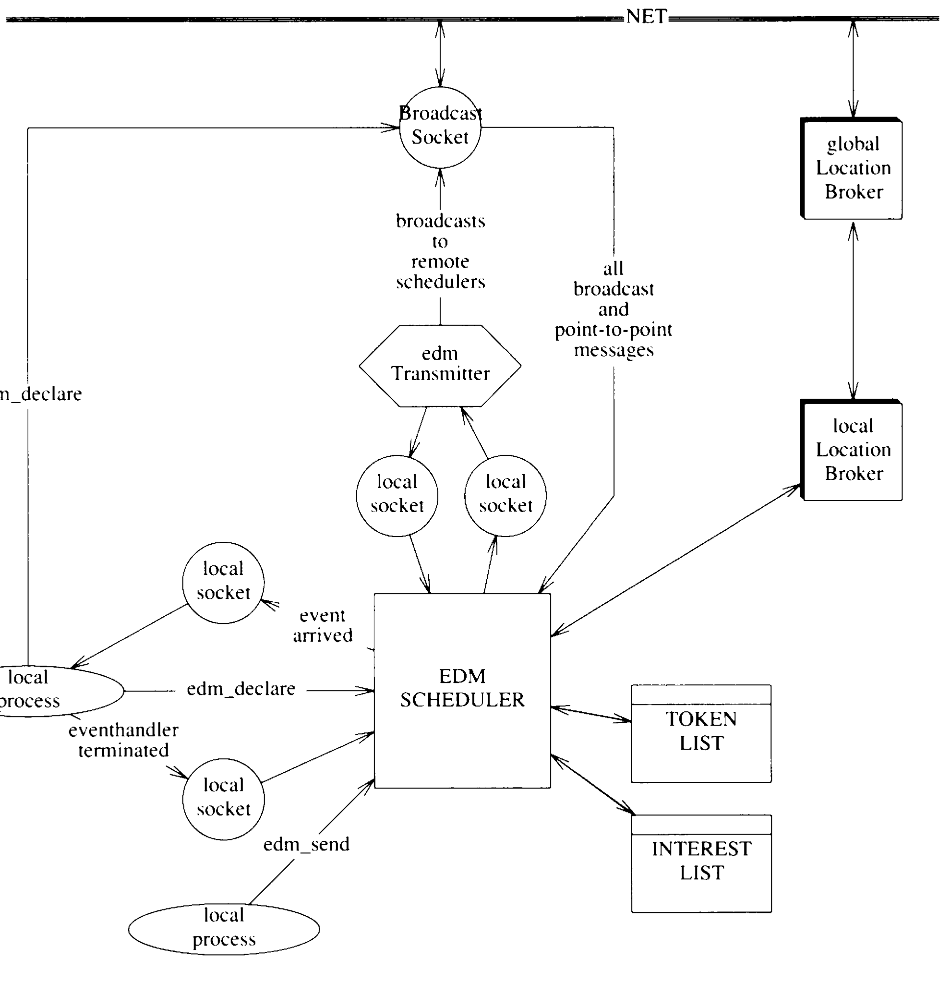


**Figure 2**: *The components of the EDM on one machine*

## 5. Implementation

### 5.1. EDM Library Calls

Except edm_await all EDM library calls are using the NCS RPC mechanism. Depending on the interface definition they are executed by the local or by the remote schedulers. The definition of the interface is described in the next section (5.2).

### Declaration of an Interest

*edm_declare* allows the declaration of blocking (BLK) and non-blocking (NOBLK) events. This call sends the eventname (character string), the name of the eventhandler (C function) and a flag (BLK, NOBLK) to all active schedulers. If the flag is NOBLK only the local scheduler gets a message.
After the notification of the scheduler *edm_declare* creates a child process by a UNIX fork. It creates a local socket and waits until it gets a trigger of the declared event. As a reaction to every incoming event (including eventual argument) the child executes the eventhandler-function. Initiation and termination of the eventhandler-function are announced to the parent process which interrupts its normal work. Upon

termination of an eventhandler-function which was triggered by a blocking event the local scheduler is notified.

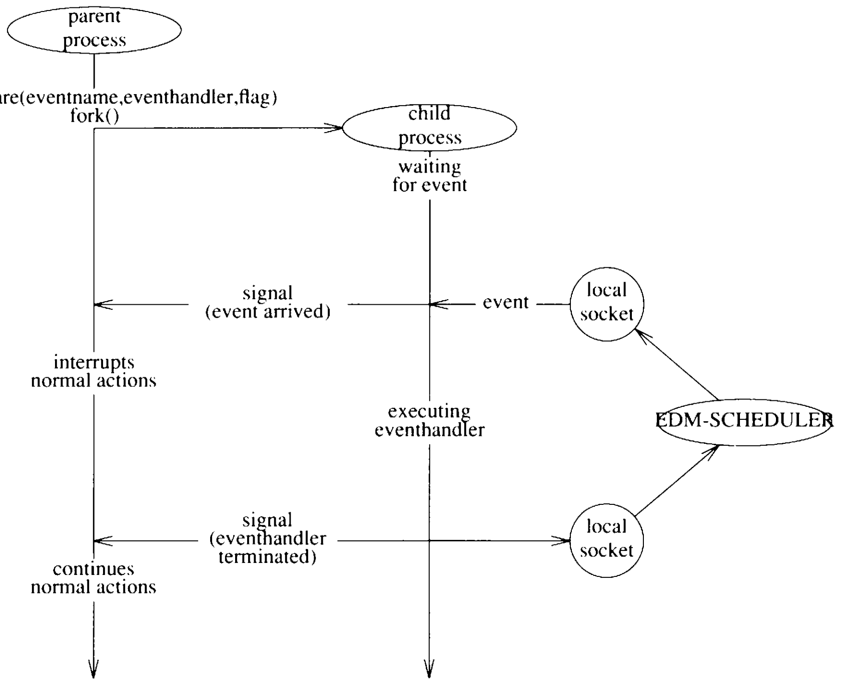


**Figure 3**: edm_declare *creating a child process*

### Delete an Interest

*edm_delete* sends the eventname and process identifier (pid) by means of a broadcast to all active schedulers. As a reaction all schedulers will delete all entries for this event declared by the specified process in the interest list. As a result the child process created by the interested process will be terminated.

### Wait for an Event

*edm_await* lets the calling process wait for the event specified by the eventname. It continues after the termination of the eventhandler-function.

### Trigger an Event

*edm_send* triggers the event (eventname) in the whole network, but the eventname and (optionally) an argument (character string) are sent to the local scheduler. Afterwards the local scheduler is responsible for the distribution of this event. In case of a blocking event this process has to wait for the termination of all events of the same eventclass which have been triggered earlier.

## 5.2. Interface Definition of EDM Remote Procedure Calls

The Interfaces of the remote procedure calls used by the EDM are described by means of the Network Interface Definition Language (NIDL).

### Identification of EDM Schedulers

Each remote procedure call addresses its communication partner with a RPC handl e. This handle is a pointer to an opaque structure that includes the information needed to access a remote object. These structures are manipulated indirectly through RPC runtime library calls. A RPC handle has to be bound to an identification address (Universal Unique Identifier, UUID) which will be registered by the Location Broker. All EDM schedulers get their own UUID's bound to two or more handles which are used as object and type identifier (objectID, typeID). When a scheduler receives a RPC of a client it first compares the incoming handle with its own handle to verify the destination address.

### Broadcasting with Return Values

RPC's using a broadcast socket are defined with a broadcast label within the interface definition. The disadvantage of broadcasts lies in the fact that schedulers can't return values back to the client. As a workaround EDM uses all RPC's as broadcasts and directly addressed as well. For this purpose each scheduler uses two objectID's:
The objectID and the typeID (both handles) used for broadcasts are the same for all schedulers in a network. Every scheduler owns an additional personal objectID with a common typeID for point-to-point communication. A client which wants to spread a broadcast message uses the broadcast objectID. To get a return value (out-parameter) of all schedulers a client demands the ID's of all active objects with a point-to-point typeID from the Location Broker and communicates with them sequentially.

## 5.3. EDM - Scheduler

The EDM scheduler executes the server part of the RPC code. Its main work is the management of the INTEREST_LIST and the TOKEN_LIST. It registers interests for clients for specified events. It distributes further lo cal declarations and triggers if necessary and sequentializes collisions of blocking events over the network.
The scheduler process listens to a broadcast socket (created by NCS library cal ls) for incoming RPC's. The following actions are executed:

### A Process Declares its Interest for an Event

The scheduler inserts the identity of the requesting client process into the INTEREST_LIST. The INTEREST_LIST has the structure showed in figure 4.

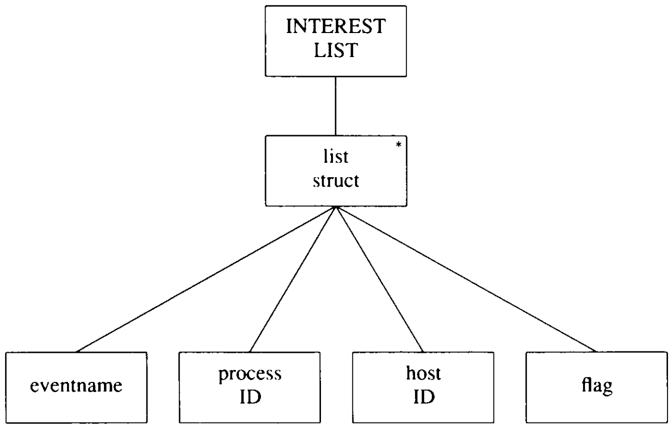


**Figure 4**: *Structure of the INTEREST_LIST*

### A Process Deletes its Interest for an Event

The scheduler deletes the matching entry in the INTEREST_LIST.

### A Process Triggers an Event

As a reaction to an incoming event the scheduler examines the INTEREST_LIST. If it is non-blocking it passes the event to all interested local processes and further broadcasts it to all active schedulers. For the sequentialization of blocking events a token mechanism is used (cp. section 5.4). As soon as the scheduler owns the required token the event is triggered by a point-to-point message to the local interested processes and a broadcast message to all schedulers. The flag in the TOKEN_LIST is set on EXECUTING. Only when all interested processes have terminated their eventhandler-functions (which they can learn about by comparison with the INTEREST_LIST) the flag is changed to NOTEXECUTING.

## 5.4. More Details

### Sequentialization of Blocking Events by Means of a Token Mechanism

For the sequentialization of colliding blocking events the EDM uses a token mechanism. For every event declared as blocking there exists one unique token in the network. A scheduler that wants to trigger a blocking event (as a reaction to a local client message) has to own first the specified token inserted in the local TOKEN_LIST. The TOKEN_LIST contains entries of tokens (eventnames) with an additional flag (EXECUTING / NOTEXECUTING) which shows whether a blocking event had just been triggered, i.e. whether the eventhandler did not terminate until now.

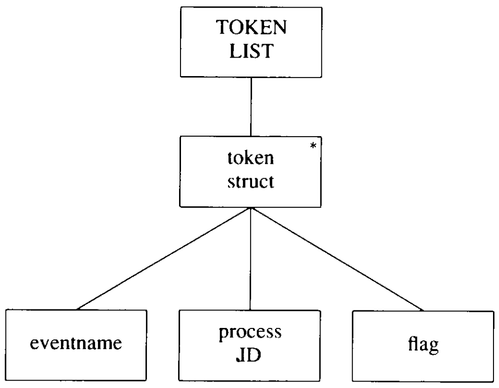


**Figure 5**: *The structure of the TOKEN_LIST*

One scheduler in the network is defined as a master. It is informed about the location of all existing tokens. If a scheduler receives a trigger for a blocking event it firs t has to look at the TOKEN_LIST if it already owns the required token. If it does not own the token, it demands the address of the owning scheduler of the master. The current owner only submits the token if the EXECUTING flag is not set. Otherwise the requesting scheduler has to wait and to retry after some time (cp. next section: Eventqueue). If there is no token for an event the interested scheduler has to produce a new one. Every owner of a newly created token has to transmit its identity immediately to the master.

### Eventqueue

If the same blocking event is triggered simultaneously at various places in the network, the leftover triggered events are entered in a logical FIFO waiting queue. To simplify the mechanism in the current version of the EDM the eventqueue was not implemented physically. Waiting events are passed back to the triggering process (which used *edm_send*). The triggering process retries the request after a fixed time or simply breaks off with an error message.

### EDM Transmitter

A simple way for replicated EDM schedulers to communicate is the usage of RPC's. Therefore the scheduler is able to act as server and client in one. Unfortunately this is not possible at the moment using the NCS. The problem is the following: if a server calls a remote procedure, only the local code of the calling server is executed. As a temporary solution the EDM_transmitter has been introduced. The EDM_transmitter is a process which the scheduler uses as server and client. Every scheduler has a transmitter process which calls the remote procedures in charge of the scheduler. The transmitter and the scheduler communicate over a local socket which is independent of NCS.

## 6. Some Exemplary Application

Because of its synchronization property the EDM is well suited to make a basically unreliable environment as e.g. a network of UNIX workstations more reliable on the application level. The EDM can be used to ensure fault tolerance on the software level e.g. by means of

- implementing distributed locking in order to allow synchronization and serializability.
- distributed atomic transactions in order to allow "all or nothing" operations.
- replicated copies of a file in order to achieve fault tolerance and improved performance by replication.

## 7. Related Work

As far as we know there are no comparable systems which have a really distributed event mechanism. The idea for our proposal originated mainly from the work of another group at our institute which is developing a window toolkit. Window systems use a comparable mechanism for the detection of events caused by usr interaction, see e.g. [Eva86], [Bro87]. Another approach proposed by [Kot88] uses an event/triggering mechanism to check semantic rules in databases, but this mechanism works just locally and does not support the notion of blocking event.

## References

[Bro87] D. J. Brown, J.P. Bowen, "The Event Queue: An Extensible Input System for UNIX Workstations"in *EUUG Spring Conference Proceedings*, 1987.

[Din87] T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato, G. L. Wyant, "The Network Computer Architecture and System: An Environment for Developing Distributed Applications" in *Usenix Summer Conference Proceedings*, Phoenix, 1987.

[Eva86] S. Evans, "The Notifier" in *Usenix Summer Conference Proceedings*, Atlanta, 1986

[Glo89] P. Gloor, *Synchronisation in verteilten Systemen: Problemstellung und Lösungsansätze unter Verwendung von objektorientierten Konzepten*, PhD & Dissertation, Universität Zürich, 1989.

[Glo89b] P. Gloor, R. Marty, M. Zellweger, "An Event Distribution Mechanism (EDM) for Synchronisation in Distributed Systems" submitted to *4th International Conference on Fault-tolerant Computing Systems*, Baden-Baden, September 1989.

[Has88] R. Haskin, Y. Malachi, W. Sawdon, G. Chan, "Recovery Management in Quicksilver" in *ACM Transactions on Computer Systems, Vol.6, No.1*, February 1988.

[Int86] IntelliCorp, "KEE Software Development System Core Reference Manual" *Document 3.0-KCR-1*, IntelliCorp, 1986.

[Kot88] A. M. Kotz, K. R. Dittrich, J.A. Mulle, "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism" in *Proceedings of "Advances in Database Technology - EDBT'88"* Springer, Berlin, 1988.

[Lis87] B. Liskov, D. Curtis, P. Johnson, R. Scheifler, "Implementation of Argus" in *Proceedings of 11th ACM Symposium on Operating System Principles*, 1987.

## Appendix: Manual Pages for EDM Library Calls

NAME

edm_declare, edm_delete, edm_await, edm_send - event distribution mechanism (EDM) operations

SYNOPSIS

```
#include edmdef.h

int edm_declare(eventname, eventhandler, flag)
tEventname eventname;
void (*eventhandler)();
tFlag flag;

int edm_delete(eventname, pid)
tEventname eventname;
tPID pid;

int edm_await(eventname)
tEventname eventname;

int edm_send(eventname, eventarg)
tEventname eventname;
tEventarg eventarg;
```

DESCRIPTION

*edm_declare* allows the declaration of blocking and non-blocking events. It creates an child process which is executing the *eventhandler*. The child process creates a local socket and waits for the declared event. As a reaction it executes the *eventhandler* and reports the beginning and termination to the parent process which is interrupted during this time. The termination of the *eventhandler* is also reported to the local scheduler. The flag can be BLK or NOBLK. Blocking events can only be triggered once at the same time.

*edm_delete* broadcasts *eventname* and *pid* to all schedulers which in turn are deleting the interest entry. The child process created by *edm_declare* is terminated.

*edm_await* lets the calling process wait for the specified event.

*edm_send* triggers an event. All eventhandlers declared by *edm_declare* are executed. Optionally an argument which is used by the eventhandler can be added.

NOTES

The eventhandler routine has to be declared as follows:

```
void eventhandler(arg)
char *arg;
```

Note that on every machine using the EDM library calls an edm_scheduler ( *edmd*) process has to be activated. In addition edm_scheduler uses a transmitter process ( *edm_transmitter*). To run the EDM the NCS (Apollo Computer, OSF standard) has to be installed. On every workstation a local NCS Location Broker has to be activated. Additionally in the whole network one global NCS Location Broker has to run.

RETURN VALUE

Upon successful completion, a value 0 is returned. Otherwise, a value -1 is returned.

# Recent Changes in North American Computer Networks

*John S. Quarterman*
*jsq@longway.tic.com*

Texas Internet Consulting
701 Brazos Suite 500
Austin, TX 78701
U.S.A.
*uunet!longway!jsq*
*+1 512 320 9031*

*ABSTRACT*

This paper gives an overview of developments in computer networks in North America since the publication of "Notable Computer Networks" (NCN) in October 1986 [Quarterman and Hoskins, 1986]. Much of the material was discovered during research for a book [Quarterman, 1989]. Although some of the figures are closely related to ones in the book, none of the text of this paper appears in the book.

## 1. Introduction

There have been many recent developments in networks in the United States, Canada, and Mexico. Some of them have been major changes that have surprised the users of the networks involved. Others have been gradual effects of clear influences or continuations of historical trends. The number and variety of changes (in scale, technology, and politics) have made it difficult to understand what has happened. This is a brief overview that should make the general context clearer.

This paper is presented in four major parts, about:

- changes in certain networks that were described in NCN;
- pervasive influences that have affected many networks;
- some new networks;
- and mergers and extensions of networks.

## 2. Changes in Certain Networks

### 2.1. The Double Death of MAILNET

Although MAILNET officially died in December 1986, as predicted by its administration, the hub machine, MIT-MULTICS, continued to support many links until December 1987, when that machine itself was decommissioned.

### 2.2. The Maturity of CSNET

CSNET now provides many things that were promised earlier. Most noticeably, all hosts on CSNET now use Internet DNS domain names. There are a number of new services, such as Cypress (which uses a protocol of the same name over leased lines), TCP/IP directly over leased lines, and Dial-up IP. The Info-Server allows retrieval of information from several large databases, and the User Name Server provides information on sites and users.

### 2.3. The Weaning of BITNET

Despite the end of funding from IBM in 1987, BITNET has continued to grow (it doubled in size in the two years since October 1986) and has become more robust by assessing fees to its members. The transatlantic link has been stablised by similar fees charged by EARN in Europe, and by a reciprocal arrangement between the two networks. There is no government funding of BITNET.

Some interesting software has been put in place, namely LISTSERV (which allows automatic subscription and deletion from mailing lists) and NETSERV (which allows retrieval of files by mail).

WISCVM, the old gateway between BITNET and Internet, ceased performing that service on 15 December 1987, but INTERBIT (a logical service supported by several gateways, including CUNYVM.CUNY.EDU and hosts at Cornell and MIT) is in place to transfer mail traffic with the Internet. Unfortunately, INTERBIT still doesn't understand MX records. But some BITNET hosts have themselves adopted Internet DNS names.

BITNET II is an experiment in using TCP/IP links as infrastructure for the NJE protocols that underly BITNET, and is partly funded by the U.S. Government. NJE itself has been implemented on a variety of non-IBM machines, and there is an increasing number of such machines on the network.

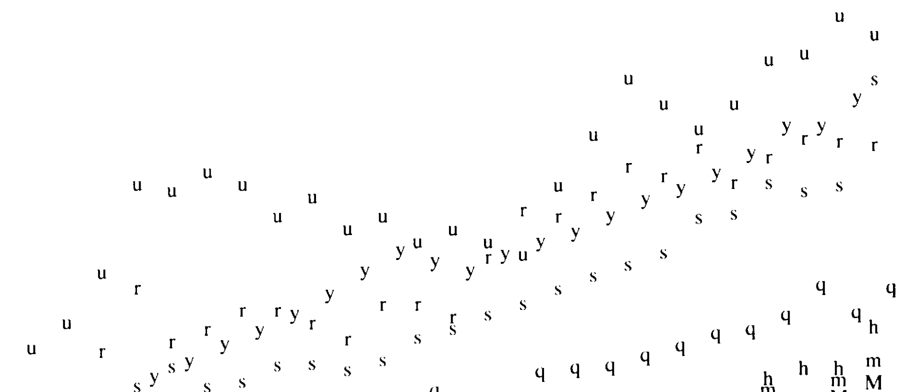
### 2.4. The Distinction of NetNorth

This NJE network has grown much larger, and is very organised, very academic, and very Canadian. Although it is integrated into the same name and address space as BITNET and EARN, it is administratively quite separate.

### 2.5. The Survival of USENET

Despite repeated cries of doom from all sides, USENET still exists and by all measures doubled in size between August 1987 and June 1988.

The development of Lempel-Ziv compression in B News 2.10.2 in 1984 brought a 1.4 times effective improvement in throughput, and the general introduction of 2400bps modems about 1985 gave an additional doubling in effective throughput. The widespread use of Telebit Trailblazers, starting about late 1987, brought effective transfer rates of up to 12000bps, a 3.5 times improvement.



u: 910000.0 users (/2500.0)
r: 143000.0 readers (/500.0)
s: 809.0 sample (/2.5)
h: 4152.0 HC 1 (/25.0)
q: 4838.0 HC 3 (/25.0)
y: 7810.0 HC 12 (/25.0)
M: 130.9 Mbytes (/1.0)
m: 57979.0 messages (/400.0)
n: 381.0 newsgroups (/10.0)

**Figure 1**: *Recent USENET Growth*
[Reid, 1988]

These technological improvements coincided with the wide availability of cheap UNIX boxes due to the introduction of UNIX ports to the Intel 386 in 1987, and with the availability of UUPC, a version of UUCP for MS/DOS, in late 1987. The resulting growth was very noticeable (and paralleled that of EUnet), as shown in Figure 1.

The network also survived the decommissioning of two major backbone hosts, seismo in September 1987, and ihnp4 in September 1988. Of course, the concurrent development of UUNET may have had something to do with this (see below).

## 2.6. The Sprawling of UUCP

The UUCP Project had some success in promoting domain names, but encountered political problems. The network itself continued to grow, with the usual haphazard links.

## 2.7. The Slow Demise of ARPANET

DARPA, which made ARPANET operational in 1969 as the earliest packet switching network of more than one node, decided in 1988 that that network

a) was not research anymore (it was being used as a production network),

b) was not state of the art (56Kbps links and PDP-11 gateways), and

c) was taking an inordinate amount of DARPA's research budget.

Therefore DARPA decided to phase out ARPANET in favor of funding and encouraging other developments such as WIDEBAND, NSFNET, and the Research Internet Backbone. All of the PSNs in the middle of the country were turned off by September 1988, leaving two pieces of ARPANET on each coast. These will eventually be made into separate networks with their own network numbers, and will probably fade into obsolescence. Most of the former ARPANET sites appear to be obtaining NSFNET connections. ARPANET itself will probably still exist through most of 1989. This is the end of the last of the early research networks, such as SERCnet (U.K.), CYCLADES (France), and HMI-NET (Germany).

The major Internet continental backbone networks are now MILNET and the NSFNET backbone, with assistance from WIDEBAND.

## 2.8. The Evangelism of CDNnet

The original developers of the first widespread partial implementation of X.400, the University of British Columbia and later Sydney Development Corporation, continue their work. Their EAN implementation still serves as the basis of X.400 MHS R&D networks around the world, even though it is mutating into something different in many places. CDNnet itself still exists and is larger.

## 3. Pervasive Influences

## 3.1. Speed

Modems that allow data rates faster than 9600bps over ordinary dialup telephone lines are readily available, and 38400bps will be soon. 56Kbps (or 64Kbps) is often found to be too slow for high-traffic dedicated links, and 1.544Mbps (T1) speeds are becoming widely used in networks like NSFNET and even in regional networks like BARRNET (San Francisco Bay Area), Los Nettos (Greater Los Angeles), and NYSERNET (New York State area). Wide area networks using 45Mbps (DS3) are being planned.

10Mbps (Ethernet) local area networks are so common that every large trade show has one. 100Mbps (FDDI) local area network technology is almost ready. 1Gbps technology is in progress.

Existing protocols get pushed to handle higher speeds. 8Mbps on a 10Mbps Ethernet has been achieved with TCP (once thought to be inherently slow) between two Sun-3s. The current TCP speed record is 350Mbps between two Crays over Hyperchannel (hardware limit of 800Mbps) or 550Mbps through software loopback.

Even protocols like UUCP that really are inherently slow get pushed to unanticipated speeds. This is often done by dropping most of the checksumming and packetisation, as in the t protocol for use over TCP (40Kbps over NSFNET is common), and the f protocol for use over X.25. The Telebit Trailblazer spoofs the regular g protocol by returning acknowledgements to packets before sending the data over the wire, so as to build packets large enough for speed; the modem handles inter-modem data consistency by its own means. 12Kbps over dialup connections is commonly acheived by this method.

Workstations commonly have 6MIPS CPUs, instead of the 1MIPS or less of a few years ago, and effective 100MIPS machines can be bought for a few hundred thousand dollars. 100Mbyte disk drives can be bought for a few thousand dollars, and 1Gbyte drives for tens of thousands.

Increasing use of window systems, sophisticated graphics, and data-intensive computations of all types, as well as increased text communications traffic, bring pressure to use these possibilities of increased speed.

### 3.2. Size

Every network listed in NCN that has not died outright has increased markedly in size. This is probably a combination of the effect of the increased availability (cheapness) of computing and communication resources, and certainly of increased demand as more people discover the utility of computer networks.

### 3.3. Commercialisation

In 1986, there were very few private companies specializing in network technology. Now there are half a dozen manufacturers of gateway machines, most of whose products support half a dozen protocols on a single machine. Long distance carriers have proliferated, and some have become very sophisticated in dealing with network technology, as witness the involvement of MCI in NSFNET.

For at least the last five years, industry pundits have declaimed that the TCP/IP protocols would be dead within two years. Nonetheless, they have been widely accepted by the commercial community, and there is hardly a computer manufacturer without an implementation of them. There is a large conference associated with them, the TCP/IP Interoperability Conference (INTEROP), run by Advanced Computing Environments (ACE). Interoperability includes gateways with other protocols such as Appletalk and with machines ranging from PCs and Macintoshes to the largest supercomputers.

Established computer manufacturers go to great lengths to get involved in networking developments. IBM supplies the nodes for the NSFNET backbone, and the NSFNET routing algorithm was developed at IBM Yorktown Heights Research Center. Digitial Equipment Corporation is heavily involved in ISO-OSI protocol development, is already selling an X.400 product, and says that DECNET Phase V will be ISO-OSI compatible. Apollo is offering some of its networking technology to the Open Software Foundation (OSF) for development of their version of UNIX. Sun Microsystems owes a large part of its success to its promulgation of its Network File System (NFS).

### 3.4. ISO-OSI

There is no network in the United States that is primarily based on ISO-OSI protocols. One can attribute this to many causes, among them the existence of working networks based on TCP/IP, DECNET, XNS, NJE, UUCP, and other protocols, so that no immediate need is felt for yet another protocol suite. Also, although there are X.25-based public data networks in the states, they are not nearly as widely used as in Europe or elsewhere.

The situation in Canada is somewhat different, as noted above under CDNnet.

GOSIP is a U.S. Federal Government recommendation to move to ISO-OSI as soon as feasible. This has been resisted by people associated with existing TCP/IP networks, on the grounds that the ISO-OSI protocols are not ready. However, it seems to have led to increased involvement in ISO-OSI development by the Internet and TCP/IP communities. For example, there is ISODE, a development package that allows testing ISO-OSI higher-layer protocols on top of the U.S. DoD Internet Protocol (IP). (ISODE is widely used outside the States as well as within.)

The Corporation for Open Systems (COS) has been established to promote ISO-OSI protocol specification and implementation. There is a large ISO-OSI convention held by Omnicom.

The Computer Systems Research Group (CSRG) of the University of California at Berkeley (UCB) is preparing a complete implementation for UNIX of the OSI-OSI protocols, under contract with the National Institute of Standards and Technology (NIST), formerly the National Bureau of Standards (NBS). It should be remembered that a primary reason for the success of the TCP/IP protocols was the implementation of them done by CSRG in 4.2BSD and 4.3BSD.

## 4. New Networks

### 4.1. DASnet

DASnet is named for DA Systems of Campbell, California, which established it in July 1987 and continue to own and operate it. It is a mail forwarding system specialising in transferring mail between commercial systems such as CompuServe, MCI Mail, the Well, Portal, and EIES. More traditional services such as TELEX and paper mail are supported indirectly. DASnet has some overseas customers such as TWICS in Tokyo. There are connections to UUCP, and indirectly to BITNET. DASnet has two Internet DNS domains registered, DAS.NET and DAS.COM. It relays mail with the Internet, although there is some contention about this.

There are two major reasons DASnet is significant:

- It connects commercial systems with one another; most of these did not formerly communicate.
- It connects commercial systems and non-commercial systems; this has never been done before on the scale that DASnet does it.

Some the larger commercial services, such as GENIE, have refused to connect, however.

### 4.2. UUNET

UUNET is a mail and news forwarding non-profit but charging service whose subscribers connect using UUCP, and which has an official Internet connection and permission to gateway among those networks. The essential features of the service are:

Anyone can connect

It has become increasingly difficult to get a news feed from a traditional USENET backbone host, because they are all saturated. Since UUNET can increase capacity with increased service, it does not have this problem.

Approved Internet gatewaying

Initially DARPA, and later also NSF, approved connection of UUNET to the Internet and gatewaying of traffic between the Internet and the UUCP network. The current connection is to SURANET, an NSFNET regional network.

Reasonable Rates

by a variety of connection mechanisms.

International Connections

UUNET may connect to more countries than CSNET, and probably carries more international traffic. UUNET is the primary UUCP mail gateway to Europe, Australia, and Japan, as well as to Chile, Thailand, and numerous other places.

UUNET has also absorbed the UUCP project.

UUNET started as an experiment proposed by Rick Adams, who had been running seismo, a major USENET and UUCP backbone host, for several years, and who was the author of B news 2.11, the current version of the USENET news software. The UUNET experiment was sponsored by USENIX, whose board of directors had been requested by its members to attempt to improve the level of service provided by UUCP and USENET. It was operational in May 1987, using a 10 32016 CPU Sequent Balance 21000 loaned by Sequent Computer Corporation of Beaverton, Oregon. In less than two years, its annual income was projected to be greater than that of the parent organisation, and it had 450 subscribers by October 1988, after almost linear growth. The machine had been bought by then, with a loan guaranteed by USENIX, and upgraded to be a Sequent S81 with three Intel i286 CPUs.

UUNET Communications Service incorporated in the State of Delaware in July 1988, although it had essentially no assets at the time, since the actual service was still owned by USENIX. However, the USENIX Board of Directors had already expressed a desire to make UUNET a separate company at its meeting in June 1988, and details regarding transfer of funds and authority were arranged by the end of 1988. USENIX retains one third of the seats on UUNET's Board of Directors, partly due to a desire by the USENIX Board of Directors to remain fiscally responsible about the loan for the machine, and partly to retain the influence over the USENET and UUCP networks that was its original goal.

The USENET backbone was declared dead in September 1988 by the maintainer of the backbone mailing list, in the belief that the huge amount of traffic carried by NSFNET, the introduction of Telebit Trailblazers, and the existence of UUNET obviated any need for a backbone. We shall see if the backbone stays dead.

## 4.3. NSFNET

NSFNET was barely started when NCN was published in October 1986. It has since been through two complete backbone organisations (the original 56Kbps and fuzzball backbone, and the current T1 backbone) and has grown phenomenally.

| | |
|---|---|
| BARRNET | San Francisco Bay Area Regional Research Network (2) |
| JVNCNET | John von Neumann Center network (1) |
| Merit | Merit Computer Network (2) |
| MIDnet | Midwest Network (2) |
| NCSAnet | National Center for Supercomputing Applications network (1) |
| NorthWestNet | Northwestern States Network (2) |
| NYSERNet | New York State Educational and Research Network (2) |
| PSCnet | Pittsburgh Supercomputer Center network (1) |
| SDSCNET | San Diego Supercomputer Center network (1) |
| Sesquinet | Texas Sesquicentennial Network (2) |
| SURAnet | Southeastern Universities Research Association Network (2) |
| THEnet | Texas Higher Education Network (2) |
| USAN | University Satellite Network (3) |
| WESTNET | Mountain States Network (2) |

(1) supercomputer consortium network
(2) regional network
(3) discipline-oriented network

**Table 1**: *NSFNET Mid-Level Networks*

NSFNET is organised in three tiers:

The Backbone
: The continental T1 links that connect mid-level networks.

Mid-level Networks
: These are of three kinds: regional; discipline-based; and supercomputer consortium networks. These are listed in Table 1. In addition, CSNET has been classed as an NSFNET mid-level network since mid-1988.

Campus networks
: connected to mid-level networks. These can be either academic or commercial.

The regionals are supposed to become self-funding within three years of their founding, and some are near that goal.

Interconnection of networks on this scale has led to much work on internetwork management protocols and implementations, such as SNMP, which is in use on NSFNET, and CMIP, which is the ISO followon that is expected to replace SNMP in a few years.

## 4.4. NRCnet

This is a TCP/IP network being established by the National Research Council of Canada, and modeled directly on NSFNET. It is to provide a continental backbone to connect regional networks such as UBCnet in British Columbia and ONET in Ontario. CDNnet and NetNorth will probably use it as infrastructure, as well. There is already a T1 link to the NSFNET backbone, from UBC to the University of Washington in Seattle. Another link, to the University of Toronto, should be in place by 1989.

### 4.5. Mexican Networks

The X.25 PDN TELEPAC is completely saturated.

The satellite network Morelos uses the Ku and C bands, is about 20% used, of which traffic about 60% is television, 30% is voice, and 10% is data. It was installed by NEC.

There are commercial networks run by PEMEX (the government petroleum monopoly), banks, and others. IBM's company network VNET extends into the country.

The Universidad Nacional Autonomidad de Mexico (UNAM) runs an academic network based on TELEPAC links.

The Instituto Technologico del Estado de Mexico (ITESM) has a large academic network based around Mexico City and Monterrey (the two largest cities). It is based on satellite links connected at the data link level with Vitalink bridges to form a single very large Ethernet, on top of which TCP/IP and other protocols are used, through cisco gateways. This is rather like NORDUnet.

There is a BITNET connection through the University of Texas at San Antonio, but still no UUCP connection.

Two connections to NSFNET have been requested, both to UCAR in Boulder, Colorado; a 56Kbps one to Mexico, and a 9600bps one to Monterrey.

## 5. Mergers and Extensions

### 5.1. CSNET and BITNET

There is continuing talk of a merger between CSNET and BITNET, although not much has come of it. The incentive is that their clients tend to be similar, if viewed at a high enough level. E.g., they both go to many universities and colleges, and thus such schools are paying twice for essentially similar services. However, they tend to go to different parts of the same schools: BITNET to the computer center; and CSNET to the computer science and other departments. Thus the actual clienteles are not that similar.

### 5.2. ESnet (MFENET and HEPNET)

MFEnet and HEPNET have both grown quite a bit. HEPNET, in conjunction with SPAN, maintains a worldwide DECNET address space and interconnections. ESnet (Engineering Sciences Network) is a planned replacement for MFEnet and HEPNET in the U.S., for all energy research programs funded by the Department of Energy. An initial backbone was in place during 1988, connecting eight sites with 56Kbps links. Five more sites are planned, as are T1 links. Protocols supported include IP, X.25, DECNET, and the MFENET NSP protocols.

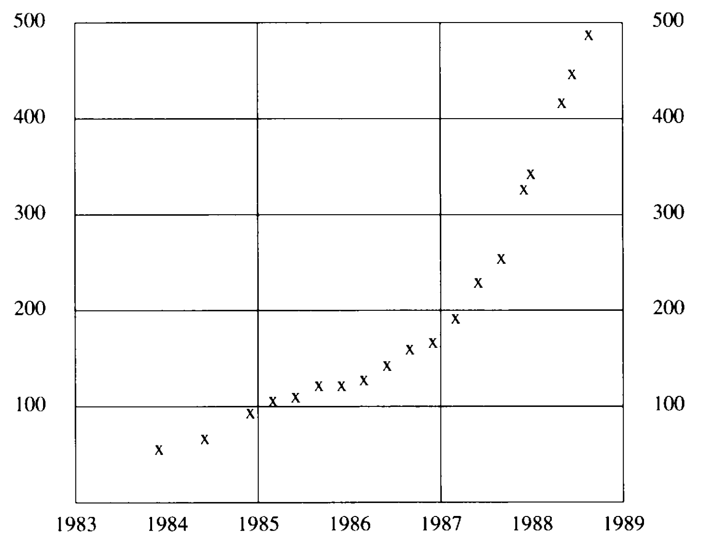


**Table 2**: *Internet Growth (1983-1988)*
Actual Connected Networks
[Brescia, 1988]

## 5.3. Internet Growth

In the two years since October 1986, the number of networks actually connected to the Internet (i.e., in the active core gateway tables) increased from about 120 to about 500, as shown in Table 2. Most of this growth occurred in 1987 and 1988. About half of that can be attributed to the expansion of NSFNET. The rest is presumably due to campus networks and networks at private companies.

## 5.4. Internet and NSFNET Foreign Connections

ARPANET no longer has any foreign hosts, and SATNET that used to connect it to Norway is no more. But MILNET currently extends to the Philippines, Korea, Japan, England, Scotland, Germany, Belgium, Spain, Italy, Greece, and Turkey. DREnet in Canada has long been connected to the Internet. However, most of these connections are not new.

But NSFNET currently has foreign links to Canada and a request from Mexico (as noted above). There has been a working link from John von Neumann supercomputer Center (JVNC) in Princeton, New Jersey to (INRIA), near Nice, France, since September 1988. There are requests for connections to Sweden from NORDUNET for NORDUnet and from University College London (UCL) for JANET. There are other requests for connection from Europe (perhaps more can be said in April). A connection to Tokyo is likely. Plans for Australia and New Zealand are rumored.

The HP Internet, a TCP/IP network that is integrated into the Internet name and address spaces, extends to Europe and Australia. Other company TCP/IP networks may, as well.

If the Internet is not already, it will probably soon be the largest worldwide network.

## 5.5. FRICC

The Federal Research Internet Coordinating Committee (FRICC) was formed in late 1987 (or early 1988, depending on which of three dates you believe), as a result of a report published in November 1987 by the Computer Research and Applications Federal Coordinating Council for Science, Engineering, and Technology (FCCSET, pronounced "fix-it") Committee of the Office of Science and Technology Policy of the Executive Office of the President of the United States. That report said that the U.S. was lagging behind other countries in establishing a national research network, and proposed building one, based upon the work already done by DARPA, i.e., the Internet. Link speeds and sites and hosts connected should be increased, with the result of a general national research network with 3Gbps speeds by 2003.

FRICC was formed to somewhat formalise already existing cooperation among several government agencies, and to work on building a network similar to that proposed by the FCCSET report, starting with a Research Internet Backbone of 45Mbps (DS3) links. The agencies involved are the Department of Defense (DoD, i.e., DARPA), the National Aeronautics and Space Administration (NASA), the National Science Foundation (NSF), the Department of Health and Human Services (HHS), and the Department of Energy (DoE). The chair is William Bostwick of DoE.

Another reason for the formation of FRICC was to have a body to coordinate with RARE (Re´seaux Associe`s pour la Recherche Europe´enne), the corresponding European body.

Unlike RARE, however, FRICC has no means of incorporating non-governmental bodies, and thus CSNET, BITNET, UUCP, and USENET are not represented.

## 5.6. CCRN

The Coordinating Council on Research Networks (CCRN) has two co-chairs, William Bostwick, the chair of FRICC, and James Hutton, the Executive Director of RARE. It serves to coordinate European and American networking developments.

## 6. Some Summary Comments

The last two years have seen great growth in speed, hosts, sites, and technological developments. Although a few networks have dropped by the wayside, there are more now than there were then. Nonetheless, there is a strong current movement towards consolidation of similar efforts into single networks. Interconnections among networks in North America and with other networks elsewhere continue to increase.

## References

Brescia, 1988.
Brescia, Mike, ''Internet Connected Networks,'' *personal communications*, ARPA Internet, Cambridge, MA, September 1988.

Quarterman, 1989.
Quarterman, John S., *The Matrix: Computer Networks and Conferencing Systems Worldwide*, Digital Press, Bedford, MA, 1989.

Quarterman and Hoskins, 1986.
Quarterman, John S. and Hoskins, Josiah C., ''Notable Computer Networks,'' *Communications of the ACM*, vol. 29, no. 10, pp. 932-971, Association for Computing Machinery, New York, New York, October 1986.

Reid, 1988.
Reid, Brian, ''network maps,'' *DECWRL netmap 1.5*, DEC Western Research Lab, USENIX, USENET, Palo Alto, 19 June 1988.

# The EDUNET Project

*Manfred Wöhrl*

Higher Level Secondary School
Spengergasse 20
Vienna
Austria

## 1. Introduction

In 1981 the first special type of school for computer science, corresponding to the level of a university, was founded in Austria. The main goal of this type of school is to give a practical education on most of the areas of computer science within 5 years. Some theoretical background is taught, but less than at universities. On the other hand more practical lessons are required. In fact at the moment each student attending this type of school has more access to computers than many students at universities.

## 2. Requirements

In Computer Science the life cycle of hardware and software has been reduced therefore it is very important to have a modular system. This is the only way of being able to expand, if necessary, or to change old machines to newer ones. Therefore we decided on:

- UNIX
- NETWORKS

Both have the opportunity to present standards, especially for LANs, when using Ethernet on the first two layers of the ISO/OSI- model.

A very important aspect is the security of the system. First of all the system itself should have a large MTBF (Mean Time Between Failure) and on the other hand the operating system must offer a lot of facilities to control all user activities. Especially file access and the use of all networking features must be secured against hacking and other bad ideas that students have.

## 3. Realisation

It was important to get a machine, using a standard CPU and if possible a standard network. The students should learn practical applications installed on many other locations in industry. Therefore we decided to buy a 68020 based computer from NCR (type: TOWER-32/600) using Ethernet (IEEE 802.3) as the network standard. A large disk (about 400 MB) was necessary to teach a lot of different subjects, such as programming in different languages, database instructions and some networking applications. For the same reasons, we decided to implement standards on hardware and installed ORACLE as one of the most "portable" database system.

### 3.1. LAN

Several PCs from different vendors have been installed for teaching within the last few years. Some others were added for the teachers to prepare their lessons. One "PC laboratory" with 10 PCs was installed. Each PC supported an Ethernet interface card (3-COM card) and connected to the TOWER system, where TOWERnet is running. All PCs use PC-connect and have two floppy disk drives, but no hard disk. Using the virtual disk support of TOWERnet this brought a lot of benefits as mentioned below.

### 3.1.1. Ethernet

#### 3.1.1.1. TCP/IP

TOWER systems work on TCP/IP using EXOS 8012-01 from Exelan Inc. This software implements the DARPA protocol standards, which is necessary to connect systems of different vendors such as NCR and HP. In this way we added an HP-9000 graphic workstation to the network.

#### 3.1.1.2. TOWERnet/PC-connect

Using many standalone personal computers, produces many problems: the distribution of new software releases running on all PCs takes a lot of time, to make backups of all hard disks takes also a long time. One solution of these problems is the installation of a PC network. A special PC with a large disk can serve all connected PCs.

Summary of PC commands:

TSLOGIN user password
: Inform the PC, that you are a authorized user of the UNIX server installed at the network.

  Example: `TSLOGIN woehrl quecoa99`

TSMAILCHK system
: Checks, if a mail is available for a PC user (TSLOGIN is necessary before)

  Example: `TSMAILCHK node01`

RLOGIN system
: Start a terminal emulation on a remote UNIX system running TOWERnet

  Example: `RLOGIN node01`

TSCREATE system filename [size]
: Create a virtual disk file on the server with the given filename and an optional size ranging from 1 MB to 31 MB in steps of 1.

  Example: `node01 /usr/acct/working.dsk 10`

TSMODIFY system filename -private
: Change the rights of access of a virtual disk file to read and write, but only for the owner of the file.

  Example: `TSMODIFY node01 /usr/acct/working.dsk -private`

TSMODIFY system filename -public
: Change the rights of access of a virtual disk file to read and write for all users of the TOWER system, but only for one user at a given point of time.

  Example: `TSMODIFY node01 /usr/acct/working.dsk -public`

TSMODIFY system filename -shared
: Change the rights of access of a virtual disk file to read only, all users can access the file simultaneously

  Example: `TSMODIFY node01 /usr/acct/working.dsk -public`

TSASSIGN drive: system filename [-r]
: Assign the virtual disk file of the TOWER system and make it available for the PC as the named drive. For option a write protection switch can be used.

  Example: `TSASSIGN e: node01 /usr/acct/working.dsk`

TSASSIGN
: Show all assigned virtual disk files and their drive letter, the virtual disk can be used with.

TSDEASSIGN drive:
: Make a used drive letter free for another assignment on the used PC

  Example: `TSDEASSIGN e:`

TSDELETE system filename
: Delete a virtual disk file on the TOWER system

  Example: `TSDELETE node01 /usr/acct/working.dsk`

SEND system [-d dir] [-o user] [-p] [-r] filename(s)
: Send one file (or more, using wildcards) from a PC to the TOWER system with the following options:

| | |
|---|---|
| dir | directory of the target system, where the file is placed |
| user | user of the TOWER who is the owner of the file |
| p | forces the creation date to be transferred together with the file |
| r | if "file" is a directory, all files of this directory will be transmitted. |

  Example: `SEND node01 -d /usr/acct/pclab *.dat`

RECV system [-d dir] [-o user] [-p] [-r] filename(s)
: Get a file from a TOWER system and store it on the local PC.

  The options are of the same meaning as above. If necessary, a password is requested. (depending on the rights of access and the "TSLOGIN -name " entered before)

  Example: `RECV node01 -d /etc -o root passwd`

RX system [-d dir] [-o user] command
: Force the Tower system to execute a command initiated from the PC.

  Example: `RX node01 -d /usr/acct/userx prog42`

  Note: If no login is allowed, using a standard terminal, RX is the only possibility, to work on a UNIX system. (it can happen, if the execute permission or the ownership of the login- program has been lost)

Many of the commands also can be used from a terminal of the UNIX system. In this case you had to type "server" on a PC node before and up to now, for instance, this PC can be attached with an RX command.

Example: `rx ncrpc dir \ws`
: This will show a directory of the subdirectory "ws" of the Pc with the node name "ncrpc" on the terminal, where the command was typed.

### 3.1.2. KERMIT

This public domain product is used to connect Personal Computers which have their own hard disk. In this case a data transfer rate of 9600 Baud will be enough. Normally these PCs are working as terminals on the TOWER.

KERMIT is also used to login from a remote PC using one of the two MODEM lines installed. These serial login lines operate either with 300 or 1200 Baud.

## 3.2. X.25

The public network is used to connect the TOWER to several machines in the university. For training it is possible to make small database retrievals.

### 3.3. SNA

Using the System Network Architecture we are just installing a link to the mainframe of the Ministry of Education. This is necessary to teach the features of large computer systems. Within the last few years this teaching took place from 25 terminals connected to this mainframe. SNA offers the possibility to do some testing on the TOWER and submit the jobs to the HOST. In this case the mainframe has less activity during daytime and night queues can be used more effectively.

## 4. Problems

### 4.1. Hardware

The most difficulties resulted from the installation of the cables. Having used a thick (standard) Ethernet cable as well as a thinwire Ethernet, we got a lot of lost connections, especially at the connecting points of the PCs. In this case the network software reported: "Please check proper termination" and the Ethernet didn't work any more.

### 4.2. Software

Following theoretical articles and definitions Ethernet allows several protocols to run simultaneously. In fact the installation of TOWERnet and TCP/IP requires a relink of the kernel of UNIX. The relink fails with a duplicate global entry error. Therefore in our network configuration we can only run either TOWERnet or TCP/IP, changing means rebooting the TOWER system using another /unix as kernel.

## 5. Solutions

At the moment we are using TOWERnet because its feature of virtual disk support. The graphic workstation (HP-9000) is connect using two serial lines and UUCP. Within the next few months we will change all applications to TCP/IP and install virtual disk support from SUN on all PCs, which costs a lot of money. But this is the only way to avoid rebooting the TOWER system and have a better Ethernet solution.

## 6. Further Aspects

### 6.1. LAN Repeater

In 1989 a second segment of Ethernet will be installed. This expansion will be used to connect the office, the bookkeeping department and the library to the TOWER system. A mail system is planned as well as some ORACLE applications, especially for the library. These software products will be developed by the students as preparation for their final examinations.

### 6.2. Multi-server

For several reasons it is necessary to expand the network by installation of further UNIX servers. First for safety and second for real distributed data processing. One server can operate as database machine, another server can be used for program development and a third one as communication server. Using Ethernet, remote login is possible all over the network, from personal computers as well as from each connected terminal.

### 6.3. Terminal multiplexing

Using special hardware, eight or more terminals can access the network without installation of a new computer system. From each workstation you can attach all UNIX servers, connected to Ethernet.

# Extending User Interface Toolkits for Picture Processing

*Dr S. T. Jones*
*Mr William L. Franklin*

Value Added Networks Lab
Department of Electronic Systems Engineering
University of Essex
Wivenhoe Park, Colchester
Essex CO4 3SQ
England

*ABSTRACT*

The name toolkit is a good description of both its purpose and what it provides. The toolkit builder provides the objects or widgets or tool-pieces to fit the types of interfaces that the toolkit builder imagines the toolkit will be used to build. Thus not all applications can be served by a given toolkit. As an analogy, a mechanic's, or electrician's toolkit provides the tools for their trades but could not reasonably be used to build a dining room table, at least not a table that would resemble that which could be made using a cabinet maker's toolkit.

User interface toolkits exist for most of the available windowing systems, e.g. X Window System, Andrew, etc. The contents of these toolkits do not always provide suitable tools for all applications. In the first part of this paper we examine the contents of the toolkits distributed with the X Window System for use in building a user interface to a Picture Editing system. In the second part of this paper we describe the development of tools for these operations and how they can be incorporated within the existing toolkits for use in future application.

The user interface requirements of a picture editing system are described, these are compared with the facilities provided by the main toolkit distributed with X/11 release 3. This provides the basic tools but does not include tools to support picture operations like region defines that require mouse input. The implementation of a track widget to provide this function for the X toolkit is described.

## 1. Introduction

In recent years we have seen an ever increasing introduction of window systems. The early window systems provided little in terms of support for reusing the interface code which had to have been written for the window manager. This meant that the design and implementation of the interface was usually done at the lowest possible level rather than using the existing interface code and mechanisms built into the window manager, the window system's clients and tools. This was somewhat of a waste as a majority of user interface code is, and should be, reusable. Also because a set of common applications should look and act the same for all of the tools in the set.

Designing from the lowest level, from scratch, each time allowed for very little shared functionality, much less, reusable code. Using this low-level method, each implementation could, and usually does, act subtly different in the manner in which user interaction is managed. [O'87a] As window systems have matured issues such as code reusability, consistent look and feel, and machine portability have became hot topics for not only engineering and marketing departments, but also legal departments. As this low-level approach does not provide much support in the way of reusability, consistency and, certainly not, portability other methods have been proposed and put forth.

Many of the window systems, in an attempt to alleviate some of the problems associated with these issues of reusability, look and feel, and portability now have companion toolkits which are layered on top of them. These generalized toolkits, combining basic building blocks and mortar, are meant to be used in the design and implementation of applications to be run under the toolkit's window system. This may solve some of the problems pertaining to reusability and common look and feel under the same window manager, but what if the toolkit is not rich enough in building blocks to design a specific interface? Generality is fine, but problems are often not general. Or better yet — that solutions, rather than the problems, are often not general enough.

What is needed is a simple and effective way to extend the base level of tools. More of a toolkit for toolkit builders, as opposed to a definitive toolkit. Extensible toolkits provide just these types of features.

Our specific need to extend the X toolkit, [Ath87a] by adding more building blocks, came from a research project on user interfaces for picture processing systems. We feel, however, that many of the problems encountered by us are also being encountered by other groups doing work not only in picture processing, but in all forms of multi-media applications.

## 2. The Value Added Networks Laboratory

The Value Added Networks Laboratory (VAN-Lab) at the University of Essex is examining value added systems, both hardware and software, for network systems. These additional network components include such things as Distributed GKS, [Siu88a] Intelligent Cards, [Jon89a] user interfaces using Open Dialogue [Fra88a] and GEd(see section 3).

The VAN-Lab is staffed with three lecturers, one research fellow and a group of post-graduate students. Available hardware resources include various UNIX based workstations, many with photovideotex hardware, a DEC VAX 11/750 and a 11/730 both running 4.3BSD, a laser printer, various color monitors and cameras, and some IBM PC/AT computers.

Much of the motivation of this particular VAN project has been to modularize GEd, a photovideotex editor originally written for the IBM PC/AT, into small component modules. It is the intent of the VAN-Lab to use GEd as a base software system for testing the ease and flexibility of building user interfaces from user interface toolkits.

## 3. GEd – a *G*raphics *Ed*itor

In order to more fully exercise the various strengths and weaknesses of a user interface toolkit and design system, a feature rich interactive application is needed to use as an application base. For the purpose, GEd [Wea88a] is an ideal choice.

GEd is a photovideotex [Gec83a] editor which supports image capture, editing and storage on to disk. The editing is done in an interactive environment much as one would edit a text file. For the purpose of this research it was not crucial that all functions of GEd be accessible from the interface. Only a small portion of the total functionality of GEd was needed for it to provide enough diverse interaction to exercise all of the user interface issues the research was to examine.

### 3.1. GEd Functions

GEd supports a large variety of editing functions for video images. Many of these functions have not been ported to the UNIX platform as they did not add any different user interface requirements than those which had already been ported. This is not to say that much of GEd lays un-ported, for that is not the case, but the point of the research was not to examine the difficulties in changing bit widths in architecture platforms and porting photovideotex editors, but in evaluating user interface toolkits. Hence only the parts of GEd that were needed to fulfill the diverse requirements to investigate the toolkits were ported. These ported functions include cutting out objects, copying and painting with the various types of cutouts, modifying the cutouts in various manners with supplied filters and functions, capturing images, placing cutouts and inputting text. Figure 1 provides a hierarchical overview of the operations available.

GEd serves as an acceptable application to test user interface design tools due to a few reasons. Obviously it is interactive. Second, it takes both keyboard and mouse device input. Third, it is written in a high level language so modern software development tools can be used in the debugging of not only the interface, but also the application. Fourth, because it is a picture editor it provides some amount of user feedback automatically as the picture is modified. Fifth it requires the user's attention focus on both the workstation monitor and the video display monitor. So care must be taken to focus the user's attention at his previous task on the workstation monitor.

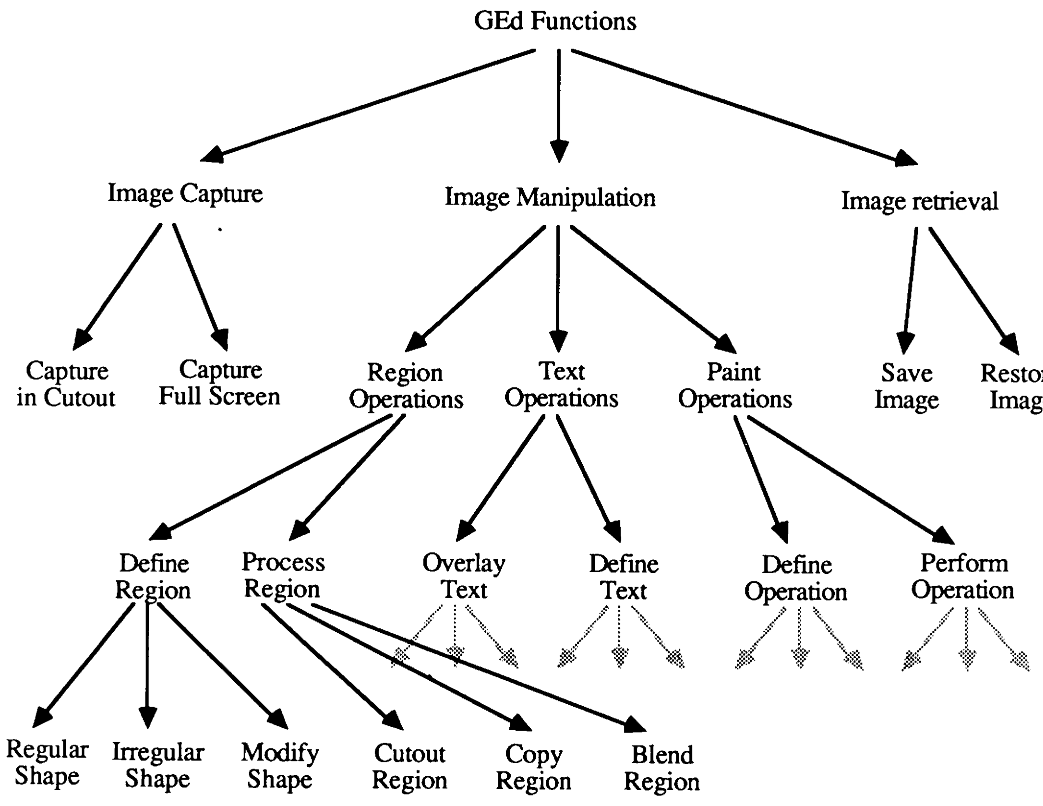


**Figure 1**: *GEd Operations*

Finally, the two most obvious requirements, the application had to be large enough to supply the needed functionality but small enough so as not to require more than one person to port it, and the application had to be available to the lab.

## 3.2. GEd Summary

GEd is large and complex enough to have all of the requirements for an application base on which to build user interfaces. GEd was available to the authors and was small enough to be single-handedly ported to the UNIX platform. GEd is written in a high level language. GEd provides some user feedback and GEd uses more than one display requiring user focus to continually change from video monitor to workspace and back again.

## 4. Window Based Application's Programmers' Tools

The tools available to support the development of window based applications can be divided into User Interface Toolkits and User Interface Prototyping Tools.

## 4.1. User Interface Toolkits

The name toolkit is an accurate analogy both to the purpose an interface toolkit supports and the tools it contains. Much as an automobile mechanic's toolkit contains the tools to work with engines and would be useless for building a dining room table, a user interface toolkit contains specific tools to build interfaces for a predetermined or visualized set of applications. Thus the designer of the toolkit selects the objects or widgets or tool-pieces to fit the types of problems, or interfaces which the designer imagined the toolkit would be used to solve. Hence it is clear that not all applications can be served by a given toolkit. The obvious corollary is that not all toolkits can be used to develop similar interfaces for a given application – just like a plumber's, or an automobile mechanic's, or an electrician's toolkit cannot be used to build a dining room table(at least not a table that would resemble what could be made using a cabinet maker's toolkit).

Ideally toolkit technology should contribute to good user interface and application design. Also, there should be no need to recompile the application when the user interface changes. A toolkit supporting these requirements would a enable a person with expertise in human factors, graphic arts or related disciplines to design and implement the interface. The ease of such a design process would encourage iterative design techniques and interface tuning. Design economies would encourage the provision of multiple interfaces to match the needs of both the advanced user and the novice. [Tei84a]

## 4.2. User Interface Prototyping Tools

Prototyping tools may provide very much the same interface component building blocks as toolkits, but are distinguished by providing explicit support for rapid development and prototyping of the window-based user interface. Thus, prototyping tools enable interactive evaluation of the user interface at a stage when the underlying application software may not be available.

Through the development of feature-rich prototyping tools, developers can develop and refine their designs for advanced user interfaces. More importantly the tools provide a development platform which ensures a consistent look and feel(LAF) to all applications. With the advent of portable windowing systems such as X [Sch86a] and Andrew [Mor86a] and toolkits built on top of them such as the X Toolkit, Open Dialogue and the Andrew Toolkit, it is now possible to develop software systems that provide a consistent LAF across a wide range of computer environments.

## 5. The X Window Toolkit and The Athena Widget Set

## 5.1. Introduction

> "The X Toolkit provides the base functionality necessary to build a variety of application environments. The X Toolkit is extensible and supportive of the independent development of new or extended components. ...
>
> The X Toolkit is a library package layered on top of the X Window System. This layer extends the basic abstractions provided by X and, thus, provides the next layer of functionality. ..." [Swi88a]

The X Toolkit consists of two libraries: Xaw(the Athena Widget library) provides a set of user interface objects(widgets) for building application interfaces, and Xt(the Tool intrinsics) provides the necessary support facilities to build, combine and operate the widgets. The widgets form the basic building blocks for interface design. The user interface is therefore defined as a tree of widgets. Hierarchical groups of widgets(composite widgets), form the nodes of the tree. The final child widgets(primitive widgets), determine the actual interface to the application and the display of application state.

## 5.2. Application Interface

The links between the application code and the user interface code are defined when the widgets are created, by binding callbacks and data structures to them. A callback is an application routine that is called by a widget in response to an event such as a button press. The callback routine then initiates the application's response to the user's action, for example display more data, exit etc. The data structures allow the displayed state of the interface to be controlled, e.g. change label, colour etc.

The structure of the interface can be changed by creating and deleting widgets during the operation of the application, e.g. a new command widgets can be added to a menu. This facility allows the interface to dynamically change to reflect the status of the application and help the user with the task.

## 5.3. The Basic Widgets

The widget set distributed with X Version 11, Release 3 contains a selection a widgets that be grouped according to the functions.

### 5.3.1. Display Widgets

Display widgets provide for the display of information to the use they include; Label Widget provides an uneditable text string that is displayed within a window, Text widget provides a way for an application to display one or more lines of text.

### 5.3.2. Interaction Widgets

Interaction widgets provide for user interaction, they include; Command button widget creates a rectangular window that contains a text label. When the pointer cursor is on the button, its border is highlighted to indicate that the button is available for selection. Then, when a pointer button is clicked, the button is selected, and the application's callback routine is invoked. The Scrollbar widget is a rectangular box that contains a slide region and a thumb(slide bar). The Viewport widget consists of a frame window, one or two Scrollbars, and an inner window. The inner window is the full size of the data that is to be displayed and is clipped by the frame window. The Dialog widget implements a commonly used interaction semantic to prompt for auxiliary input from a user. The Grip widget provides a small region in which pointer events are handled.

### 5.3.3. Management Widgets

Management widgets are the tools required to manage the presentation of the other widgets on the screen for operations such as creation, exposure, resize etc. They include The Box widget provides geometry management of arbitrary widgets in a box of a specified dimension. The VPaned widget manages children in a vertically tiled fashion. The Form widget provides geometry management for its child widgets, based upon information provided when a child is added to the Form.

## 5.4. Extending the Widget Set

The set of widgets that is included in the core of the third release of X/11 may appear rather limited but is extended by additional widgets that are included in the user contributed software. Nevertheless complex interfaces can readily be built as the widgets available provide most of the fundamental interface components that are required.

The structure of the widget set has been designed so that it is simple to introduce new widgets providing different or extended functions. In most cases an interface requirement can be met by extending one of the existing widget classes. This is a moderate task for a competent programmer as the X windows distribution includes the source code for all of the system.

Extensibility is facilitated by the toolkit supporting a single-inheritance class hierarchy. [McC88a] This allows a new widget to inherit all operations from its superclasses and therefore has only to implement any new features or changes. New widgets are thus generated by examining the existing widgets to identify the one that most closely approximates the features required and then copying and extending its code to include the required new features.

## 6. A Model For Application's Interface Interaction

All windowing systems present the user with a graphical representation of the interface to the application. This interface is normally abstracted from the application and hides the details of the application's internal operation from the user.

From the programmers viewpoint the divisions within the application are important. For most windowing systems this can be generalized into application and interface code. The application code describes the specific operation of the application. The interface code describes the interface to the user and controls the operation of the interface devices, typically screen, keyboard and mouse.

The boundary between the application and interface code varies between windowing systems. For example, in the X window system the division between interface and input/output device control code is explicitly made. The input/output devices are controlled by the display server process and the interface defined by library code linked into the application process.

## 6.1. Direction of Control Flow; Abstraction of Command

In the application development process the programmer must make the distinction between interface operations and applications operations within the program and define the interaction between the interface and the application. A fundamental design decision that has to be made at this stage is the source of control. Does the interface drive the application or does the application drive the interface? Traditionally applications drove the interface, that is they had full control of the information presented to the user and prompted the user for input when required. In programming terms the thread of control remains within the application. Most user interface tools however,are designed to control the application, the interface making callbacks to the application code in response to users actions. The interface tool treats the application as a library of functions. In programming terms the thread of control belongs to the interface and is passed to the application, as a function call, to fulfill a user's operation request.

If the interface is to drive the application then the application should be configured as a set of functional routines, with each routine being activated in response to a user action. The application's state should be updated by the routines before returning to the interface code. If the application is to drive the interface, some mechanism must be provided to pass control from the interface code into the application. This can be achieved by an initial input from the user. Once the application has control, the interface can be manipulated by changing the data structures for the interface, which is equivalent in the X toolkit to mapping and unmapping widgets. Control is only passed back to the interface when user interaction is required.

These rules cannot be enforced for all applications. For example an interactive printer queue monitor needs the application to update the information displayed regularly but simultaneously should allow user input at any time. Solutions to this problem are not explicitly supported in the X toolkit. With the X toolkit, one approach to this problem would be for the application to regularly use the low level X interface, Xlib, to test for events and then return to the interface code to process them.

### 6.2. Interface Abstraction Model for GEd

By viewing GEd as a series of task libraries being called from an interface driving mechanism certain design criteria are established. Much of the original function of GEd was user interaction. All of this had to be moved to the interface component of the X toolkit version of GEd. As GEd was originally written for the IBM PC/AT much of the application had interface code, or more specifically mouse code, sprinkled throughout. All of this had to be removed.

One of the observations which was made during this time was that applications written under the older style of the application driving the interface could be very difficult to re-model to the newer style of interface driving the application. The level of difficulty is obviously somewhat dependent on the level of structured programming techniques employed. But even extremely well structured programs can present problems due to the shift in control for application functionality to interface. Also it became apparent that once an application has been re-styled a variety of interfaces can be produced.

## 7. Matching Toolkit Facilities With Applications Requirements

Given a toolkit and an application one of the difficulties during the design phase is how to match the interface requirements with the available tools or widgets. This problem is somewhat akin to what data structures to use in writing an application. But it is more than that. There is much more of a semantic meaning associated with some widgets than with data structures. Menus imply a grouping of similar tasks or functions. Command buttons imply, be it visual or audible, feedback and labels imply a lack of feedback.

While there are methodologies, both formal and informal, which can be applied to the design of interactive applications. These methodologies deal little with the actual implementation decisions on command grouping or hierarchies, muchless with the methods with which command selection is to be carried out. Much of these implementation decisions are intuitive. Referring back to the figure 1, the functional description of GEd, one can see many possible interface designs. It could be one large series of walking menus, a very long menu, a large selection table, or a mixture of all three. Other options are also possible. It this creative intuition that separates the good interface designer from the lesser one.

### 7.1. GEd's Interface Requirements That Were Supported

After the decision of GEd's interface abstraction model much of what was once viewed as application functionality now became a portion of the interface. Individual routines in GEd needed to be called from some sort of selection mechanisms, be it command buttons or menus. GEd had to be able to display text into a dialogue window. These functions trivially map to X toolkit widgets. However this was not the case with all of the interface requirements.

### 7.2. GEd's Interface Requirements That Were NOT Supported

GEd needs to be able to track the mouse, or at a minimum read mouse coordinates. There is not a track the mouse widget in the X toolkit, hence this paper. There is no simple implementation of help in the X toolkit. Individual help cannot be bound to individual widgets. While solutions for this are possible they are beyond the scope of this paper.

## 8. Extensions to the X Toolkit to Fully Support GEd

GEd as an interactive editor has the requirement for direct manipulation of picture objects on a display that is not under the control of the windowing system. This requires that mouse motion detected by the windowing system display is presented to the application for the manipulation of the picture objects. The principal extension required to the X windows toolkit was the support for mouse motion input.

The current selection of widgets provide for interaction from button and region enter/exit events, motion is only supported internally within the slide bar widget that provides for the positional selection of information. The toolkit must be extended to provide the required facility. In common with all toolkit extensions the existing tools must be examined to determine how much of the new requirement is supported by an existing widget and into which class the new widget should be introduced. Hence the functional requirement of the new widget must first be specified, as it "tracks" the motion of the mouse it was named the track widget.

### 8.1. Functional Requirements of the Track widget

These requirements were determined by an analysis of the operations associated with mouse motion that existed within GEd.

Requirements:

- Mouse motion callbacks in picture display coordinates
- Button events callbacks

The mouse motion events are used by GEd to allow various cursors to be positioned on the picture display to define regions etc. During these processes button events are use to signify the change between origin and extent specification, definition completion and to allow an operation to be aborted. To simplify application programming the widget should be configured to provide positional information with the same dimensions and resolution as the picture frame store.

### 8.2. Implementing the Track Widget

The existing X windows toolkit widget that implements the most of the functional requirement of the track widget is the command button widget. This correctly makes the track widget a member of the simple widget class. The track widget is implemented by extending the command widget, modifying the button callback to provide the status of all three buttons and introducing a motion callback.

The widget is created with a size that corresponds to the picture displays resolution so that the same local coordinate system can be used.

The motion callback is implemented by extending the event notification table of the original command widget to include motion events. When a motion event occurs the widget's internal routine is called with an event pointer, from this, the widget calculates the mouse's position within its local coordinate set and returns this information to the application by a callback. As the widget has no control over how long the application takes to process the callback all motion events that occur during the callback are compressed to a single callback at the current position. In practice this leads to a fast moving picture cursor with some visible steps, this is however much more acceptable that a cursor that lags the mouse's motion.

The motion values returned by the callback are directly related to the size of the track widget, this means that any changes to this will effect the operation of the widget. The resize strategy must always maintain the range and resolution required for the picture store. This imposes a minimum size on the track widget, in pixels, equal to the picture store. Resize requests less than this will be rejected. If however they are forced then only part of the picture can be used. Resize requests grater than the picture store resolution can be supported, in this case an active region of the correct size is maintained with variable size boarders up to the widgets actual size. This is achieved when processing the mouse position associated with the current motion event.

An extended resize strategy, that was not implemented, could monitor resize events and when the current widget size is greater than an integer multiple of the picture store size rescales the returned coordinates to create a lower mouse sensitivity. This means that accurate picture cursor positioning is easer but produces step changes of the sensitivity which could be very confusing to a naive user.

### 8.2.1. The Popup Track Widget

The provision of a track widget to provide a single bit resolution for a large picture store requires, with X windows, a widget that has the same pixel dimensions as the picture store. For a store with a resolution much greater that 256 by 200 pixels the widget visually dominates the applications window. This is considered unacceptable as an interface. The only solution is to popup the widget when a track operation is required. This new window then can take the required dimensions and provides the additional advantage of strong feedback to the user that a track operation is in progress.

### 8.2.2. Implementation of the Popup Track Widget

Problems were encountered during the implementation of the popup track widget that fall into two areas; those related to callbacks and those related to the hierarchal position of the popped up window with respect to the existing application.

The problem encountered with callbacks is the need for the application to attach callbacks to the track widget before it may have been realized. This will occur when the application function that is to process the positional information is attached before the widget is poped up. The problem is due to the application not knowing the widget id of the poped up widget as it does not yet exist. The solution adopted is to create an intermediate widget, the poptrack widget. This widget is used by the application for all forms of interaction; realization, adding callbacks etc. This widget responds to a button event in a similar manner to the command button, the application is called to notify it that a track type operation is to be initiated. The application adds the correct motion callback and returns. The poptrack widget then realizes and popups an actual track widget attaching two of its internal routines to the track widget's callbacks. These routines act as intermediaries that receive the motion and button data from the track widget and then call the application's callback routines. This allows the application to change it's callback routine without having to determine the actual track widget's identification.

By adopting this hidden interface between the actual track widget and the application, the application does not need to know if the user interface is programmed with a permanent or a popup widget. So, by maintaining a common interface between the application and the widgets, different users interfaces can be programmed for the same application with just a simple name change, from trackWidgetClass to poptrackWidgetClass, in the interface definition code.

Potential problems may occur with this approach if an application needs to dynamically change the size of the track widget whilst it is active. This is an unlikely requirement and the current restriction, of defining the widget's size at realization time, is considered acceptable.

The problem encountered that is associated with window hierarchy is believed to be associated with the implementation of the display server used. It was found that the new window containing the poped up track widget could be separately iconified with obvious user disorientation. As the action of the display server did not follow the documented responses these problems were not further investigated.

## 8.3. Functions NOT Provided by the Track Widget

When the widget is activated as a consequence of some picture operation initiation the user's attention should change to the picture display and therefore the user will not be concerned with the image on the windowing system's display. If, however, the user is distracted from both displays it is advisable to indicate on the windowing system's display that some operation was in action on the picture display, this will already be noticeable on the picture display.

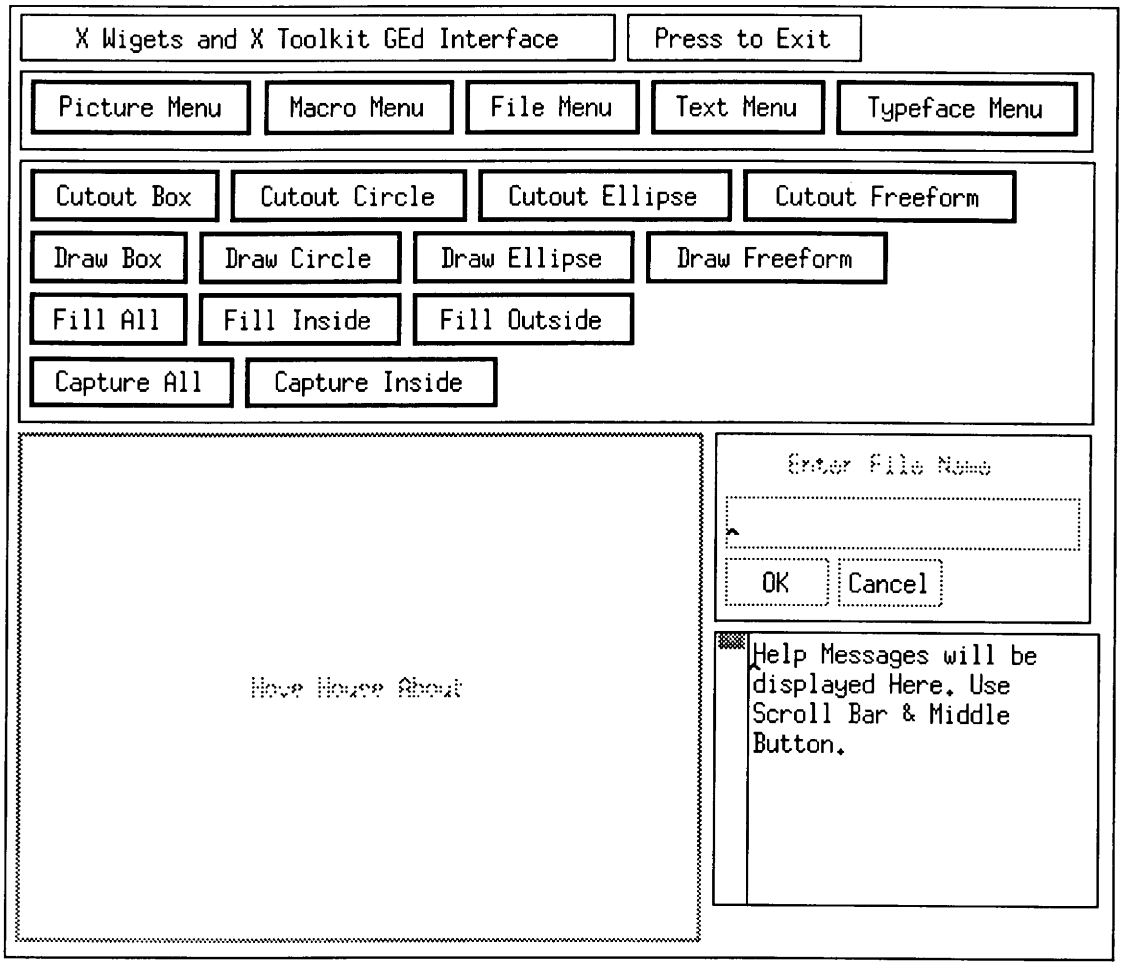
Currently only an indication that a track operation is in progress is shown, which particular application operation this is controlling is not immediately clear. To overcome this detailed operation feedback should be provided by the application.

## 9. An X Windows User Interface For GEd

The implementation of the **track** widget extended the functionality of the available widgets to allow an X Windows user interface to GEd to be implemented. The functionality of GEd available to the user is shown in figure 1. This figure highlights the hierarchical nature of the operational groupings. User experience with the original PC based version showed that presenting groups of operations on the screen simplified the learning process for new users. Therefore the design implemented attempts to present good sets of operations to the user.

## 9.1. The Design of The X Windows Interface For GEd

The interface developed does not use popup menus for two reasons; there was no suitable menu widget available within release 2 of X that was in use when the implementation started and that persistent menus offer a static view of the current group of operations. The user interface display is shown in figure 2.



Help Messages will be displayed Here. Use Scroll Bar & Middle Button.

Figure 2: X Toolkit Interface to GEd

The interface consists of a number of vertically ordered sections. The top section is the program title and a command button to exit the program. The second section consists of menu selectors that select the persistent menu displayed in the next section. Each menu selector corresponds to a major grouping of system functions. In practice the hierarchy shown in figure 1 is not exactly implemented in the interface, instead the menu selectors were chosen based on operation usage observed with the original system.

The middle section of the interface is the persistent menu for the currently selected operation group. The one shown is for the **Picture Menu** that implements the image manipulation operations. These menu items remain displayed until a new menu selection is actioned. They form the main interaction between the user and the application. When one of the menu buttons is pressed a callback is made to the application.

The lower section contains three interaction widgets, a track widget for mouse input, a dialog widget for file name operations and a text widget for help display. The track and dialog widgets are shown dotted as their sensitivity is controlled by the menu items.

When a user wishes to preform an operation, for example to define a region on the picture display, the **Picture Menu** selector is pressed, this updates the persistent menu to display the picture operations. Next the desired region, or cutout, operation is selected, e.g. **Cutout Box**. This causes the menu item to be highlighted, the callback generated attaches the correct track callbacks, sensitizes the track widget and desensitizes all other menu selectors and items. This provides visual feedback to the user of the type of track operation that is in progress. Details of the operation are displayed in the help section and the operation is performed by mouse events in the track widget. When the operation is completed the track widget is desensitized and the menu selectors and items re-sensitized.

### 9.2. The Implementation Of The X Windows Interface For GEd

The facilities provided by GEd are being extended as new versions of the system are implemented. This requires that new menu items and their corresponding application routines have to be added to the system. If the interface structure is directly coded as specific calls to the X widget libraries, extensions would require the program structure to be updated. To avoid this the implementation of the interface separates the details of the interface structure away from the interface coding. The concept of persistent menus and selectors is directly coded but the details of the selectors and items are read from a table. The table specifies the details of each selector and the table describing the associated menu. Each persistent menu is specified as a table with one entry per menu item. The table entry specifies the items label, button callback, any track callbacks and the help text.

Currently this table is a static data structure that is initialized at compile time. It would, however, be a simple task to allow the layout to be initialized at run time by reading from a data file. This would allow the interface layout to be adapted to an individual user's requirements.

## 10. Conclusions

Some of the problems associated with window systems — namely those of reusability, consistency in both look and feel, and portability — have been solved with the introduction of toolkits. However as a result of the inherent generality of toolkits not all application interface needs can be satisfied. Therefore in order to satisfy the needs of all applications and still maintain the goals of reusability, consistency and portability toolkits need to be extensible. This extensibility, however, should not come at the expense of reusability and integrity of the toolkit.

The X toolkit allows for extension. The track widget can now be used by other application interfaces, or different interfaces for GEd. The track widget does not interfere with the operation of any of the other widgets in the toolkit. Thus the extension has been completed without damage to either the reusability or the integrity of the toolkit or the individual widgets and therefore has maintained the goal of being more of a toolkit for toolkit builders.

### 10.1. Where Next? Or a Wish List

As a result of the design and implementation of an X toolkit interface to GEd possible future areas of investigation present themselves.

#### 10.1.1. Help

It might be nice to bind a help text message to each individual instance of a widget in a given interface. This text could be displayed in a popup, or a help display window, and be activated by a mouse button. Of course the mouse button would be constant across all widgets and as such would mean the loss of a button for other purposes. This could be done using some basic help object which would be included in all of the objects. This would necessitate modifying all of the base classes and as such would need to be done before the toolkit becomes commonly used or extensively extended.

#### 10.1.2. Debugging support

One of the most obvious short-comings of user interface toolkits is in the area of debugging support. With almost every software produced the idea comes first and the support for the ideas or utilities comes later. The need for strong debugging tools for both the interface level and the window system level are obvious to anyone who has programmed in a window environment.

This is somewhat difficult because some toolkits are built on top of more than one window system. The authors are aware of four toolkits alone that run on top of X. It would be nice if the debugger could examine, trace and monitor at both the toolkit level and the window system level. Debuggers need to be interactive so that window creation and destruction can be tracked. Also state changes in windows need to be reported. Since the window systems are quite dynamic by definition this requires interactive debugging mechanisms. Support for debuggers must be built in earlier if modifications to toolkits must be affected.

## 11. Acknowledgements

The authors are indebted to Mr. R. Weaver for writing GEd and his assistance in porting it from MS-DOS to UNIX. And Dr. M. J. Clark for his ideas and for putting together the VAN-Lab.

Dr. Jones a past member of the University of Essex, Department of Electronic Systems Engineering is now an employee of British Telecom's Research Laboratory at Martlesham Heath, Suffolk, England.

Mr Franklin has been funded under a Rotary International Scholarship during the course of the 1987/88 academic year. He wishes to acknowledge and thank Rotary International for the opportunity. It was during this year that much of this research was carried out.

## Copyrights

## References

Ath87a. Project Athena, *X toolkit*, MIT, September 15, 1987.

Fra88a. William L. Franklin, R. Weaver, and M. J. Clark, "Building User Interfaces for Image Display Systems," *Windows on the World*, DECUS, Cannes, France, September 5–9, 1988. Proceedings of the DECUS European Symposium

Gec83a. Jan Gecsei, *The Architecture of Videotex Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

Jon89a. S. T. Jones and M. J. Clark, *Authentication in a Unix Network Using Smart Cards*, EUUG, Brussels, Belgium, April 3–7, 1989. Conference Proceedings of the EUUG

McC88a. J McCormack and P Asente, "Using the X Toolkit or How to Write a Widget," *USENIX*, Summer 1988.

Mor86a. James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, vol. 29, no. 3, pp. 184–201, ACM, March 1986.

O´87a. Michael D. O´Dell, "What They Don't Tell You About Window Systems," *EUUG Conference Proceedings*, Trinity College, Dublin, Ireland, September 1987.

Sch86a. Robert W. Scheifler and Jim Gettys, "The X Window System," *ACM Transactions on Graphics, Special Issue on User Interface Software*, no. 63, 1986.

Siu88a. Y. T. Siu and M. J. Clark, "Distributed GKS For Multimedia Integration," *Windows on the World*, DECUS, Cannes, France, September 5–9 1988. Proceedings of the DECUS European Symposium

Swi88a. R. R. Swick, "X Toolkit Intrinsics – C Language X Interface," *MIT Project Athena*, p. 2, 1988.

Tei84a. Warren Teitelman, "The Cedar Programming Environment: A Midterm Report and Examination," Technical Report Number CSL–83–11, Xerox Corporation, Palo Alto Research Center, June 1984.

Wea88a. Roy Weaver, "An Image Editor and Network Display System," Master's Thesis, University of Essex, January 1988.

# The Design of a UNIX Workstation Environment for Medical Image Processing.

*Dirk Tombeur*
*Frank Deconink*

Experimental Medical Imaging Laboratory
Free University of Brussels
Laarbeeklaan 101,
B-1090 Brussels, Belgium.
*dirk@primis.vub.ac.be*
*frank@primis.vub.ac.be*

*ABSTRACT*

We are building a medical image processing environment, based on general purpose UNIX workstations. These workstations are integrated in a heterogeneous network, which allows resource sharing and communication. To promote modularity and extensibility, the environment is based on a hierarchy of modules. These building blocks are loosely modeled after Smalltalk-80 classes. At the application level an interactive image processing tool has been created, which is manipulated through a graphical user interface. C++ is used as an implementation language. To complement the object oriented constructs of C++, we implemented a garbage collector, for managing complex dynamic data structures.

## 1. Introduction

The Experimental Medical Imaging Laboratory at the V.U.B. has a major interest in problems related to medical imaging. Currently we are investigating practical and theoretical aspects of algorithms, software environments and hardware architectures for applications in Nuclear Medicine (N.M.).

Our main project consists a novel design for a Positron Emission Tomography (PET) camera, the High Density Avalanche Chamber (HIDAC) camera, of which a prototype is currently operational. PET is a relatively new quantitative N.M. imaging modality, that allows to in vivo measurement human physiology and metabolism. The main stages in PET imaging consist of positron data acquisition and tomographic reconstruction of the data, followed by display and manipulation of the reconstructed images. The computing environment for the PET project is composed of six Apollo workstations, linked in token ring local area network (LAN), that is connected to the Medical Faculty Ethernet LAN. This configuration has proven to be very flexible and efficient for data acquisition and reconstruction of the PET images. Its major drawback is the lack of a powerful, easy-to-use interactive environment for image processing (I.P.) and visualization. Therefore we decided to build an integrated environment, that supports the specific needs of the PET project and allows the collaborators of the clinical N.M. department to carry out the clinical evaluation of PET examinations and compare them with other N.M. imaging modalities.

Our entire software environment is based on an object oriented architecture. It currently runs on an Apollo DN3000, equipped with 4 colourplanes and running 4.2BSD UNIX, it is implemented in C++ and features a graphical user interface (G.U.I.), which enforces a consistent interactive behaviour. Both non-interactive (tomographic reconstruction) and interactive image processing tasks are supported. We developed a consistent framework which unites scalar, 1-D and 2-D data processing and data organization. A key element in this framework is the garbage collector, which keeps track of dynamic data structures used by the both the G.U.I. and the I.P. routines.

It is difficult to compare our approach with commercially available N.M. systems, which are usually dedicated stand alone systems that come bundled with the imaging equipment and which do not take into account issues such as hardware and software standards or the use of modern programming languages. However, we believe that our approach offers a better protection of the investments made in software, by allowing the system to evolve to more powerful hardware whenever it becomes available.

The aim of this paper is to give an overview of our design choices and to treat in some detail the innovations introduced in our system, in comparison to commercial systems. In section 2, some background about N.M. imaging modalities is presented. In section 3 we first present a general system overview and then proceed with a discussion of the graphics support classes, high level dynamic data structures, the garbage collector and image I/O. Section 4 discusses the application components. Section 5 states the conclusion and describes future developments.

## 2. Background about Medical Imaging and Nuclear Medicine

Major research efforts in medical imaging are currently devoted to problems of Picture Archiving and Communication Systems (PACS). PACS systems are traditionally associated with radiology, because they aim to duplicate the capabilities of traditional film-based radiological departments, without the use of film. The large amounts of data produced by radiological systems and the high spatial resolution of radiological images explain why image processing techniques are considered to be of secondary importance in radiology.

From a historical point of view, Nuclear Medicine has been the first medical imaging discipline, that used computers for both data acquisition and data processing (1960's). Computers are not only indispensable for the processing and visualization of N.M. images, but also made computed tomography and dynamic imaging feasible. In contrast to radiology, N.M. makes heavy use of I.P. techniques because images produced by N.M. modalities have a much lower spatial resolution than radiological images and are inherently noisy. However, because N.M. images are also much smaller than radiological images, they are more suitable for interactive graphical and computational manipulation on general purpose computers, than radiological images.

In N.M. both static and dynamic imaging modalities are used. In static examinations, tomographic reconstruction is used to determine the distribution of radiolabeled pharmaceutical compounds or radiolabeled monoclonal anti-bodies. After reconstruction, visualization requires real-time extraction of 2-D slices along the main body axes (sagital, coronal and transverse), as well as oblique slices, from the reconstructed 3-D pile of 2-D image slices.

Dynamic modalities are used to study flow through organs (cardiac, renal) or effects on the nervous system of external stimuli (neurological studies). Data reduction techniques must be applied to extract meaningful descriptions from raw data, which vary both in space and time. In cardiac imaging, the spatial distribution of both amplitude and phase of the first harmonic component of the movement of the cardiac blood pool is computed by Fourier analysis techniques. The resulting phase-amplitude images "summarize" the kinetic behaviour of the heart muscle. In neurological studies, brain metabolism is measured under varying conditions of visual or auditory stimulation, by determining the spatio-temporal distribution of radiolabeled compounds.

In the examples cited, 2-D results are obtained from operations on 2-D or 3-D images, whereas other operations such as histogram and profile calculations typically generate 1-D curves. This implies that a system for N.M. must allow flexible manipulation of mixtures of scalar, 1-D and 2-D objects. The combination of relatively small data sets, relaxed requirements for visualization functionality and the need for interactive I.P. capabilities makes N.M. an interesting subject for the introduction of new approaches.

## 3. General System Overview

The main high level components which we have implemented are a file browser, an overview facility for large collections of images, a colour scale browser and an interactive image processing tool which we call the Image Calculator (ImCa).

We decided initially not to distribute these tools over different processes. Although this approach would be the natural one to follow in a UNIX environment, we first wanted to concentrate on the user interface, the internal data organization and the enduser functionality. This approach also temporarily eliminates the problems involved in interprocess data communication. Only the tracking of the interactive input devices (mouse, keyboard) is done in a separate process.

In the main process, graphics, calculations and data I/O operate in a serial fashion, which effectively limits our present system to single tasking. The two processes share an event-queue, on which the device tracking process posts events, which can be retrieved by the main process. When lengthly activities monopolize the system, the separate process for interactive input tracking allows the user to send an abortion signal to the other process. Of course a suitable fault handler must be installed, to enable a graceful restart of the activities in the main process. We used C++ as an implementation language, because we needed a language that supported object oriented programming without the notorious performance penalties of pure object oriented languages such as Smalltalk-80.

## 3.1. C++ Class Hierarchy

Our classes are organized in a hierarchy, modeled after Smalltalk-80. The root of this hierarchy is the class `Object`. All other classes, except those involved in low level memory management, are descendants of the class Object. The main functional categories provided consist of collection classes, G.U.I support classes, classes to support I.P. algorithms (real and complex matrices and vectors) and image I/O classes. The low level classes implement the interface to operating system and device dependent functions. The protocol of the lowest layer allows access to I/O services (file system, interactive devices), graphics services and memory management primitives. On top of this layer, classes which support dynamic data structures, computational classes for I.P., high level image I/O and G.U.I. classes are built.

## 3.2. Graphics Support Classes

The class `GrafPort` implements a drawing canvas with a local coordinate system and clipping to rectangular areas. Class `RenderDevice` executes graphical primitives on `GrafPorts`, such as stroking and filling of geometrical shapes and raster operations, by translating them (in our case) to Apollo graphics system calls. The `Shape` class defines empty (virtual) mathematical functions, such as `intersect(aShape)` and `containsPt(pt)`, and empty (virtual) graphical functions such as `frame()`, `fill()`, `paint(aPattern)`, `erase()` and `invert()` for geometrical shapes. Subclasses `Rect`, `Polygon`, `Arc`, `Circle` and `Line` each implement the requested behaviour in the corresponding functions.

## 3.3. Graphical User Interface Implementation

Apollo provides a window manager for overlapping windows, connected to concurrent process and Dialogue, a toolkit to build graphical interfaces. We decided not to use the window manager because it did not allow full access to the graphical primitives and the interactive input devices. The U.I. building toolkit was rejected, because it is a closed, proprietary system and it is tied to the Apollo window system. The user interface is implemented on top of the geometry and graphical classes.

The base class for all G.U.I. objects is `Control` which defines virtual functions such as `show()`, `hide()`, `track(whichPart)`, `checkInControl(pt)` and `drag(whichPart)`. The concrete visual and operational behaviour is implemented in subclasses of `Control`: `PushButton`, `RadioButton`, `CheckBox`, `ScrollBar` and its descendants `HScrollBar` and `VScrollBar`, `Slider` and its descendants `HSlider` and `VSlider` and `ListBox` and `MultiListBox`. The class `TextEdit` implements single line editing with cut and paste functions.

The windowmanager classes are implemented using a `GrafPort`. The base class `Window` defines standard functions to draw, hide, move and resize windows and make a backup of the screen area it covers. The main intention of its descendant classes is to draw each window with a different visual flavour and to implement non-standard behaviour. The class `Cursor` manages shape and visibility of the cursor. This class organization is conceptually very simple and it liberated us from the limitations of the closed Dialogue toolbox, but it also introduced a lot of new complications.

In our system, sizes of controls are defined in screen pixels and all control coordinates are specified in absolute window coordinates. Some systems allow interactive design of U.I. elements with a Resource Editor, which effectively eliminates a lot of tedious programming. In Apollo's Dialogue environment, the programmer creates a description of the U.I. elements in terms of minimal, optimal and maximal size. The spatial composition of different controls can be done in a declarative fashion, using `row`, `column`, `popup` and `oneof` composition primitives. This relieves the programmer from explicit positioning and it supports automatical resizing of controls.

Another problem is the fact that synchronization of control actions, such as clicking a scrollbar arrow, and the corresponding action of scrolling the image, requires to much programming effort. More complicated still is the synchronization of different views of the same data object, when one of the views changes. This occurs e.g. when the position of three orthogonal slices is controlled by reorientation of the image volume in 3-D space. To remedy these problems, we are now in the process of reimplementing the U.I. classes, to incorporate synchronization among the actions of the U.I. elements and the (visual) representations of the data objects on which they operate.

## 3.4. High Level Dynamic Data Structures

Dynamic data structures are of prime importance for the construction of flexible user interface tools. They are also extremely valuable to serve as data structures in complex image processing algorithms, such as graph based segmentation procedures or hierarchical image representations. We designed a number of general support classes for commonly used dynamic data structures. These classes are again loosely modeled after Smalltalk-80 classes, although they provide a much leaner interface. We also reduced the depth of the hierarchy because deep class hierarchies, although conceptually cleaner are much more difficult to grasp than shallow hierarchies.

The often heard complaint, that it is impossible to create truly generic container classes in C++, should be put in the proper perspective. Because it only makes sense to group related objects, container classes can be designed to accept a suitable pointer type as object reference. When all the objects in the collection share the same characteristics, it will be safe to use explicit typecasting, to recover the full functionality of the objects. This is illustrated by the following example. In our system, all windows are kept in an `OrdCltn`, which is maintained in the global variable WindowList. OrdCltn's use a pointer of the type Object* to reference their members. Because the OrdCltn that manages the windowlist only contains references to objects belonging to class Window or its descendants, it will always be safe to use constructions such as:

```
    topWindow = (Window*)(WindowList->first());
```

It somehow would not make sense to store image slices or textstrings in WindowList!

## 3.5. Garbage Collection

When objects become part of tangled, linked structures, it becomes increasingly difficult to decide when an object has become unreachable, so that it may be deallocated. C++ allows flexible automatic deallocation of objects through destructor functions, but memory compaction cannot be supported easily. Therefore we decided to implement an automatic garbage collector, which reclaims unreachable objects and compacts the free memory. The G.C. uses the *Generation Scavenging Algorithm* , which was recently developed for Smalltalk-80 systems by David Ungar of University of California at Berkeley. Generation Scavenging uses a combination of interactive and offline memory reclamation, which ensures low overhead and no noticeable disruptions in interactive applications. Basically reachable objects are copied a number of times among two memory subspaces. Objects that survive a number of subspace swaps are moved to an area which is only collected offline.

In our implementation, objects reside in different memory subspaces, according to their size, to minimize copying overhead. Each subspace can then be managed by a suitable G.C. or not at all. The class `Memory` defines empty virtual functions `newChunk(numOfWords)` and `garbageCollect()`, which must be reimplemented in subclasses. One subclass, `DynMemory`, is intended to hold small data structures, which are heavily used in various collection classes and in geometrical or U.I. primitives. These data structures are managed by the Generation Scavenging G.C..

Because the repeated copying of images would introduce an intolerable overhead, they are managed by another subclass of Memory, which is implemented as a large array, containing a fixed number of fixed size, square image matrices. Because all images within one memory space have a fixed size, memory compaction is not necessary. Free slices are kept in single linked list and also maintain a reference count. When a slice header changes its pointer to another slice, the reference count of the original slice is decreased and the new one is increased. When a slices reference count reaches zero, it is linked again in the free list.

The use of the G.C. facilitates management of multiple referenced objects, but it also has some drawbacks. In our implementation, all variables which reference garbage collected data structures have to be pointers. This induces some notational inconvenience, because standard operators can only be overloaded, when at least one of the operands is a class type object. This implies that in our case, at least one of the operands in an operator expression must be dereferenced. This leads to notations such as:

```
    *image1 = log(*image2) + 2.0 * (*image3);
```

which somewhat reduces the merits of overloaded standard operators.

When object references are transmitted in function arguments, the copy of the arguments in the stack frame must be accessible by the G.C., because it is possible that memory compaction could start during a heavily nested function call. During memory compaction all data structures (in DynMemory) are moved, which would invalidate the pointers in the stack frame. Unfortunately the G.C. must be informed by the programmer which arguments have to be tracked, and this is not only annoying, but also error prone.

## 3.6. Image storage and image I/O

Currently no consensus exists about image storage formats among manufacturers of medical imaging equipment. This implies that for each particular storage format, procedures must be designed to do I/O operations in the required format. To isolate this problem from the rest of the environment, we designed a mechanism which enables different formats to be treated through a common interface. During the opening of an image file, the type of the file is determined by inspecting a descriptive header file, which MUST be present in the same directory. The object that is returned by the open operation resembles a stream object, for which operations such as `getSlice(index, view)`, `getNextSlice()` or `getNumSlices()` are defined. All interactions with the image file are executed through the image stream, which isolates the application from particular storage formats. This implies e.g. that if an image is stored in a compressed format, the decompression is done in the image stream, rather than in the application. At present however, we load the entire file content immediately into a virtual memory buffer on which all operations are conducted. Access to the virtual memory buffer is also controlled by an image stream. The connection between this memory buffer and the original diskfile is maintained throughout the life of the buffer. If the buffer is discarded, the application allows the user to merge the buffer with the original diskfile, or rather record it in a separate file. This extra level of indirection is much safer than operating directly on files.

## 3.7. Implementing Image Processing Algorithms in C++

A flexible I.P. environment should allow use of data structures with various internal representations, without having to worry about details. Overloaded standard operators and overloaded functions make this possible. In our system, classes define operations on real and complex (both cartesian and polar representation) valued scalars and 1-D and 2-D objects, with a set of standard coercions e.g.

```
    real + complex = complex.
```

To avoid problems related to overflow and transcendental functions, computational image structures always use floating point representations. Besides the usual arithmetical and transcendental operations, typical I.P. operations are provided, such as spatial and frequency domain filtering, orthogonal transforms (Fourier, Wash-Hadamard, Discrete Cosine Transform) and geometrical registration operations. These classes are used by the Image Calculator (discussed next).

## 4. Application Level System Components

We will now present an overview of the system components which are visible to the user of the environment.

## 4.1. Image I/O and File Browser

All I/O is controlled through a graphical file browser, which shows the current directory path and the files in the current directory. A user can walk the directory tree, either by selecting a component in the directory path or by selecting another directory in the current directory. Only names are displayed in the file browser, but the user may select a name and request detailed information (ls -l) by choosing the *UNIX info* button in the browser window. This will pop up another modal dialogue window, which contains extensive information about the object selected and allows modification of the protection modes through checkboxes. When an image is selected and the load button is clicked, the browser window will disappear and a progress window will give an animated picture of the loading of the image.

### 4.2. Image Overview Component

After an image is transferred to a buffer, the user can display an overview of the content of the buffers. For a tomographic reconstruction, the overview usually contains three groups of 64 orthogonal slices, totalling 192 slices. On the average, we need 10 seconds to put these 192 slices in a 64x64 pixel format on the screen. On the screen, the real dimension of the preview images is approximately 15 mm. These miniature images provide an initial impression of image quality and features contained in them. The user can select individual slices from the overview, which are enlarged by pixel replication and shown in a separate view. The overview is backed up in a main memory pixelmap, which is used to present selections of slices in the ImCa.

### 4.3. Colour Table Browser

This tool allows modifications to be made to the colour lookup table (L.U.T.). The user can choose among a number of predefined palettes, such as Gray, InvertedGray or Cyclic (for phase images). He can also compose his own palette, by defining mixtures of red, green and blue values for the lookup table entries. These values are controlled through Sliders, which are depicted as linear potentiometers. This feature can also be used for highlighting a particular range of pixel values.

### 4.4. The Image Calculator (ImCa)

One of the main problems in interactive image processing is the organization of calculations. Users must be able to control in an intuitive and consistent way which operations are to be applied to which images. Operations can be unary, binary or n-ary. In our system, images can only be operated on, if they are put on an image calculation stack. This is done by presenting overviews of memory buffers to the user. From this overview, images can be selected and pushed on the calculation stack. Operations on image stacks are modeled after the RPN calculators from Hewlett-Packard. They include control operations (push, pop, last, swap, hop, clear, roll) and mathematical operations on images (convolution filters, arithmetical and transcendental functions, ...). Only images of the same size and dimension are allowed on one stack. This is no limitation, since multiple stacks may be active simultaneously and transfers of images, which reside on different stacks, are supported by using subsampling or enlargement functions, defined in the image classes. On the screen, each calculation stack has an associated view. These views are kept in synchronism with the contents of the stacks. Using stacks to organize images considerably simplifies control structures. Unary operations are always executed on the top element of the stack. Binary operations are executed on the two first elements on the stack. They are popped off the stack, and the result of the operation is pushed on the stack again. If the result of an operation on a 2-D image is a 1-D curve, the problem of differences in dimensionality arises. Therefore we extended the paradigm from images to curves and scalars. If e.g. a histogram of grey-values of the top of the current image stack is created, the result is automatically pushed on one of the curve stacks. Continuing this line of thought, if the means and standard deviations of the histogram data are calculated, the results are pushed on the scalar stack. Our approach excludes syntactical ambiguities, because the stack paradigm amounts to postfix expression of all operations.

## 5. Conclusion and Future Developments

We have presented an overview of the interactive image processing environment, that we are developing to support Nuclear Medicine applications. It differs from commercial systems, because it uses modern graphical workstations, which run the UNIX operating system. It is implemented in C++, and features a Graphical User Interface, a Generation Scavenging Garbage Collector and a large collection of I.P. tools. The general purpose Apollo workstations perform remarkably well in terms of computational and graphical speed, even in comparison to special purpose commercial N.M. systems. The system is continuously evolving and we are preparing it for clinical evaluation.

We plan a number of extensions and enhancements of the system:

- We need to implement a binding of the graphics primitives to the X Window System, to further enhance the portability of the system.
- The G.U.I. classes need to be redesigned in more object oriented way. We will continue to use a single look-and-feel, but provisions will be made to allow experiments with other visual designs.

- We need a computational and a visualization tool for 3-D images.
- We will also implement a spreadsheet based image processing tool, to complement ImCa. A spreadsheet allows an intuitive and consistent organization of operations, which is different from a stack based approach, but which seems more appropriate for single images.
- Currently the networking facilities of our environment are only used to access remote files. To make better use of the computing facilities of the network, client-server models for distributed computing have to be introduced.
- We will also have to implement a frontend, that maps the clinical entities patient and examination on the physical storage structures, which are based on the hierarchical UNIX file system.

## 6. References

[1] B. Stroustrup, *The C++ Programming Programming Language*, Addison Wesley, 1986

[2] A. Goldberg, D. Robson, *Smalltalk-80 The Language and its Implementation*, Addison Wesley, 1983

[3] D. M. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, MIT Press, ACM Distinguished Dissertation 1986

[4] D. W. Townsend, R. Clack, M. Defrise, H-J. Tochon-Danguy, P. Frey, S. Kuijk, F. Deconinck, *Quantitation with the HIDAC positron camera*, Information Processing in Medical Imaging, Plenum Press, 1987

# Occursus Cum Novo – Realistic Movies rendered in an UNIX-Environment

*Wolfgang Leister, Burkhard Neidecker,*
*Achim Stößer, Heinrich Müller*

Department of Informatics
University of Karlsruhe
West German
*leister@ira.uka.de*

*ABSTRACT*

The goal of the project **Occursus Cum Novo** was to generate a complex photo-realistic animation of nontrivial length in reasonable time at reasonable cost. Photographic realism comprises complex geometric models as well as the implementation of several optical effects. Both can be achieved by simulation. Simulations guaranteeing high quality, as ray tracing does for rendering, are known to be very time consuming. The film has a length of 5 minutes and was done entirely on a network of about 30 UNIX work-stations of type SUN-3. The organising scheme is described. Processing is done automatically in such a manner that it does not interfere with interactive users. The results of the project are of general interest since they show a way leading to efficient high quality photo-realistic animation synthesis in the future.

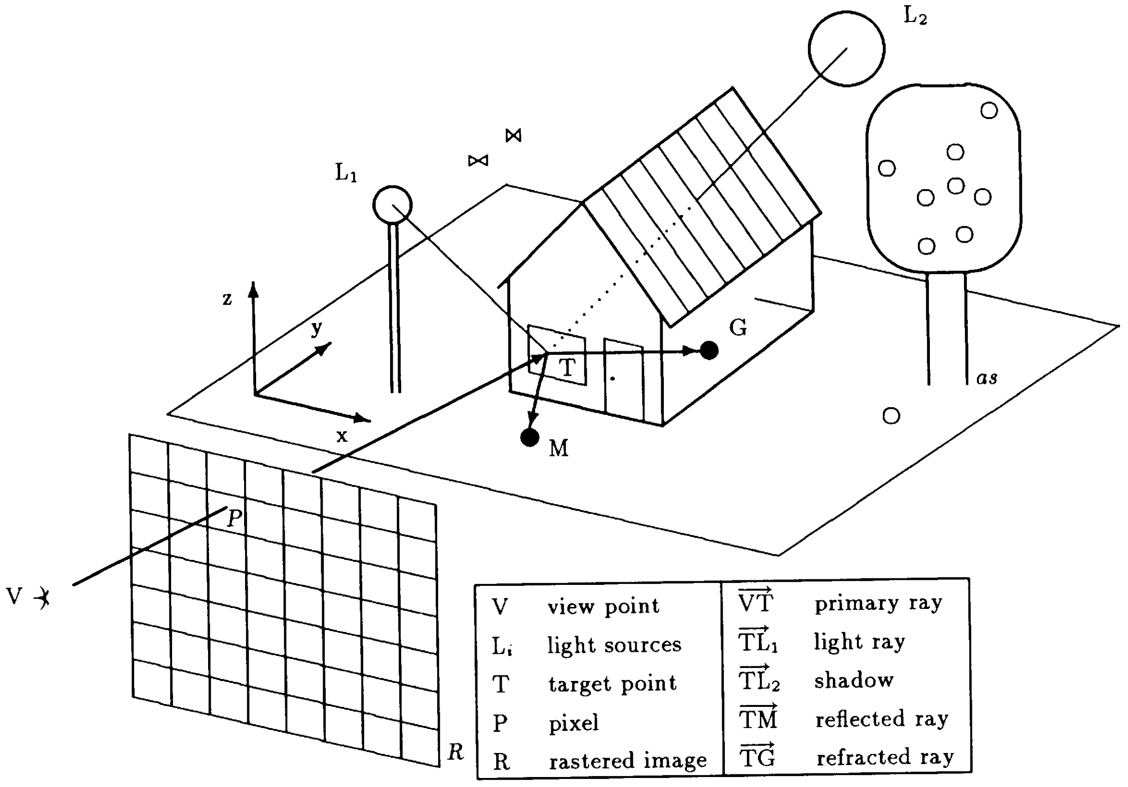
## 1. The VERA Raytracing Software

The correct simulation of various lighting effects is the most important part of photographic realism. Perfectly looking realistic images can only be generated on powerful computers using sophisticated software. This holds in particular for *ray tracing*. Ray tracing was brought to a broad knowledge by Whitted [Whi80]. In its simplest form, for each pixel of the image a ray is traced from the viewpoint into the 3d scene to calculate its first intersection with an object. For reflecting or refracting objects, a generation of new rays will be drawn according to the laws of physical optics. These new rays are treated analogously. In order to calculate shadows, we must find out whether the ray between the intersection point and the particular light source is intersected by another object. If there is an object intersecting this ray, the intersection point lies in the shadow of this light source, and its intensity is not taken into account for intensity calculations. The calculation of intensity is carried out according to a lighting formula incorporating parameters of material attached to the geometric objects of the scene. To give you more understanding on this subject look at figure 1.

VERA (Very Efficient Raytracing Algorithm) is able to generate highly realistic raster images of scenes consisting of a great number of objects. The program, written in standard Pascal and therefore portable to many computers, consists of three main parts. In a first pass, the scene is read in, while the second pass transforms the scene into a suitable internal data structure for the third pass, where raytracing is done. The output is a compressed raster image.

For VERA, independent of other researchers [Gla86,FTI86], the grid technique including grid traversal by a DDA was developed. A difference to the other approaches is the marriage of hierarchical scene specification with the grid structure. This alliance allows us to render large scenes, since the grid structure adapts to the spatial distribution of the geometric primitive objects. More details on VERA are beyond the scope of this paper, and are outlined in [SML87].

Knowing about the quality of the images generated by VERA, a committee of the Austrian Television (ORF) decided to give a grant for a computer animation project to be submitted to the Prix ARS Electronica '87 in Linz, Austria. The script of Occursus cum novo tries to plot out an amalgamation of four realms: science, technics, nature and art. This is done by citing various *real* objects like an alarm clock, chess pieces, trees, wind mills, a painting by Mondrian, a sculpture similar to one of Jean Arp etc. The world in the animation sequence is separated in the *hard* and *soft* world, characterised by speed and rough motion in the hard world and calmness in the soft world. While the film is running the worlds amalgamate.



| | | | |
|---|---|---|---|
| V | view point | $\overrightarrow{VT}$ | primary ray |
| $L_i$ | light sources | $\overrightarrow{TL_1}$ | light ray |
| T | target point | $\overrightarrow{TL_2}$ | shadow |
| P | pixel | $\overrightarrow{TM}$ | reflected ray |
| R | rastered image | $\overrightarrow{TG}$ | refracted ray |

**Figure 1**: *The principle of raytracing*

## 1.1. Specifying Realistic Spatial Scenes

The operational basis for rendering are the VERA scene files. A VERA scene file contains the textual description of a static scene, *ie* every frame is described by a separate file. An example is shown in figure 2. VERA accepts a set of primitive objects, which are used to build up a scene. Let us mention triangles (P3)[1], spheres (SPH), and rotational shapes (RS). The design of more complex scenes is greatly simplified by applying a hierarchical concept. We can use subscenes (SCENE), that can be specified several times (SSC) under different transformations. This technique is very similar to the procedure- or macro-concept of programming languages. Some advanced features to describe patches, a horizon, camera facilities and automatic exposure measurement are included. In this manner complex scenes can be built. A file like this is either specified by a text editor, or results by conversion of the output of an interactive graphics editor or a CAD system.

## 1.2. Preprocessing

The specific power of VERA is to process large scenes in relatively moderate time. This is made possible by decreasing the number of intersection tests to be performed among rays and primitive objects by preprocessing the scene into a suitable data structure. The quality of this data structure is crucial for the practicability of ray tracing. In the preprocessing phase, VERA partitions the bounding box of a scene into a regular grid of congruent cells. For each cell, a list of objects is made up, containing all those objects that have a nonempty intersection with the cell. Subscenes are treated analogously to primitives, using their bounding box for assigning them to cells. The resulting data structure is a tree of grids, which usually shows a good adaption to the distribution of the primitive objects of the scene.

1 The identifier in parantheses denote the identifiers used in the VERA-System

```
Name Newton                          name in pixel file header
Size 780 576                                        frame size
Anti-Alias                                   alias treatment on
RayTrDepth 3                          maximum ray tracing depth
Camera 0 -500 300                                   view point
 0 0 25                                      point of interest
 0 10 140                                       top edge point
EyeLight 300000 1 1 1          white light source in view point

Col red                                apple (for test: red sphere
Sph 0 0 92 12                    with center 0; 0; 92 and radius 12)
Col metal                                 table: reflecting square
P4 -200 -200 0   -200 200 0
 200 200 0   200 -200 0

Id red
 Drf 1 0.2 0                              diffuse reflection white
Id metal
 Drf 0 0.5 0.7                        diffuse reflection sea green
 Rld 0.3 0.3 0.8                                    bluish mirror
```

**Figure 2**: *Example of a VERA scene specification.*

## 1.3. Tracing the Rays

Ray tracing is carried out by a recursive procedure *RayTrace* that reports all cells of the grid structure of a scene intersected by the ray. Reporting the cells intersected by a query ray can be implemented in a highly efficient manner by using a 3d-vector generator, for example the DDA. For the objects in the list belonging to a reported cell, an intersection test is carried out. This works straightforward for the primitive objects; for subscenes, the ray is transformed into the grid of the subscene and processed analogously. Of all intersections the closest one to the starting point of the ray is taken. A recursive call of *RayTrace* is performed for reflecting or transparent objects. Rays to the light sources are treated analogously.

## 1.4. Illumination

When an intersection point is found, its contribution to the total intensity of the pixel under consideration must be calculated. This is done according to the lightness formula. For each of the basic colours red, green and blue we calculate the colour intensity due to the circumstances given for geometry and color specification. For the geometric specifications see figure 3. Besides the well known coefficients for reflection we also use a coefficient for self-luminosity, both angle-dependent and diffuse. The formulas used by our programs are outlined in the figures 4 and 5.

## 2. Modeling an Animation

In the above section we highlighted how to model a scene by describing the objects in textual form. The question is, how to use this technique for animation purposes. One possibility is to describe each frame separately. However if you look closer to an animation sequence you realise that each image differs only slightly from the next one. So it would make no sense to describe each image separately.

One solution is to introduce variables into the VERA source code. This can be done by using a special escape sign in front of a text string, *ie* `#FRAMENUMBER`. Thus you can use UNIX tools for replacing these strings with the actual value. In our case we use `cpp` or `sed` as processing tools. For an example consult figure 6.

For each image we must process the assignment of variables in the scene description file. The rendering takes place. After this we clean up by compressing the raster images (about 1 MByte per image) and by removing temporary files.

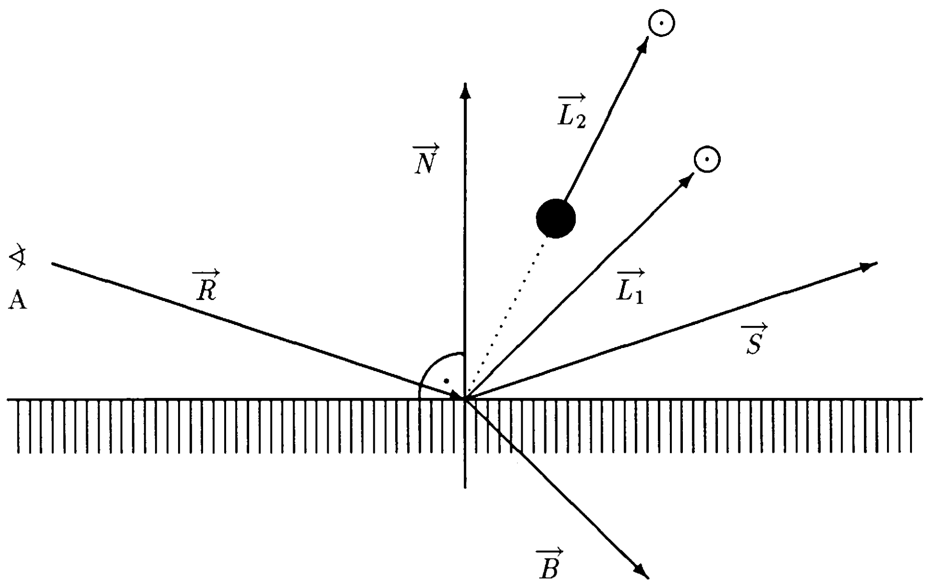


**Figure 3**: *Geometry of the ray on a surface*

The shell script in figure 6 is called once for each image. Thus we need an other instance that calls this file with the correct parameters. This is also a shell script using one line per image, due to the automatically working rendering procedure in the network we are using for distributed animation calculation. An example for such a script is outlined in figure 7, where Newton's apple falls from the apple tree. The frame number is in the first parameter, the height of the apple in the second.

Naturally it is not intended to calculate the movement by hand but by a special purpose program that does the animation. Here we need a small program that calculates the values by using the formula $s = \frac{1}{2}gt^2$ Each value is written on one line together with the frame number and the procedure calls.

This is a rather simple example. For some scenes we had to use special purpose programs to be used once, in other cases we built some more complex simulators. Another technique we used was to include some files dependant on the image number. This is an important feature, if some complex subscenes change while other parts of the image remain motionless. In this manner the trees, the rays hitting the chess pieces or the balls were given birth.

## 3. Organising Distributed Rendering

Our distributed rendering environment is based on UNIX 4.2BSD driven work stations coupled by ethernet. Besides low level protocols, the Sun Network File System (NFS) [Sun86] is installed. We were actually using mostly SUN-3 machines. About 30 machines were installed by the time we worked on the project.

For the organising scheme we defined the following goals,

- make good use of idle times
- do not interfere with interactive users on all machines
- do not interfere with any users on certain machines
- make it easy to submit rendering tasks
- full automatisation with simple recovery in case of crashes.

The solution can be sketched as follows. Rendering is carried out by a number of jobs, one for each frame. The jobs are organised in a queue, called the **network queue**, which is held on one machine acting as a server. All other machines are client machines, getting jobs from the server, executing them, and reporting the results back to the server. If a client is accessed by a user, the currently running job is killed. Killed jobs are reported to the server to be reinserted into the network queue.

**portions of intensity**

| | | |
|---|---|---|
| $I(Rvec)$ | total intensity | |
| $I^{(d)}(Rvec)$ | intensity at raytrace depth d | |
| $R$ | portion of reflected light | |
| $H$ | portion of high light | |
| $A$ | portion of ambient light | |
| $S$ | portion of self luminosity | |
| $R_{refl}$ | portion of reflected ray | |
| $R_{refr}$ | portion of refracted ray | |

**vectors**

| | | |
|---|---|---|
| $\vec{R}$ | direction of visual ray ($\overrightarrow{P_E P_T}$) | |
| $\vec{N}$ | surface normal | |
| $\vec{L_i}$ | ray to light source $i$ ($\overrightarrow{P_T P_{L_i}}$) | |
| $\vec{R_{refl}}$ | reflected ray | |
| $\vec{R_{refr}}$ | refracted ray | |

**rgb properties**

| | | |
|---|---|---|
| $R_d$ | diffuse reflection | `Drf` |
| $R_a$ | angular dependent reflection | `Arf` |
| $S_d$ | diffuse selfluminosity | `Sdi` |
| $S_a$ | angular dependent selfluminosity | `San` |
| $D_{refl}$ | dimming of reflection | `Rld` |
| $D_{refr}$ | dimming of refraction | `Rrd` |
| $\Psi$ | ambient light | `Light` |
| $L_i$ | colour of light source | `Li, EyeLight, Spot` |
| $\Gamma$ | intensity of highlight | |

**real values**

| | | |
|---|---|---|
| $r$ | reflection lobe | `Rlb` |
| $f$ | refractive force | `Rfc` |
| $l_i$ | intensity of light source i | `Li, EyeLight, Spot` |
| $n$ | number of light sources i | `Li, EyeLight, Spot` |
| $d$ | raytrace depth | |
| $d_{\max}$ | maximum raytrace depth | `RayTrDepth` |
| $\delta_i$ | 0 if hidden, 1 else | `Shadow` |

**points**

| | | |
|---|---|---|
| $P_E$ | eye point | `Camera, Projection` |
| $P_T$ | target point | |
| $p_{L_i}$ | position of light source i | `Li EyeLight, Spot` |

**Figure 4**: *The variables of the illumination formula*

On each client a demon (background process) is installed during boot time. This demon asks every five minutes whether the machine is idle. If idle, a job is accepted and a child process is initiated to execute this job. Then the demon continues checking every 10 seconds whether the machine is still idle. If not, the child process is killed and the corresponding file is moved back from `work` to `jobs`. Process killing requires that VERA can continue at any time independent of the moment of the interrupt, without loosing too much of the already processed image. Furthermore, the status of the child process is supervised every 10 seconds. If an accident happens, the job is moved to `bad`. When a job is finished, it is moved to `done`. The sets of jobs are represented through directories and files in the file system (see also figure 8).

Technically, the network queue organisation is based on NFS [Sun86] The network queue is implemented as a part of the file system of the *server*. It resides in a subdirectory named `/netq`, cf. figure 8. This directory is subdivided into two parts: The directory `work` contains all input-, output- and intermediate files required for frame rendering. The directory `ctrl` contains all data necessary to control the network queue. Figure 8 shows the subdirectories of `ctrl`. These subdirectories are used as follows:

$$I(R) = R + H + A + S + R_{refl} + R_{refr}$$

$$R = \sum_{i=1,\delta_i \neq 0}^{n} \left[ \vec{L}^{o}{}_{i} \cdot \vec{N}^{o} \cdot R_d + \frac{1 + \vec{L}^{o}{}_{i} \cdot \vec{R}^{o}}{2} \cdot R_a \right] \cdot L_i \cdot l_i \cdot \frac{1}{\vec{L}_i^{\,2}}$$

$$H = \sum_{i=1,\delta_i \neq 0}^{n} \begin{cases} \Gamma \cdot L_i \cdot l_i & \mathit{if}\ \vec{L}^{o}{}_{i} \cdot \vec{S} \leq r \leq 1 \\ 0 & \mathit{else} \end{cases}$$

$$\Gamma = \left[ \frac{(\vec{L}^{o}{}_{i} \cdot \vec{R}^{o}{}_{refl} - r) \cdot (\vec{L}^{o}{}_{i} \cdot \vec{R}^{o}{}_{refl} + r - 2)}{L^{o}{}_{i} \cdot (1 - r)^2} \right]^2$$

$$A = (\vec{R}^{o} \cdot \vec{N}o \cdot R_a + R_d) \cdot \Psi$$

$$S = \vec{R}^{o} \cdot \vec{N}^{o} \cdot S_a + S_d$$

$$R_{refl} = \begin{cases} D_{refl} \cdot I'(\vec{R}_{refl}) & \mathit{if}\ d \leq d_{\max} \\ 0 & \mathit{else} \end{cases}$$

$$R_{refr} = \begin{cases} D_{refr} \cdot I'(\vec{R}_{refr}) & \mathit{if}\ d \leq d_{\max} \\ 0 & \mathit{else} \end{cases}$$

**Figure 5**: *The equations of the illumination formula*

```
#!/bin/sh -xe
sed -e "s/#GEO/$1/g" "s/#HIG/$2/g" <$3.geo >$3.$1.inf
bildcomp.int.1.6 -i $3.$1.inf -o $3.$1.pix -p /dev/null
compress $3.$1.pix
rm -f $3.$1.inf
```

**Figure 6**: *Example of a shell script for rendering one image*

```
cd newton.dir; exec newton.run 000 92.00 newton
cd newton.dir; exec newton.run 001 91.86 newton
cd newton.dir; exec newton.run 002 91.44 newton
  ...
cd newton.dir; exec newton.run 025 12.00 newton
```

**Figure 7**: *shell script for rendering a sequence of images*

jobs This subdirectory contains all jobs waiting for execution. Each job is a file containing a shell-script to be executed which starts the VERA renderer. The files in the UNIX operating system are sorted alphabetically. The jobs are processed in this order yielding a simple priority scheme by naming conventions.

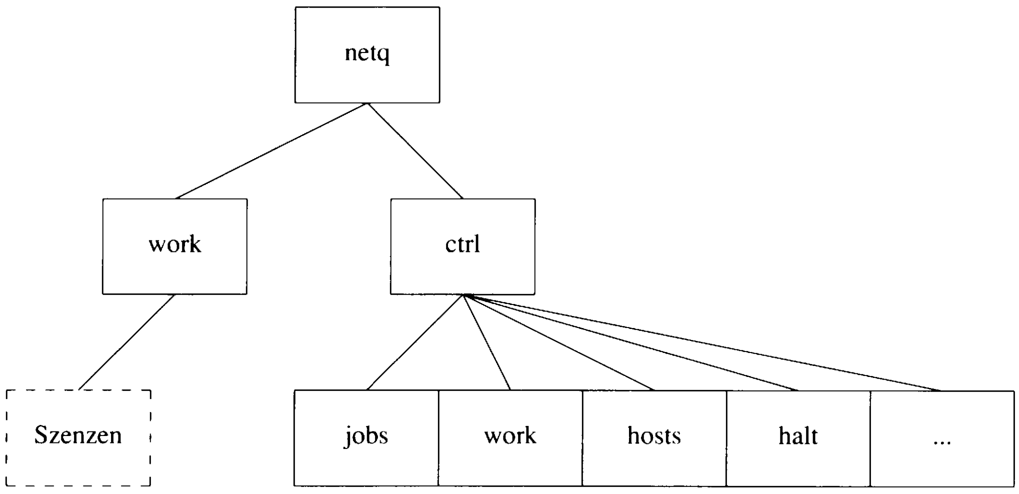


**Figure 8**: *The directory structure of the directory queue*

| | |
|---|---|
| work | Contains all jobs currently running. When a client picks up a job it moves the file for this task from `jobs` to `work`. |
| done | All files are moved to `done` when they have been processed successfully. The directory `done` is used for documentation and error recovery. |
| bad | All jobs leading to an error which cannot be solved automatically are moved to `bad`. This happens if the job goes down or is killed by a `kill -9` command. After error analysis such a job can be moved to `jobs` again for another attempt. |
| hosts | In the subdirectory `hosts` all work stations are mentioned that are ready for work. For each work station a file with the stations name exists, in which the status of the machine is given (executing, dispatching, idle, ...). |

There are further directories, necessary for protocoling, statistics, and error recovery. In one directory we find the most recently executed jobs. This directory allows the controlling demon to determine whether a machine has gone down. In another directory we can put all hosts planed to be stopped for administrative purposes. Finally statistics on job interruptions and run times are protocolised.

New jobs are initiated by the command `nq file` on any machine, with `file` containing the shell script for rendering.

On each *client* a demon (background process) is installed during boot time, by the command `netqd [-u time] [-r] [-w workdir]`. The `-u` option is used to determine the period in seconds for picking up a job after the last input of interactive users. The `-r` option is used for machines without having mounted the netqueue directory. In this case the mount will be done on the flight. The `-w` option will determine the name of the netqueue directory.

Our approach differs from other implementations, known in the graphics community [Pet87] in various points:

- In contrast to Apollo's strategy with the raytraced computer animation *Quest*, our network is totally automatised.
- item A centralised method implemented by Matthew Merzbacher, UCLA, assumes the availability of computers at fixed time periods. Interruptibility is hard to carry out with this approach.
- The method of Mike Muuss of the Army Ballistic Research Lab using "chunks" of scan lines leads to relatively high overhead and would have made necessary severe changes in our rendering software.
- Paul Heckbert's solution at NYIT is totally decentralised and uses disc capacity on all machines. Further he mentioned problems with synchronisation.
- John W. Peterson in Utah had a heterogeneous network. This required even more overhead in controlling the different machines, in particular in case of system crashes.

## 4. Results and Conclusion

Occursus cum Novo consists of 7550 frames. Due to still pictures and cyclic replication only about 4000 of them were really calculated. The frames were calculated at a resolution of 786 x 576 leading to a data volume of 9 Gigabytes. Using UNIX-compress, data was reduced to 2.2 Gigabytes, with a majority of the images having a compression factor of 25 %. Figure 9 gives a survey on the calculation. Almost three years CPU time on 68020-based machines were required for rendering, delivered by the network in about two months actual time. An overall average of 90% of the total computational power of the network was used for **Occursus Cum Novo**. This rises from the fact that the work stations were mostly used interactively by other users, and not for background number-crunching jobs.

| What | Sum |
|---|---|
| tasks (frames and previews) | 6454 |
| dispatches | 9544 |
| interrupted runs | 3090 |
| cpu-hours (rendering) | 23307 |
| cpu-month (rendering) | ≈32 |
| actual number of machines | 22-34 |

**Figure 9:** *Statistics of the calculation in the network.*

The average data production was about 70 MB a day. Approximately every other job was broken at least once, thus showing the necessity of interruptible rendering. About 70 jobs, *ie* 1%, were recognised 'bad' and could be found in the appropriate subdirectory. About 15 % of them were caused by bugs in the VERA renderer. The other bad jobs were originated in machines unexpectedly switched off by users, and transmission problems on a newly installed fiber optic cable.

The network queue works completely automatised. The only manual operating requirements are saving frames on magnetic tapes and error handling for jobs in the directory `bad`. The magnetic tapes posed a problem on weekends. It occurred that processing stopped due to the file system being full.

The VERA language for static scenes and the general purpose language Pascal for animation are simple enough to specify complex motions quickly, at least for people somewhat experienced in writing programs. However, tools for visually judging the quality of a design are necessary, in particular for such time-consuming rendering techniques like ray tracing. The Occursus cum novo environment offers the possibility of previewing. Previewing is done by raytracing a sequence of frames at low resolution, typically 156 x 115 pixels, and reduce it to bitmap quality. This requires colour reduction and dithering. Several methods with different properties were implemented for this purpose. The frames are held in the main memory of a work station, and written at a repetition rate to 25 frames per second onto the bitmap display. The previews are processed on the network queue with preference. Preference can easily achieved by the sorted arrangement of the files in a UNIX directory.

## 5. Bibliography

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. "Arts: Accelerated ray tracing system" in *IEEE CG&A*, pages 16–25, April 1986.

[Gla86] A. S. Glassner. "Space subdivision for fast raytracing" in *IEEE CG&A*, pages 15–22, October 1986.

[Pet87] John W. Peterson. "Distributed animation summary" in *USENIX Fourth Computer Graphics Workshop*, 1987.

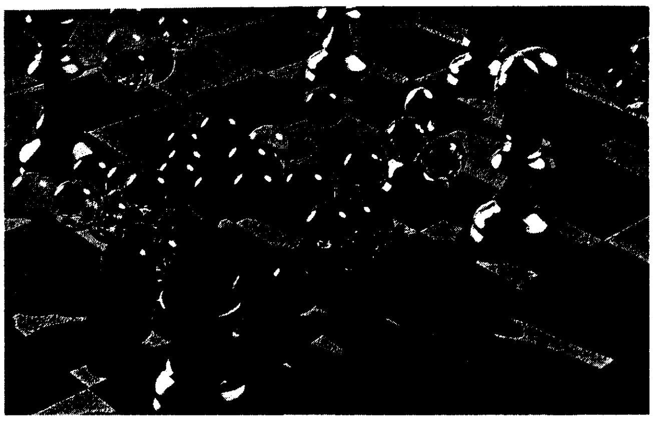
[SML87] Alfred Schmitt, Heinrich Müller, and Wolfgang Leister. "Ray tracing algorithms – theory and practice" in *NATO ASI Series: Theoretical Fundamentals of Computer Graphic*, volume F 40. Springer Verlag, July 1987.

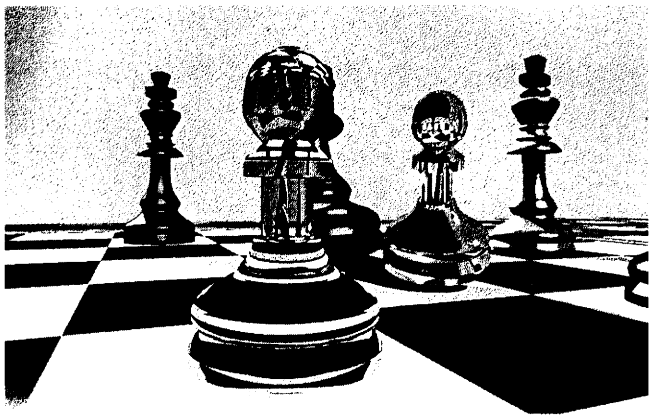
[Sun86] Sun Microsystems. *Network file system protocol specification, Revision B*, February 1986.

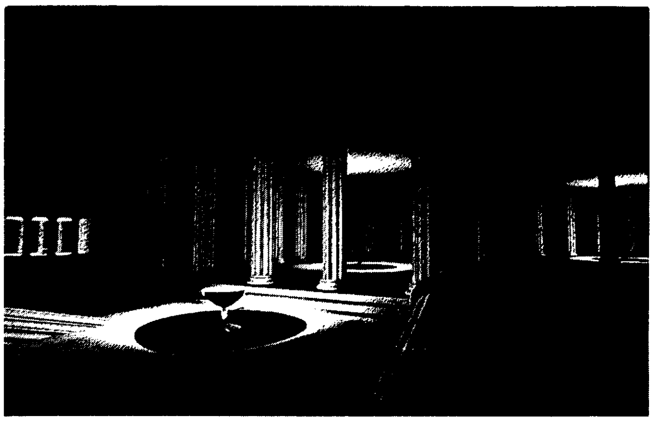
[Whi80] Turner Whitted. "An improved illumination model for shaded display" in *CACM, 23:343–349*, 1980.

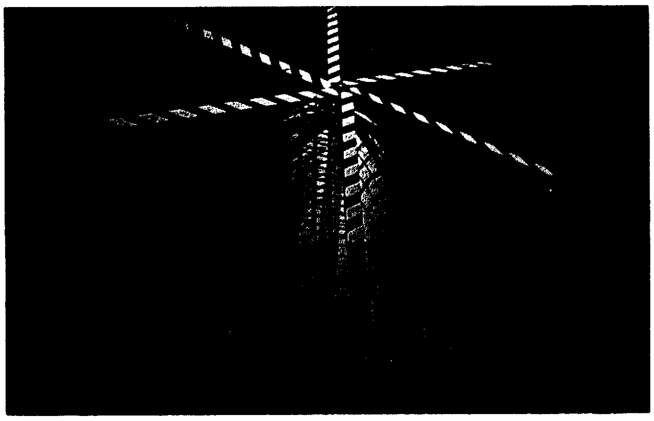
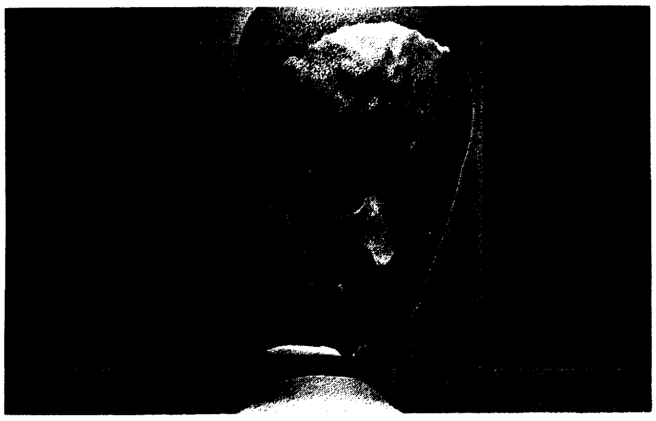
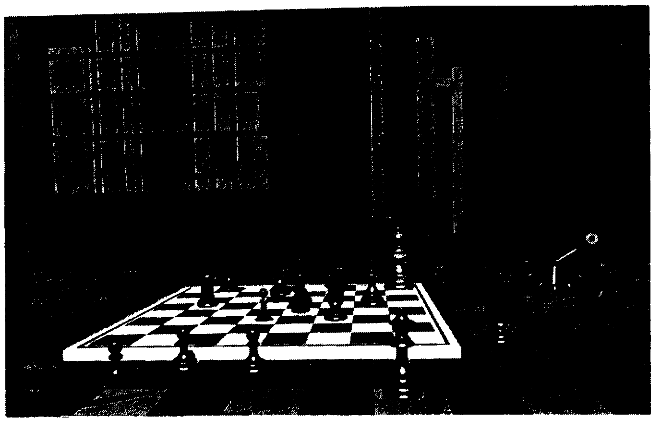
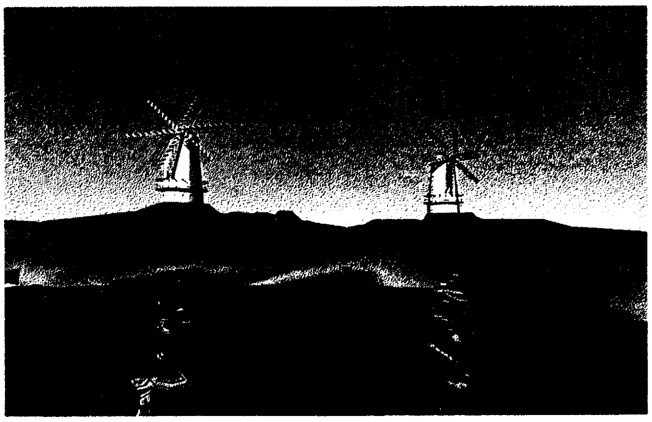
## 6. Image Section

The images on the following pages are taken from the computer animation Occursus cum Novo. The reproduction technique for black and white images was developed at our institute by Markus Pins.







# Designing A Virtual Toolkit for Portability Between Window Systems

*Marc J. Rochkind*
*Carol J. Meier*

Advanced Programming Institute Ltd.
Boulder, CO
*meier@boulder.colorado.edu*

*ABSTRACT*

Today's application developers wishing to take advantage of modern graphical windowing technology are faced with the difficult decision of which platform(s) to develop their application for. While all of the popular window systems support certain basic functionality (windows, events, menus, mice, ...), the implementations vary widely.

The Extensible Virtual Toolkit (XVT) is a high-level interface that allows graphical, interactive applications to be easily ported to various window systems, such as X-11, MS-Windows, OS/2 Presentation Manager, and the Macintosh. Behind the common interface there is a separate implementation in the form of a C object library for each host system.

This paper describes the design principles behind XVT and its key programming features. It then reviews the main problems in creating an implementation for X and explains our short-term solutions. Also discussed are plans for more thorough long-term solutions using industry-standard toolkits.

## 1. Today's Programming Challenge

The latest user-interface technology has issued a real challenge to application developers already struggling to achieve an acceptable degree of portability. Old fashioned, easy to program command line interfaces have been replaced with menus, mice, windows and icons. Applications need to run on an increasingly diverse set of systems. Furthermore, programmers are asked to do this using complicated toolkits that differ considerably from one window system to the next!

Portable programs are certainly nothing new; software designers have struggled for portability since the earliest days of computing. Originally, starting in the late 1950s, portability concerns focused on the standardization of programming languages, Cobol and Fortran, across different machine architectures and operating systems. By the mid-1970s, with C and UNIX, portability had been extended to system programming languages and operating systems. The development in the mid-1980s of X and NeWS has proved that even windowing systems can be portable.

Traditionally application portability has meant writing programs that did not depend on the underlying hardware or operating system. With the compiler, operating system, and window system implementors doing most of the work, application programmers lucky enough to restrict themselves to standardized environments, a combination, say, of UNIX and X, have gotten a free ride. For them the portability issue is essentially solved. But computers running proprietary system software, mainly personal computers, still account for a huge market. Application developers who want to go after it still have to deal with portability across operating environments, only one of which is UNIX and X. Today application portability means writing programs that avoid dependencies on the underlying window system as well.

Fortunately, nearly all of today's computers have C compilers, so at least the language problem has been solved; the advent of ANSI C compilers will improve the situation even more. But for many developers portability across operating systems, and certainly across window systems is still an obstacle. The current generation of popular PCs, Macintoshes and IBM compatibles, will probably never run UNIX and X as their primary systems. And even if they do, it won't be for years. For now, popular graphical window systems (MS-Windows, Macintosh, X Windows, NeWS, GEM, etc.) all support considerably different abstractions, as well as implementations of windows, events, fonts, menus, icons and other resources.

## 2. Portability between window systems?

Developers have three options: They can target UNIX and X systems, and ignore the rest; they can recode their programs for each environment; or they can write to a common interface that hides the differences between the underlying operating and window systems.

Clearly, the third choice is the most attractive. The solution to portability lies in raising the level of abstraction. Just as operating systems and high level languages came along to insulate programs from hardware differences, a new software layer is needed to insulate today's applications from window system differences.

But is portability across window systems possible? And, if it is, is it practical? Are the various window systems and user interface toolkits similar enough in their basic capabilities? Will portable applications be fast enough? Small enough? Will too many features have to be sacrificed? After all, history has shown that portability becomes accepted not when it can be first be demonstrated, but when it first becomes reasonably competitive with non-portable approaches in terms of efficiency. The Extensible Virtual Toolkit (XVT) is, as far as we know, the first commercially available C library for portability across window systems. It was first shipped for the Macintosh and MS-Windows in January, 1988, and there is now substantial evidence that such portability, at least across those environments, is indeed practical. We are now completing XVT libraries for OS/2 running Presentation Manager (PM), for X.11, and for a character-based window system [Roc88] (Fig. 1).

OS/2-PM is a successor to Windows, so that port presented no conceptual difficulties, just lots of tedious details to be changed. The X.11 job was tougher, and is the subject of most of the rest of this paper.

The character-based implementation by necessity has less functionality; XVT calls such as `draw_oval` aren't implemented at all, and others, such as set_font, are only partially implemented (you can set the text style to bold, for example, but you can't set the face name or point size). Yet, we were surprised at the number of XVT functions that made sense as character-oriented operations. Event handling, window creation and manipulation, menus, dialogs, and printing all have equivalents in the character-based world. They operate with a choice of coordinate systems: A 25-by-80 system in which the characters are numbered, and a 200-by-640 system in which the mathematical lines between the pixels are numbered and the characters are all 8-by-8 pixels in size. The pixel coordinate system makes porting to a graphical system easier, whereas character coordinates are more natural for those already used to character displays.

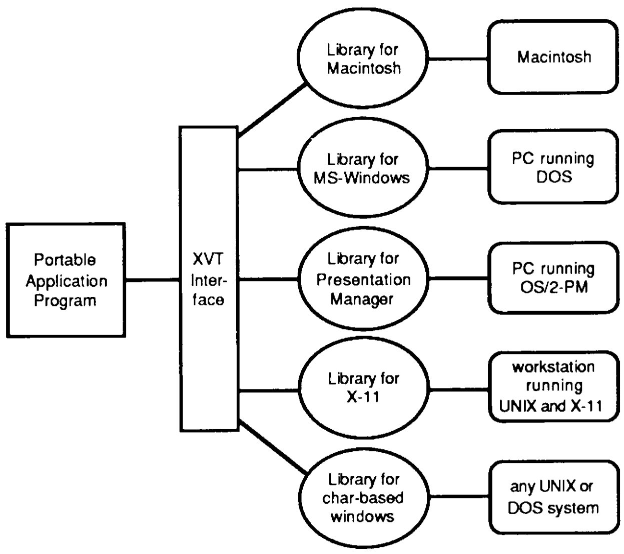


**Figure 1**: *The XVT interface shields a portable applicationfrom details of a particular window and operating system. This picture was created with XVT-Draw, a portable drawing program written with XVT.*

## 3. Ground rules for the Design of XVT

In designing a portable window system interface it's important not to aim for too low or too high a level of abstraction. At the extreme low end you have C itself, which offers portability but hardly qualifies as a window system. At the high end you have application programs like PageMaker, which runs on several window systems. What we want is in between: Its function calls should look like those of a window system and toolkit (with events, windows, graphics, fonts, a clipboard, and so on), but it should be simpler than any of the underlying systems on which it runs.

So we established some ground rules to ensure that XVT would be at the right level of abstraction, would solve the right set of problems, and would support the right kinds of computers and window systems:

- It must be practical to program real applications, not just toys, and those applications must include highly graphical ones. This can be tested only by releasing the package commercially; a few examples coded by XVT's developers don't prove anything.
- At least X, MS-Windows, OS/2-PM, and the Macintosh must be supportable, and, perhaps eventually, NeWS. These environments are sufficiently dissimilar that abstractions designed to bridge them are likely to work for other systems, too.
- The XVT interface must be thin; that is, it must use whatever the host window system and toolkit offer for window management, menus, resources, dialogs, and so on. XVT must not be yet another window system that interfaces with the hardware or requires vast amounts of memory.
- XVT programs must be written in ordinary C, with no special pre-processing or interpretation. That is, XVT is just a library and some header files, not some new programming environment.
- On a given target machine, the application must look like it belongs there, with a look and feel that's compatible with native applications. This is crucial in the Mac community, less so in the others. But it is likely to become increasingly important even for X.
- No conditional compilation or execution that depends on the native toolkit should be required in application programs. Portability must be achieved through abstraction, not through multiple paths.
- Resources (precompiled definitions for menus and dialogs) don't have to be virtualized. They can be rewritten for each native system. (Portable resources are probably possible too, but this restriction reduced the labor to get the initial version of XVT completed. We do however have a utility to automate translation of Macintosh resources into MS-Windows resources, which will undoubtedly be extended.)
- The interface must be extensible, in that it must be possible for applications to access underlying window system features. This reduces portability, of course, but it ensures that application developers won't find themselves boxed in if they need a few features that aren't part of the XVT model. Surely 90% portability is better than none!

These ground rules aren't the only viable ones. With different rules other solutions to portability across window systems are possible, and under the right conditions each of them is optimal. For example, if stylistic issues and the need for graphics aren't important, then an ordinary classical UNIX-style program (with an "argc, argv" style user interface) is portable, since all of the target systems (even the Mac) support such programs. Without the requirements for C, for a thin interface, and for applications that look native, Smalltalk-80 is a good portability solution.

There isn't room in this paper to describe the XVT interface in detail. In the next section I'll try only to mention its key features and to indicate its scope. Then I'll show the obligatory example program that displays (what else?) "Hello World!". Finally, I'll discuss the main challenges we faced in implementing XVT for X. The remainder of this paper assumes general familiarity with X [Nye88].

## 4. A quick look at XVT

It's convenient to look at XVT, or any window system, in terms of four levels. The *operating system* level includes device support, networking, task management, memory management, and a file system. The *window* level includes windows, events, graphics, fonts, cursors and carets, and resources. The *user-interface toolkit* level includes support for menus, dialogs and alerts, window controls (i.e., scroll bars), printing, and the clipboard. Finally, the *style* level includes look-and-feel policies and desktop management.

For the PC window systems these four levels aren't clearly demarcated. The Mac is the most monolithic: Its operating system doesn't even have a name (some call it the "finder," which is like calling UNIX the Bourne shell). MS-Windows and Presentation Manager do run on distinct operating systems, but neither they nor the Mac distinguish between the window and toolkit layers. These systems also support a particular look and feel; although developers can stray from it if they wish (there's more than enough flexibility), Apple and Microsoft strongly discourage it.

X is much more modularized. X itself handles only the window layer, it was never intended to do more. The operating system level is UNIX. The toolkit can be chosen from an ever-growing list (e.g., Xt, Sony Widgets, DEC Windows). A consistent style is optional: Each application can have its own, or a particular look-and-feel, such as Open Look, can be mandated.

Because XVT is designed to support real applications, which need support on all four levels, its abstractions have to encompass a wider scope than only the window layer.

The PC operating systems are weak relative to UNIX, so, to ensure portability, XVT has to support memory allocation (to handle large blocks outside an application's local heap) and file operations, such as changing directories and reading files with line-ending sequences that include almost all permutations of carriage return and linefeed. Even ANSI C functions such as `ctime` have to be included in the XVT interface, because the current Mac compilers don't match the standard, or even the UNIX C library, very well.

At the window level XVT includes features to create and destroy windows; to change or query their position, size, title, activation, and clipping rectangle; to handle events; to draw graphics, with various pens, brushes, and transfer modes; to select fonts and measure text; to manipulate the cursor and caret (insertion point); and to access resources. The XVT feature list in this layer essentially parallels that of X, but X is much richer and more complicated.

XVT includes user-interface toolkit features as first-class citizens, as the Mac, MS-Windows, and Presentation Manager do, rather than as an optional add-on, as X does. There are functions to access or modify menus; to put up modal or modeless dialogs and alerts, with push buttons, radio buttons, check boxes, editable and static text, list boxes, and scroll bars; to manage user-activity during dialog processing; to put text, pictures, and application-defined objects onto the clipboard, and to get them off; and to print.

At the style level XVT needs to do little, as most look-and-feel issues are up to the underlying window system. However, XVT directly supports standardized dialogs for choosing file names, and an elaborate online help system. Guidelines in the XVT programmer's manual help the designer code resources so as to follow the style of each host system while maintaining a uniform interface to the program, which must be invariant across systems.

Physically, XVT consists of a single interface, `xvt.h`, that's identical across all systems. It consists of about 200 functions along with type definitions for abstract objects such as windows and fonts. There's a separate implementation of this interface for each host system in the form of an object library. In the case of X, a client program is linked with the library `xvt.a`. Normally, a client makes no Xlib calls for itself, most of its XVT calls result in Xt or Xlib calls by XVT. XVT makes no use of any X internals, nor does it require any modification of X.

Conceptually, we treat X as a black box (like the Mac ROM) and don't attempt to exploit the fact that we have access to its source code. We don't supply X for workstations, but rather run on the port provided by the hardware vendor.

Extensibility hooks allow an application to access the underlying window systems native event stream if necessary. This is done at the cost of some portability, but has the advantage that an application with a special requirement is not precluded from using XVT.

The preceding paragraphs have only scratched the surface of XVT, just to make the point that XVT is broad, in that it spans all four levels, and deep, in that its feature set is reasonably complete (and growing). More details on XVT itself are in [Roc87].

## 5. Hello World in XVT

It's traditional when writing about window systems to include a "Hello World!" program. The following short but entirely complete XVT program is our competitive entry. If anybody comes up with a shorter program than this for any window system, we're prepared to fight back by eliminating the feature that allows the user to quit.

In case you are rummaging around through this paper looking for the rest of the program, rest assured that it is all right here. Since XVT takes care of most of the tedious details of window programming, this application simply needs to supply code to respond to an update event and the quit selection on the file menu. The XVT library supplies everything else, including the main function, creation of the application's initial window, and management of window movement, sizing and focus issues.

```
#include "xvt.h"
#include "xvtmenu.h"
void main_event(win, ep)        /* called when an event occurs */
WINDOW win;                     /* window affected by event */
EVENT *ep;                      /* pointer to event structure */
{
        RCT rct;
        switch (ep->type) {
        case E_UPDATE:
                get_client_rect(win, &rct);
                set_pen(&white_pen);
                draw_rect(&rct);
                draw_text(10, 100, "Hello World!", -1);
                break;
        case E_COMMAND:
                if (ep->v.cmd.tag == M_FILE_QUIT)
                        terminate();
        }
}
```

The appendix shows a more interesting program that allows the message to be changed (to "Goodbye World!") from the keyboard, with the mouse, or by choosing a menu command. The message appears in a custom-sized rounded-corner rectangle. The user can change font and text-style, and even open up multiple windows. Figure 2 shows this program running on a Mac. The same source can be compiled and run on an IBM-compatible running Windows or OS/2-Presentation Manager, or on a UNIX workstation running X.

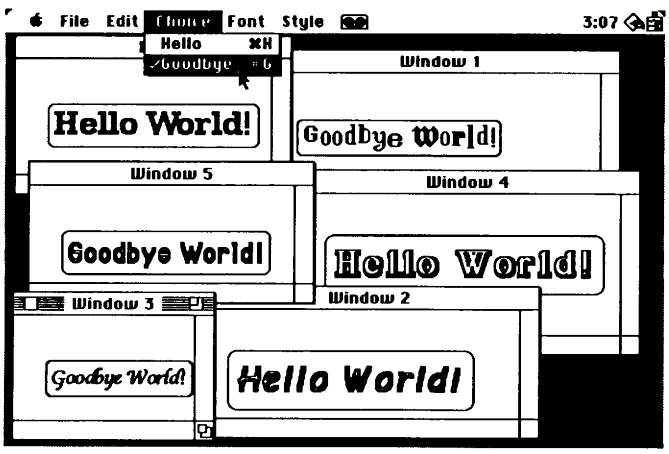


Figure 2: *Six hellos and goodbyes fromthe XVT example program shown in the Appendix.*

## 6. The X Implementation of XVT

One of the challenges we faced was to try to provide a smooth interface to graphical window system primitives, where the level of primitives varies highly both within a given system as well as across systems. This gives a certain texture to the chosen set of abstractions. In some areas, such as window creation, the abstractions must hide a lot of rough details from the application. In other areas, such as graphics, the abstractions are very close to the underlying primitives because graphics primitives have always been a fundamental part of windowing systems and they agree on the interfaces. Every system provides primitives to draw lines, rectangles and other shapes, so in this area the XVT layer is very thin and serves mostly to provide a consistent syntactic interface to these graphics primitives. The texture of the XVT implementation for X Windows is the subject of the next section.

We were pleased that some features, such as graphics, moved over as easily as we had hoped, and others (such our implementation of `stat`) could be skipped altogether. We were not pleased that other features, such as printing, turned out to be major programming projects. We'll break this discussion into the four levels outlined earlier: operating system, window, user-interface toolkit, and style.

## 6.1. Operating System Level

X is much richer in operating system features than are the PC window systems. More precisely, it's not X that has these features, but rather UNIX.

The PCs usually lack memory management hardware, and the Mac and DOS operating systems ignore it even when it is there, so they use a scheme involving doubly-indirect references to allow them to rearrange the heap without invalidating application-held addresses. In XVT the call `galloc` allocates a block and returns a handle; `glock` locks it and returns a pointer; `gunlock` frees the pointer (but retains the handle) so the memory manager can move the block if it needs to. Implementing these functions on UNIX is easy: `galloc` is a macro that calls `malloc`, `glock` is the identity function, and `gunlock` is a no-op. The only disadvantage of this implementation is that proper use of `glock` and `gunlock` can't be tested, an accidentally unlocked pointer would remain valid. (Programs that shouldn't work but do are even harder to test and debug than program that should work but don't!)

XVT's directory manipulation interface maps onto UNIX system calls easily. Its implementation was taken from the DOS version of XVT, since the UNIX and DOS file system calls are similar.

Most of the other operating system features in XVT (such as `ctime`) aren't needed at all on UNIX.

## 7. Window Layer

This is X's forte. We didn't anticipate any fundamental problems in implementing this part of XVT, and we found none. There were lots of niggling details to be addressed, however.

## 7.1. Window Manipulation

By implementing our own document window widget, XVT's `new_window` and `close_window` map well into `XtCreateManagedWidget` and `XtDestroyWidget`. XVT's other window manipulation functions (`move_window`, `set_title`, etc.) also map smoothly into X. XVT's double-line border style has no direct X equivalent, so an appropriate pixmap border is used instead.

To XVT the notions of frontmost window and holding the input focus are tied together, so its `set_front_window` call results in calls to both `XRaiseWindow` and `XSetInputFocus` on X.

## 7.2. Events

XVT has only 15 event types, most of which have direct equivalents in X, as shown in this table:

| *XVT Event* | *Explanation* | *X Event* |
|---|---|---|
| `E_MOUSE_DOWN` | mouse button down | `ButtonPress` |
| `E_MOUSE_UP` | mouse button up | `ButtonRelease` |
| `E_MOUSE_MOVE` | mouse moved | `MotionNotify` |
| `E_MOUSE_DBL` | mouse button double-clicked | [see text] |
| `E_CHAR` | keyboard character typed | `KeyPress` |
| `E_UPDATE` | window needs updating | `Expose, GraphicsExpose` |
| `E_ACTIVATE` | window gets/loses focus | `EnterNotify, FocusIn, FocusOut, LeaveNotify` |
| `E_KILL_WINDOW` | last event for closed window | [internal] |
| `E_VSCROLL` | vertical scroll bar action | [internal] |
| `E_HSCROLL` | horizontal scroll bar action | [internal] |
| `E_COMMAND` | menu command | [internal] |
| `E_CLOSE` | request to close window | `DestroyNotify` |
| `E_SIZE` | window being resized | `ConfigureNotify` |
| `E_FONT` | selection from font/style menu | [internal] |
| `E_QUIT` | request to quit application | [not used] |

Several XVT events (`E_KILL_WINDOW`, `E_HSCROLL`, `E_VSCROLL`, `E_COMMAND`, and `E_FONT`) are generated internally, and don't require direct support from the underlying window system. For example, an `E_COMMAND` event is generated when the user chooses a menu command. Since the menu manager is inside of XVT, it simply passes the event on to the application. Of course, the menu manager handles lots of events while the user is interacting with the menu, but those are hidden from the application.

The `E_QUIT` event only applies to certain systems that use a two-phase shut-down: Applications are first polled for approval, and then ordered to terminate if all are willing to do so. X doesn't use that protocol.

The double-click event (`E_MOUSE_DOUBLE`) isn't supported directly by X (or by the Mac), but it's easily simulated because each button event includes a position and time. A double click is inferred when a button press follows a button release within a sufficiently small interval of time and area of the screen. The code to make this inference was taken from the Mac version of XVT. Note that if the time of the button event weren't passed in from the X server it would be infeasible to infer double clicks in the client, because time measurements there don't correspond accurately to the user's actions.

When XVT passes an `E_UPDATE` to an application it doesn't include information about what part of the window needs updating. Instead, the application calls the Boolean function `needs_update` to test each candidate rectangle (usually bounding boxes of graphical objects or blocks of text). XVT automatically sets the clipping region to the part that needs updating, so the application can just draw the entire window if it wants to, without calling `needs_update` at all. For the Mac and Windows, XVT's job is easy because those systems coalesce pending update events into a single event and arrange for the proper clipping. While Xlib's support for exposure events is at a lower level, the Xt toolkit allows widgets to specify that exposures should be compressed, thus allowing an implementation similar to the Mac and Windows.

## 7.3. Graphics

XVT uses a pen and brush model to draw shapes; the pen specifies the perimeter, and the brush describes the interior. These correspond to X's line and fill style in a graphics context. XVT doesn't maintain a graphics context as an object separate from a window, so to an XVT programmer it is the window's pen and brush that are set.

In XVT a single call (e.g., `draw_rect`) is used to draw both a filled and unfilled shape. If only the perimeter is wanted, a hollow brush is specified. For the X implementation, each shape routine tests the brush and calls the perimeter or filling variant of the appropriate X function. For example, if the brush is hollow, `XDrawRectangle` is called; if not, `XFillRectangle` is called. The Mac implementation uses the same method.

All but two of the XVT shapes map directly into X shapes. The exceptions are `draw_roundrect` and `draw_pie` (which draws a wedge). Both are built up in pieces by calling the X functions `XDrawSegments` and `XDrawArcs` (and their corresponding filling variants).

The XVT function `draw_picture` draws a picture that's normally stored as a collection of discrete objects, rather than a bitmap. This permits the picture to be rendered at the full resolution of the output device rather at the resolution of the device on which it was created. Both the Mac and Windows support such pictures (PICTs and Metafiles, respectively), but X does not. As a placeholder, we've used pixmaps to implement pictures, we'll implement our own object-oriented structure when time permits or, if an industry standard format emerges, we'll use that. Note that the obvious candidate here, PostScript, won't work because only those X implementations that support Display PostScript can draw such an object on the screen.

## 7.4. Fonts and Text

Selection of a font in which to output text is one of the most difficult XVT features to virtualize. The problem is that in most window systems the important attributes of a font are specified by a name (e.g., "fr3-25") which is meaningless in and of itself. Only MS-Windows has a font mapper that can select a font based on its characteristics, such as serif or sans-serif, weight, size, quality, and so on.

XVT sidesteps this problem by defining a custom font and style menu for each system on which it runs. The contents of this menu are hidden from the portable application; when the user makes a selection, an `E_FONT` event is passed to the application along with an abstract FONT object (whose internals are secret). To draw in that font, the FONT is passed as an argument to the function `set_font`. This scheme works fine for fonts that the user selects, but it doesn't work for fonts that the application wants to pick. That is, XVT provides no way to synthesize a locally meaningful FONT object from a list of portable attributes. Some of the most common needs are met by three predefined FONTs, `big_font`, `normal_font`, and `small_font`. We're working on a more general solution to the problem of synthesizing fonts that will probably employ a font mapper of our own design.

Meanwhile, the standard XVT font menu for X contains several of the most common font names. The user can change the list by editing the `.Xdefaults` file.

### 7.5. Cursors and Carets

XVT offers a choice of five predefined cursor shapes: arrow, I-beam, cross, plus (thicker cross), and waiting. All of them (and lots more!) exist in X's cursor font and can be set with a call to `XCreateFontCursor`.

In XVT terms a caret is a blinking line or solid block that indicates the insertion point for text. An application positions the caret with a call to `caret_on` and suppresses it with a call to `caret_off`. MS-Windows supports carets directly and handles the blinking automatically. The Mac doesn't, so XVT blinks it itself by XORing a small rectangle at fixed intervals of time. This works, in the grand tradition of PCs, because the Mac is a single-user machine with only rudimentary multitasking (the foreground application gets all the CPU cycles it wants). X lacks support for carets and one can't be blinked accurately or efficiently from a client process, so a non-blinking inverse rectangle is used instead. This works fine for character-oriented programs like Xterm, but WYSIWYG systems prefer to put the caret between the letters, not over a letter. A thin caret that doesn't blink is too hard for the eye to pick up rapidly. We're still pondering what to do about this.

### 7.6. Resources

Resources on the Mac and Windows are used more generally than those in X, primarily because memory is usually tight on those systems and loadable-on-demand resources provide an efficient place to store strings, menu definitions, dialog box layouts, and the like. Both systems come with elaborate dialog editors that allow an application designer (not necessarily a programmer) to create a dialog layout interactively. The design is then stored as a resource along with the executable image and is accessed at run time by a dialog manager. Aside from economizing on memory, resources also allow for late binding, which makes it easy to alter the design of a menu or dialog or to translate it into a foreign language.

We've designed a simple resource manager for X that supports just the XVT resource-related functions. These resources aren't kept in the `.Xdefaults` because that's under the user's control, and XVT's resources are part of the application. The same resource manager is used in the character-based version of XVT mentioned earlier.

## 8. User-Interface Toolkit Layer

XVT's facilities in this layer support menus, dialog boxes, the clipboard, and printing. Only the clipboard is directly supported by X itself. Various toolkits have been developed to run on top of X to support menus and dialogs.

We had two constraints on our design of the X version of XVT that affected our choice of a toolkit:

- The toolkit must be universally available, wherever X is.
- It must be supported by the workstation vendor (certainly not by us!).

When we set down our plans the Xt toolkit (intrinsics only, not the Athena Widget set) came closest to meeting these criteria, but that is much less robust than we would like. Several promising toolkits were on the horizon, such as Open Look, DEC Windows, Sony Widgets, and Andrew, but none of them were available to us or to all but a few X users. We declined to postpone our port of XVT or to get involved with installing and debugging a public-domain toolkit ourselves.

We decided to pursue three lines of development. First, we would implement as much of XVT as Xlib and the Xt toolkit intrinsics could handle, and make that available to our customers as a subset implementation. Second, we would design our own printer driver. Third, we would implement our own toolkit widgets based on the Xt intrinsics to support just the features that XVT needs.

We consider these initial toolkit projects to be a stopgap only, so we didn't want to put too much effort into them, and we didn't worry about the aesthetics of the user interface. We plan to produce a final version of XVT for X as soon as the chaotic toolkit area settles down. That version will most likely use AT&T's Open Look X-based toolkit. There will be another port for the toolkit chosen by the Open Software Foundation. We'll begin those ports when these toolkits are up and running in our shop, it's too hard for us to start earlier (we already tried using the viewgraphs in our code, but the linker complained).

The next few paragraphs discuss some specifics in our design of the XVT toolkit level for X.

## 8.1. Menus

XVT needs very little in the way of menus. It uses a two-level menu hierarchy inspired by the Mac menu bar, although the top level need not be a bar. We're implementing a simple menu manager with slide-off menus.

The Mac and Windows menu manager also handle keyboard shortcuts that allow a function key or control-key combination to be used as a synonym for a menu item. However, how the command is invoked is hidden from an XVT application, so there was no requirement to implement that feature for X right away, and we put it off.

## 8.2. Dialogs

XVT expects the system on which it runs to support a sophisticated dialog manager that takes over user interaction when a dialog is entered. It calls an application-designated callback function whenever the user does anything notable (such as pressing a button). Dialogs themselves are described by *control lists* of *control items*. Each item consists of a type code (button, check box, edit field, etc.), bounding rectangle, text string, activation flag, and a few other attributes. When a dialog is brought up the dialog manager reads the control list from the resource file and builds a displayable box containing the specified controls. It then supervises the user's interaction while the focus is in the box.

As of January, 1988, we were leaning towards using Xt widgets to handle dialogs. On one hand we had already begun designing a character-based manager that had all the right semantics; adapting it for a graphical world would involve replacing the rendering and action functions for each control type (button, radio button, check box, static text, editable text, scroll bar, and list box). We knew that our own design would work, but we wanted to use the Xt system because it would give us a leg up on using the commercial toolkits later. We may end up initially with only a partial implementation of dialogs for X, with the intention of filling in the gaps later. Indeed, even the original XVT implementation for the Mac and Windows lacked scroll bar and list box control types.

## 8.3. Clipboard

XVT supports the placing of one object on the clipboard, in as many formats as the application desires. Ideally, an application should be willing to put both text and picture formats onto the clipboard, and to take either off. An application can also use one or more private formats, available only to itself or to other instances of itself. It's OK for an application to use only text or only pictures if one of them isn't meaningful (e.g., an electronic mail program might not handle pictures).

These facilities are supported by X itself, so no toolkit support is needed. X's selection mechanism works well, except for the lack of a standard object-oriented picture format. As mentioned earlier, we decided to use pixmaps temporarily.

## 8.4. Printing

Printing was our single biggest challenge in moving XVT to X. On the Mac, in Windows, and in XVT one prints by opening a special, non-displayable print window and then drawing on it with the identical drawing functions used for normal windows. It's extremely easy to print a document (containing text and/or graphics) that the user has composed in a screen window: You just open a print window and make like you got an update event for it.

Unfortunately, X doesn't work that way. It doesn't support printing at all, although most X implementations allow you to print the screen or the contents of a window. But you want 1200 dots-per-inch on your typesetter, not seventy-five!

We had no choice but to implement our own printer driver.

To make the job tractable, we did limit it to PostScript printers only. The interface is documented so that others can implement additional drivers. If a proper printing facility is ever added to X (perhaps part of one of the commercial toolkits), we'll just implement a "driver" that uses the standard printer interface and throw away our real driver.

Here's how our simple design works: In each XVT drawing call, such as `set_pen` or `draw_oval`, we branch according to whether the current window is a screen window or a print window. In the former case, we call the regular X functions. In the latter case we call a driver entry point. These are named after the corresponding XVT calls: `pr_set_pen`, `pr_draw_oval`, etc. Our PostScript driver simply issues appropriate PostScript code to perform each action. Debugging this driver is extremely easy with a utility called Lasertalk running on a Mac connected to a LaserWriter. Lasertalk allows you to view printer output in a window on the screen, without printing anything. The output is guaranteed to be authentic because the PostScript is executed in the printer itself. Debugging would have been even easier with Display PostScript, but, alas, we didn't have access to it.

## 8.5. Not Yet the Conclusion

The XVT implementation on X will work well enough to demonstrate the viability of our approach, but it isn't yet ready for prime time. Support in X and UNIX at the operating system and window layers is good, no major problems there. The user-interface toolkit level is still confused. We've taken some temporary steps to get XVT running, but we knew from the start that we would recode most of it later in 1989 or 1990 once the toolkits stabilize. Meanwhile, XVT does indeed run, and its now possible to write a portable program just once and run it on a Macintosh, IBM-compatible running Windows or Presentation Manager, or a UNIX-based workstation running X.

## 9. References

[Nye88]

Adrian Nye (Ed.). *Xlib Programming Manual.* O'Reilly & Associates Inc., Newton, MA, 1988. ISBN 0-937175-27-7

[Roc87]

Marc J. Rochkind, *Technical Overview of the Extensible Virtual Toolkit (XVT).* Advanced Programming Institute Ltd, Boulder, CO

[Roc88]

Marc J. Rochkind *Advanced C Programming for Displays.* Prentice-Hall, Englewood Cliffs, NJ, 1988. ISBN 0-13-010240-7

## 10. Appendix: Example XVT Program

The following is an example XVT program that can display either "Hello World!" or "Goodbye World" in one of several windows opened by the user. The choice of message can be chosen via the Choice menu, by double-clicking the mouse, or by typing a "g" or an "h". The font and style for each window can be independently chosen. Figure 2 (appearing in the body of this paper) shows a sample session.

As discussed in the paper, this program is portable without change to each environment on which an XVT library is implemented.

```
#include "xvt.h"                          /* standard XVT header */
#include "xvtmenu.h"                      /* standard XVT menu tags */
#define M_CHOICE_HELLO                  MAKE_MENU_TAG(MENU3, 1) /* tags for our Choice menu */
#define M_CHOICE_GDBYE                  MAKE_MENU_TAG(MENU3, 2)
#define STARTX                  30        /* x coordinate for message */
#define STARTY                  40        /* y coordinate for message */
typedef struct {
        enum {HELLO, GOODBYE} choice;     /* message to display */
        FONT font;                        /* font */
} DOCUMENT;                               /* contents of one document */
APPL_SETUP appl_setup;                    /* setup for initial window */
/*
        Function to associate a DOCUMENT structure with a new
        window.  Each window carries a long word of data that
        can be set and accessed by the application.
        We use it to hold a pointer to a "document," which for
        this simple example consists of only the message choice
        and its font and style.  More generally,
        a document structure would hold text, a picture, etc.
*/
static void new_doc(win)
WINDOW win;                               /* window to be initialized */
{
        DOCUMENT *d;
        if ((d = (DOCUMENT *)malloc(sizeof(DOCUMENT))) == NULL)
                fatal("Out of memory.");
        d->choice = HELLO;
        d->font = normal_font;
        set_app_data(win, PTR_LONG(d));
}
/*
        Function to destroy a document structure, to be
        called when a window is closed (assuming that a
        document appears in only one window).
*/
static void dispose_doc(d)
DOCUMENT *d;                              /* document to be destroyed */
{
        free((char *)d);
}
/*
        Function called by XVT to initialize application.  It
        selects the font for the initial window (std_win),
        makes sure that it's marked appropriately in the
        Font/Style menu, initializes the window's document,
        sets a check mark in the Choice menu, and enables
        the New item on the File menu.
*/
BOOLEAN appl_init()
{
        set_cur_window(std_win);
        set_font(&big_font, FALSE);
        set_font_menu(&big_font);
        new_doc(std_win);
        menu_check(M_CHOICE_HELLO, TRUE);
        menu_enable(M_FILE_NEW, TRUE);
        return(TRUE);
}
/*
        Function to handle update events.  The screen is painted white
        and then the chosen message is drawn inside a rounded rectangle.
        To fit the shape properly, the message is first measured in
        width and height, and a rectangle is initialized that fits the
        text with a margin all around that's equal to the descent.
        (Nothing magical about the descent - it's just a convenient
        amount that's proportional to the size of the font.)
*/
static void do_update(win)
WINDOW win;                               /* window to be updated */
{
        RCT rct;
        char *msg;
        int ascent, descent, width, corner;
        DOCUMENT *d = (DOCUMENT *)get_app_data(win);
        get_client_rect(win, &rct);
        set_pen(&white_pen);
        set_brush(B_WHITE);
        draw_rect(&rct);
        msg = d->choice == HELLO ?  "Hello World!" : "Goodbye World!";
        set_font(&d->font, FALSE);
        get_font_metrics(NULL, &ascent, &descent);
        width = get_text_width(msg, -1);
        set_rect(&rct, STARTX - descent,
                STARTY - descent,
                STARTX + width + descent, STARTY + ascent - 2 * descent);
        corner = (rct.bottom - rct.top) / 3;
        set_pen(&black_pen);
        draw_roundrect(&rct, corner, corner);
        draw_text(STARTX, STARTY + ascent, msg, -1);
}
/*
        Function to set check marks on the Choice menu.
*/
static void fix_choice_menu(d)
DOCUMENT *d;                            /* active document */
{
        menu_check(M_CHOICE_HELLO, d->choice == HELLO);
        menu_check(M_CHOICE_GDBYE, d->choice == GOODBYE);
}

/*
        Function to handle menu commands.
*/
static void do_menu(cmd)
MENU_TAG cmd;                           /* tag identifying menu item */
{
        char title[50];
        RCT rct;
        static int count = 0;
        WINDOW win;
        DOCUMENT *d;
        switch (cmd) {
        case M_FILE_NEW:
                set_rect(&rct, 100, 100, 300, 200);
                sprintf(title, "Window %d", ++count);
                if ((win = new_window(&rct, title, W_DOC,
                                TRUE, FALSE, FALSE, TRUE)) == NULL)
                        error("Can't create window.");
                new_doc(win);
                break;
        case M_FILE_QUIT:
                terminate();
                break;
        case M_CHOICE_HELLO:
        case M_CHOICE_GDBYE:
                if ((win = get_front_window()) != NULL) {
                        d = (DOCUMENT *)get_app_data(win);
                        d->choice = cmd == M_CHOICE_HELLO ?  HELLO : GOODBYE;
                        fix_choice_menu(d);
                        invalidate_rect(win, NULL);
                }
        }
}
/*
        Main application entry point.  After changing the message
        choice or the font, the entire client area of the window
        is invalidated so as to force an update.  On a font event
        (user selected from the Font/Style menu) the font associated
        with the document is changed and the menus themselves are
        modified to show the current font.  An E_QUIT event is
        generated by the system when it wants to shut down.  In
        phase one, each running application is polled as to whether
        it is willing to quit (our little application is, so it calls
        quit_OK).  If the vote was unanimously positive, in phase
        two each application is ordered to quit.  If any application
        declined in phase one, phase two is skipped and the system
        stays up.  (Not all window systems use this protocol.)
*/
void main_event(win, ep)
WINDOW win;                             /* window receiving event, if any */
EVENT *ep;                              /* pointer to event structure */
{
        DOCUMENT *d = NULL;
        if (win == NULL)
                win = get_front_window();
        if (win != NULL)
```

```
                d = (DOCUMENT *)get_app_data(win);
        switch (ep->type) {
        case E_MOUSE_DBL:
                if (d->choice == HELLO)
                        d->choice = GOODBYE;
                else
                        d->choice = HELLO;
                invalidate_rect(win, NULL);
                break;
        case E_CHAR:
                if (d != NULL) {
                        switch (toupper(ep->v.chr.ch)) {
                        case 'G':
                                d->choice = GOODBYE;
                                break;
                        case 'H':
                                d->choice = HELLO;
                        }
                        invalidate_rect(win, NULL);
                }
                break;
        case E_UPDATE:
                do_update(win);
                break;
        case E_ACTIVATE:
                if (ep->v.active && d != NULL) {
                                /* make menu state reflect document */
                        set_font_menu(&d->font);
                        fix_choice_menu(d);
                }
                break;
        case E_COMMAND:
                do_menu(ep->v.cmd.tag);
                break;
        case E_FONT:
                if (d != NULL) {
                        d->font = ep->v.font.font;
                        set_font_menu(&ep->v.font.font);
                        invalidate_rect(win, NULL);
                }
                break;
        case E_CLOSE:
                if (d != NULL)
                        dispose_doc(d);
                close_window(win);
                break;
        case E_QUIT:
                if (ep->v.query)
                        quit_OK();
                else
                        terminate();
        }
}
/*
        Function called by XVT just before terminating
        application.
*/
void appl_cleanup()
{
        /* no cleanup (e.g., files to be removed) needed */
}
```

# Transport Interfaces in SINIX Systems

*Norbert Richter*

SIEMENS AG,
Munich,
Communication and Information Systems Group
*mcvax!unido!sinix!richter*

*ABSTRACT*

Open systems and Open System Interconnection represent a major advance with mutual benefits for the software vendors, the system suppliers and the end user community. Modern design of open systems in information technology follows closely the Reference Model of Open Systems Interconnection as defined by the International Standards Organisation. In this model, the interface between layers 4 and 5 provides abstract transport service that is independent of the details of the underlying network implementation. Application programs access the transport service through an application programming interface. It is a most important aspect for writing portable higher-layer or end-user software intended to work across different networks as well as on various machines.

The X/Open transport interface XTI is defined to meet this requirement. It provides support for international and industry standards of transport services in UNIX systems. Using XTI, communication applications achieve the necessary independence from the application environment and the underlying networks.

This paper discusses the XTI implementation in SIEMENS' SINIX system.

## 1. Introduction

The contemporary system architecture for information processing in the commercial market is based on a three-level model. The topmost is the enterprise level, the medium is the departmental or workgroup level, the lowest is the personal level. These levels need to cooperate tightly through a networking infrastructure suited to meet the organisational requirements.

In order to establish this architecture in reality, two major building blocks are required: open systems carrying the applications which process the information, and open systems interconnection which enables information exchange between the open systems within and across the three levels of the system architecture.

The current dominance of proprietary systems has significant disadvantages. It fragments the market, thus discourages the independent software vendors from writing applications for a limited system base. The end users get locked into particular system environments in order to save the investments made into applications processing their data and training the staff using these applications in every day's work. Growth is usually only possible within the already established machine environment. To support growth, most computer suppliers do offer a broad range of processing power among their machines covering one or more orders of magnitude. But the sales potential for their machines is limited and the market fragmentation remains.

The concept of open systems offers a breakthrough. Open systems are built around a set of agreed standards for hardware and, more important, for software. In order to meet their business needs economically, end users can choose freely from open systems offered by different suppliers and move their applications between those systems without rewriting software or retraining staff. Independent Software Vendors find a considerably increased system base for software and thereby development of new applications becomes attractive. The application base in turn results in a higher potential market for open systems with benefits for existing and new system suppliers. Eventually this further reduces the investment by the users, thus this process is mutually reinforcing.

For corporate needs open systems are spread over national or international areas and thus require interconnection. Open Systems Interconnection has been of growing importance during the last decade and its standardisation is firmly established in international bodies like the International Standards Organisation.

Open Systems Interconnection is based on the well known reference model [ISO7498], which functionally decomposes the communication facilities into 7 layers interacting on a protocol and service basis. Its lower four layers deal with reliable transport of information between peer application entities – represented by the top three layers – in open systems across public or private networks.

The access to the functions for information transport – the transport interface – is of fundamental importance for applications intended to operate in an open system environment. It relieves the rapidly increasing base of applications from dealing with the details of an also growing variety of networking technologies and facilities.

This paper deals mainly with the transport interface as supplied with the SIEMENS' SINIX system. The next two sections contain basic considerations about two important standards for communication applications, the Common Application Environment and the Open Systems Interconnection. Then a brief discussion of the X/Open Transport Interface (XTI) follows and a comprehensive description of how XTI is implemented within the SINIX system. Finally, the last section deals with future direction of the XTI development.

## 2. Open System Standards for Communication Applications

The focal points in todays public discussions about information technology are open systems and their interconnection. They are built from a collection of well specified standards set by national and international standardisation bodies. Applications written against those standards can operate on the wide range of systems supporting the same specifications.

There are two important concepts for open systems, the Common Application Environment and the Open Systems Interconnection. The former is a standards platform that enables applications to be ported to different open systems without change. The latter is the framework for exchanging information among open systems.

### 2.1. X/Open Common Application Environment

To encourage the provision of standards within the UNIX systems of different manufacturers, many of the world's leading information systems companies have joined forces in the X/Open Company Limited and agreed on a set of standard interfaces, a Common Application Environment (CAE) based on the UNIX operating system [XPG3CAE].

The CAE defines interfaces between the application and its environment. It comprises application programming interfaces (API), programming languages, utilites and user interfaces, together with interfaces for interconnection of systems carrying the CAE. For any application developed in accordance with the CAE, porting into the CAE on another system reduces to transferring the source code to the target machine and recompling it with the appropriate compiler.

Thus the concept of the CAE offers source level portability.

### 2.2. ISO Open Systems Interconnection

The Open Systems Interconnection (OSI) is based on an international standard [ISO7498] set up by the International Standards Organisation (ISO). It provides a common basis for the coordination of standards development for the purpose of systems interconnection. In this concept, a system is a set of one or more computers and associated software, peripherals and physical transfer media, that form an autonomous whole capable of performing information processing and/or transfer. An open system with respect to OSI is one that obeys OSI standards in its communication with other systems. The means for the transfer of information are physical telecommunications media, like public networks. OSI is concerned with the exchange of information between open systems and not with the internal functioning of an individual open system. OSI is also concerned with the capabilities of systems to cooperate to achieve a common (distributed) task. This cooperation includes activities like interprocess communication, data representation and storage, process and resource management, security and program support.

The OSI Reference Model decomposes the communication facilities into 7 layers. Entities residing in these layers interact with entities in adjacent layers through a service interface, and with peer entities on the same layer in another open system through a protocol. An entity offers a service to the layer above by exchanging protocol elements with a peer entity using the service provided by the lower layer.

## 2.3. ISO Transport Service

The lower four layers in the OSI reference model together provide the Transport Service (TS) [ISO8072]. It is offered by layer four which uses a transport protocol [ISO8073] through the service provided by layer three. Two modes for the transport service are defined, connection-oriented mode and connectionless mode.

In connection-oriented mode, the transport service provides reliable, sequenced and transparent information exchange between applications represented by the top three layers. A logical connection is established between the peer entities prior to the exchange of information. It enables parameters and options to be negotiated that govern subsequent data transfer. In addition, it provides an identification mechanism that avoids the overhead of address transmission and resolution and a context in which successive units of information are logically related. The service primitives defined with connection-oriented mode are as shown in figure 1.

```
T-CONNECT.request T-CONNECT.indication
T-CONNECT.confirmation T-CONNECT.response
T-DISCONNECT.request T-DISCONNECT.indication
T-DATA.request T-DATA.indication
T-EXPEDITED-DATA.request T-EXPEDITED-DATA.indication
```

**Figure 1**: *Service Primitives in Connection Oriented Mode*

Connectionless mode supports data transfers in self-contained units – so called datagrams – with no logical relationship among these units. There is no negotiation of options, all the information required to deliver a unit of data is presented together with the data to be transmitted in one single service access with no relation to any other service access. With this mode it is possible that the same data unit is delivered not at all, multiply or out of sequence. In that case, protocols at higher layers detect data units lost, duplicated or out of sequence and recover them appropriately. The service primitives defined with connectionless mode are as shown in figure 2.

```
T-UNITDATA.request T-UNITDATA.indication
```

**Figure 2**: *Service Primitives in Connectionless Mode*

Thus the concept of the OSI, particularly the transport service offers application connectivity.

## 2.4. Local and Wide Area Networks

In the three level architecture for corporate information processing the sites at the enterprise level communicate usually over relatively long distances with sites at the departmental and/or personal levels. The networks used are often public wide area networks (WAN). The information exchange between the departmental and personal level for resource sharing and information retrieval often uses private local area networks (LAN). LANs and WANs have particular protocol stacks for layers one through three, common protocols are used from layer four and above.

A WAN may be defined by a packet switched data network (PSDN), an integrated services digital network (ISDN) or a point-to-point connection between end systems over a fixed circuit. The commonly used protocol stack is the CCITT recommendation of X.25 at layers one, two and three. For communication across a WAN the connection-oriented transport protocol [ISO8073] offering a connection-oriented transport service is the natural choice in the spirit of the OSI. Many system vendors, however, use proprietary transport protocols too, with these their systems are not open. But they need to be retained for compatibility reasons during a migration period.

A LAN is a privately maintained geographically strongly limited high speed network. Though there are protocol stacks defined by ISO and IEEE within the OSI framework for LANs, the UNIX user community has adopted the INTERNET protocol suite [RFC973], commonly referred to as TCP/IP (Transmission Control Protocol/Internet Protocol). The IP at layer three offers a connectionless mode of operation, while the TCP at layer four is a connection-oriented transport protocol with a connection-oriented transport service.

## 3. The X/Open Transport Interface

As shown in the previous section, even with the CAE and the OSI there are still some open issues to be solved: usage of proprietary and international standard transport protocols in public WANs and usage of industry standard protocol suites in LANs. To achieve application portability and connectivity across WANs as well as across LANs an application programming interface (API) is required that takes these issues into account. In absence of an agreed standard for such an API, X/Open has defined the X/Open Transport Interface (XTI) first published in the X/Open Portability Guide Issue 3 [XPG3XTI].

Since X/OPEN members are committed to support the X/OPEN standards, XTI will soon be available on the UNIX systems of different suppliers.

### 3.1. General Features of XTI

XTI offers uniform access to the transport service independent of the transport service provider used. In particular, it is independent of any specific UNIX derivative and specific implementations of the transport service. XTI is concerned primarily with the services based on [ISO8072] and optionally based on TCP/IP, but is in principle adaptable for use with other types of transport services. Thus applications implementing XTI are enabled to cooperate across networks without dealing with the details how the data transfer is performed and application connectivity is finally achieved.

XTI is defined in terms of function calls of the C programming language. These are collected into the so called transport library which is linked to the application program. Most of the calls are mandatory, some are optional. Any implementation of XTI has to enclose at least the mandatory function calls. Thus an application using optional calls will not work in an environment where the optional calls are not implemented. The mandatory calls are sufficient to cover the service primitives as defined in the OSI and TCP transport services.

XTI distinguishes between the transport service user (TS user, the application proper) and the transport service provider (TS provider, the entity providing the transport service). The TS user calls the functions as specified by XTI. The TS provider implements those functions in a way not visible to the TS user. Within a single system many TS users and several TS providers can coexist. TS users and TS providers meet at predefined transport endpoints to exchange service requests and service indications. Any transport endpoint offers access to exactly one specific TS provider, it is opened by a TS user prior to the first service request and closed explicitly by the TS user after the last service request or implicitly by the TS provider upon premature termination of the TS user.

A TS user requesting access to different TS providers has to open several endpoints, one for each TS provider. Also a transport endpoint can support only one established transport connection at a time. The transport endpoint is, once opened, identified by a UNIX file descriptor. Though not all operations available on file descriptors are defined for XTI file descriptors, they are still subject to the UNIX inheritance concept across process spawning (using fork()) and program execution (using exec()). Several TS users may in that way share the same transport endpoint. Once a transport endpoint is opened, several activities are possible. A commonly used sequence is for example the establishment of a connection to a peer TS users with subsequent bidirectional data transfer and final release of the connection.

### 3.2. Functions of XTI

An overview of the XTI functions gives figure 3, optional calls are in brackets.

XTI is used by TS users which constitute from UNIX processes. One process may simultaneously handle several transport endpoints. Conversely, several processes may share the same transport endpoint but have to synchronize themselves so as not to violate the transport endpoint's current state. The XTI function *t_sync()* supports sharing of transport endpoints by returning the current state of the transport endpoint and synchronizing the TS user and the TS provider. To the TS provider, all users of the same transport endpoint appear as a single user.

```
INITIALISATION AND MANAGEMENT
t_open()         t_close()
t_bind()         t_unbind()
t_sync()         t_look()
[t_alloc()]      [t_free()]        [t_error()]
[t_getinfo()]    [t_getstate()]    [t_optmsg()]
CONNECTION ESTABLISHMENT
t_connect()      t_rcvconnect()
t_listen()       t_accept()
CONNECTION RELEASE
t_snddis()       t_rcvdis()
[t_sndrel()]     [t_rcvrel()]
DATA TRANSFER
t_snd()          t_rcv()
t_sndudata()     t_rcvudata()      t_rcvuderr()
```

**Figure 3**: *Overview of XTI functions*

XTI supports two modes of service: connection-oriented mode and connectionless mode. These correspond directly to the modes of the ISO TS as discussed previously.

Both modes allow synchronous or asynchronous operation. In the synchronous mode, the XTI primitive (*eg* to receive data) waits for a specific event (*eg* arrival of data) before returning control. While waiting, the TS user cannot perform other tasks. Synchronous mode is the default mode, it is used by TS users that maintain no other I/O source. Calls blocking in synchronous mode are terminated by UNIX signals, thus infinite blocks can be avoided by means of the ALARM signal. When certain asynchronous events (*eg* connection release) occur while the TS user is synchronously waiting for another event (*eg* arrival of data), control is immediately returned to the TS user in order to process this event.

In asynchronous mode, indicated by setting the O_NONBLOCK flag at the transport endpoint, there is a mechanism for notifying the TS user of some event (*eg* the arrival of data) without forcing the TS user to wait for such an event (*eg* by calling the function to receive data). In that case the TS user can periodically poll for a desired event until it arrives.

In both modes, eventing is restricted to a single transport endpoint. To wait for events from different sources simultaneously is currently not possible. The future directions section deals with event management in CAE.

The connection-oriented mode TS distinguishes five phases of communications: initialisation, connection establishment, data transfer, connection release, de-/reinitialisation. The phase transitions are controlled by a state machine. The legal procedure is as follows:

1. An transport endpoint to a TS provider is opened using *t_open()*; this establishes a communication path between TS user and TS provider through the transport endpoint.
2. An address is associated with this transport endpoint using *t_bind()*; this makes the TS user connectable from and to the outside world.
3. Using the appropriate connection functions *t_accept()*, *t_connect()*, *t_listen()* and *t_rcvconnect()*, a transport connection is established; the set of functions to use depends on whether the TS user is initiator or responder.
4. Once the connection is established, normal and expedited (if negotiated) data can be exchanged by the functions *t_snd()* and *t_rcv()*.
5. The transport connection can be released at any time either abortively using *t_snddis()* and *t_rcvdis()* or (optionally) orderly using *tsndrel()* and *t_rcvrel()*.
6. Once the transport connection is released, the transport endpoint may be deinitialised by closing it with *t_close()*, it may be reinitialised by unbinding the address with *t_unbind()* and binding a new one, or a new transport connection may be established.

Connectionless mode in contrast to connection-oriented mode has no connection establishement and release phases and data transfer is done with the *t_sndudata()*, *t_rcvudata()*, and *t_rcvuderr()*, functions.

## 4. Implementation of XTI in SINIX

### 4.1. General Architecture

Implementations of transport functions in UNIX systems differ in many details, but a commonly used design principle is a three level architecture. Its top level is a function library implementing the API (the transport library) which is linked to the TS user's program. These function calls represent the API in the most convenient way and map them appropriately onto system calls. At the medium level these system calls are handled by one or more state automata within the UNIX kernel. There global state information about each TS user process is maintained and the operation of the TS user processes with respect to the state machines controlled. Some well defined internal interface then separates the state automata from the actual transport protocol software at the bottom level. Usually, the lower parts of the protocol stack of layers one through four resides – as firmware or software – on a dedicated hardware.

Implementation of the transport functions in 4.2BSD and System V follow these principle. In 4.2BSD systems, the API is formed by the socket library. The socket layer covers the medium level together with the protocol modules of layers four and three. The lowest level is the network interface module that drives the network interface hardware across the hardware/software interface.

The same situation exists in System V. The API consists of the function calls of the Transport Level Interface (TLI). They are mapped onto system calls handled by kernel modules designed within the STREAMS framework. At the lowest level, a device driver interfaces to the network hardware.

This design principle is also true for the XTI implementation within SINIX. All of the design principles can be summarized as follows:

- The general architecture follows the scheme as outlined above.
- The API routines merely map the API requests onto system calls; they do not maintain any state tables to reduce error probability;
- The state automaton is implemented as a character device driver; this isolates it from the kernel as far as possible, simplifies porting into other UNIX system and takes advantages of the UNIX file protection mechanisms for special files;
- The state automaton in the kernel monitors and controls each communication path between the TS user and the TS provider and maintains state tables for all TS users in a centralized manner.
- The software implementing layers one through four is downloaded to intelligent Communication Controller (CC) hardware. This decouples development and maintenance of protocol software from the SINIX operating system and supports the independence of this software from the operating system, *ie* it can be (and is) used with non-UNIX systems too.

The next sections deal with the major building blocks of the architecture.

### 4.2. The Transport Library

The routines of the transport library provide the API through which the transport service is supplied to the TS user. The transport library is linked to the TS user programs. In the SINIX implementation, its purpose is reduced to mapping the convenient and mnemonic syntax of the XTI in an efficient way onto a UNIX character driver's system call interface. Most of the functionality is defined through the ioctl-interface of that driver. This reduces the probability of errors within the applications. In case of failures of the transport functions, only the operating system kernel needs to be exchanged, the applications remain unaffected by these corrections.

### 4.3. The State Automaton

The routines of the transport library map the API onto a driver's system call interface. These are handled by a state automaton embedded as a character device driver. It uses only the standard driver support a UNIX kernel provides and is in principle independent of special features of the UNIX derivative used (file system structure, new system calls, memory management).

All requests are initially processed by a distribution module handling the driver's open, close and ioctl routines and then forwarded to other modules for further processing. These modules perform basic tasks:

- select the TS provider chosen by the TS user;
- associate the transport endpoint to a Transport Service Access Point of the TS provider;
- establish and release transport connections by issuing appropriate requests to the TS provider or by passing connect or disconnect indications to the TS user;
- exchange data with the TS provider or another (local) TS user.

All these tasks are performed in a module which has no knowledge about the syntax of the API. This module is the core of the transport interface. It communicates with the TS providers and maintains all the state tables. They contain all the necessary information about the TS providers (operational and administrative states, limits, I/O control blocks, etc.) and relate the transport endpoints, the TSAPs of the TS providers and the transport connections.

The TS providers reside on intelligent hardware called Communication Controllers (CCs). There are currently two different types of CC, a CC-LAN for the connection to a local area network and a CC-WAN for the connection to public wide area networks.

The hardware/software interface of the CC is hidden from the state automaton and from the top layers of the TS provider downloaded to the CC by a pair of corresponding software adapters; on the host side, it is called WAN and LAN adapter respectively, on the CC side, it is called Host Port. The adapter software transfers requests and indications transparently between the automaton and the TS provider on a CC. Syntax and semantics of these requests and indications are specified in a way that is common to all TS providers. A new type of CC (say, for the connection to an ISDN) would just require a new adapter on the host side and a TS provider capable of understanding the requests and indications.

Each type of TS provider (ISO-WAN, ISO-LAN, etc.) has its own transport address format. With this, the selection among the CCs is accomplished. A TS user can, through different transport endpoints and TS providers, simultaneously maintain multiple connections across different networks.

There is also a LOOP Adapter that does not correspond to any real CC. It is used for the local communication between two TS users that reside in the same system. This facility is not intended for use as a substitute for the UNIX interprocess communication. Two TS users, however, should be able to communicate by the same API, whether they reside in the same or in different end systems.

A boot module takes care of the initialization of the state automaton at boot time and needs no further explanation. The administration module enables layer-management applications on the user level to access Management Entities of the adapters and the Layer Management Entities on the CCs directly. These entities provide information on the available resources (*eg*, state of the CCs), permit the protocol software to be loaded and controlled (layer-specific traces and statistics), and protocol parameters to be set.

## 4.4. The Protocol Software

The TS provider downloaded to a CC is implemented by a Communication Control Program (CCP). There are several advantages to having the complete protocol software of layers 1 to 4 downloaded to the Communication Controllers. The decoupling from the SINIX kernel gives great flexibility in the development process. New transport protocol stacks can be developed in the course of time and used with a number of SINIX versions and with all machines of the SINIX family. Maintenance and enhancement of the existing software are facilitated as well, and specific customer needs can be satisfied more easily. This flexibility is especially important in the European market where protocols sometimes have to be slightly adapted to conform to the different regulations of the national PTTs. A worldwide convergence process is required that will remove the remaining interoperability problems. Once a mature state is reached one might think about a VLSI implementation of protocol stacks.

The number of available machine slots for the CCs varies throughout the family and so does the maximum number of TS providers supported. The range starts with one slot for the single-user machine and ends with eight slots for the system at the upper end of the family. In this last case, one may have two CC-LAN loaded with equal or different LAN profiles and six CC-WAN, also with equal or different profiles. This gives way to embedding the systems simultaneously in a number of different network environments.

The CCPs for use in WANs (CCP-WAN) implement TS providers based on ISO transport protocol class 0 and 2 (TP0 and TP2) as well as proprietary transport protocols from SIEMENS' TRANSDATA architecture. They can be used with leased lines, circuit-switched networks (V.24/X.21) and packet-switched networks (X.25) and are loaded to the CC-WAN. For LANs (CSMA/CD) TS providers based on ISO transport protocol class 4 (TP4) and TCP or UDP are implemented. Still other CCP variants support the connection to SNA networks and to proprietary multipoint environments.

### 4.5. The Transport Name Service

The Transport Name Service (TNS) constitutes an essential support for a TS user. No professional communication software wants to encode the multitude of transport addresses for the different TS providers. The simplest approach is to introduce symbolic names and to have a look-up table in each end system. This, however, becomes inefficient and hard to maintain in larger networks with many machines and many TS users. Needed is a TNS with one logical database distributed across the machines in the network.

SIEMENS developed a TNS according to the guidelines prescribed in the X.500 recommendations resp. the correponding ISO/CCITT convergence documents about a Directory Service in an OSI environment. Its building blocks are a TS directory containing entries indexed by symbolic names holding the transport addresses. A background process *tnsxd.* maintains the TS directory. A TNS user accesses the TS directory by means of a TNS User Agent that interacts via interprocess communication with that background process. Currently, existing communication applications have no knowledge about the location of the TS directory.

In its first release, the TS directory still resides locally on each machine, it is not yet fully distributed. However, the concepts have already been fixed and the TS user will not be affected by the next step in the development of this service, the transition from a local to a fully distributed TS directory. This step will affect only the background process, but not the TS user.

## 5. Future Directions

### 5.1. Name Management

The current XTI specification makes no assumptions about protocol addresses. Addresses are passed transparently across XTI specified only by a buffer an its length. It is the TS user's responsibility, to develop some mechanism hiding protocol-specific addressing from the user program. This mechanism opens a significant gap in the concept of application protability and connectivity.

X/Open has already recognized that there is more need for standardisation. Work is underway to further connectivity on the basis of a common understanding of name and address management. There is strong emphasis on adapting the Directory Service as layed out in the X.500 recommendation and the corresponding ISO standards.

### 5.2. Event Management

With XTI as defined in [XPG3XTI] it is not possible to wait for several events from different sources, particularly from several transport connections at a time. In order to use XTI in a fully asynchronous manner this feature is desirable for applications handling many connections simutaneously within a single process. To meet this requirement, X/Open is currently planning to establish a event management facility within its CAE and work is underway to define a generic event management model.

### References

[ISO8072] *Information processing systems – Open Systems Interconnection – Transport service definition*, ISO 8072-1986 (E).

[ISO8073] *Information processing systems – Open Systems Interconnection – Connection oriented transport protocol specification*, ISO 8073-1986 (E).

[RFC973] "Transmission Control Protocol" in *Request for Comments 973.* Network Information Center, SRI International, September 1981.

[XPG3CAE] *X/Open Portability Guide Issue 3*, to be published

[XPG3XTI] *X/Open Portability Guide Issue 3, Volume 7: Transport Interface*, to be published

# Implementing Internet Protocols on a small UNIX System

*Danny Backx*[†],
*Rudi Derkinderen*,
*Marc Snyers*,
*Pierre Verbaeten*,

K. U. Leuven,
Dept. of Computer Science,
Leuven,
Belgium
*mcvax!prlb2!sunbim!db*

*ABSTRACT*

The authors have designed and implemented networking software implementing the TCP/IP protocol suite for a small system running XENIX. A library emulating BSD sockets is also provided, as the systems are integrated in a network of workstations and minicomputers.

The paper describes the most important design issues and their consequences. Although we chose for simplicity and a layered structure, measurements show an acceptable performance.

## 1. Introduction

In the framework of an IBM study contract a project was set up to add network support to small UNIX systems. The hardware available consisted of IBM PC-AT's, with interface boards for Ethernet, IBM's PC network and later on token ring. The operating system used was XENIX. Most of the work described here was done by master students in computer science as part of their thesis work.

Several aspects of network software had our attention, such as the integration of small microcomputers into a LAN, how to build gateways, the pros and cons of structuring network software, and porting network programs.

The remainder of this text is structured as follows: first, the design issues for our network software are presented. Then a complete overview of the structure of the implementation is given, followed by the assumptions that we made about undocumented properties of XENIX, and a few workarounds for technical problems. Buffer management and special aspects in each layer are then explained in detail. Finally, some remarks about testing, the system's performance, and the current state of the implementation are given.

## 2. Design issues

Before starting this project, we already had experience with the use of the 4.2 BSD UNIX network facilities, the structure of the UNIX kernel, and related subjects.

The design issues for our network project are grouped in several categories.

### 2.1. A simple design

We believe that the construction of good software is only possible by starting with the right structure. The software should remain simple to build, debug, and understand.

† Currently at B.I.M., Kwikstraat 4, B-3078 Everberg, Belgium

### 2.1.1. Layered software

The OSI reference model [Zim80a] provides a clear structure for network software. Its layering approach divides network software into well-defined functional units, each of which can be easily understood. These layers should remain visible in our implementation.

### 2.1.2. Fixed size buffers

Since this is a small-scale project, we will use only one kind of fixed size buffers. Having only one kind of buffers to deal with simplifies the software.

The size of these buffers is important: if it is smaller than the MTU† of a network, then a large packet received from that network will have to be fragmented into several smaller parts immediately after reading it. Since this is both inefficient and bad practice, we choose to use a buffer size larger than the MTU's of all directly connected networks.

### 2.1.3. Sending packets is always done by a process

Without this choice, part of the code for transmitting a packet would have to be inside the interrupt routine, another part would remain were it is now. So it simplifies the structure of the software, as it prevents the code from being scattered over several places.

Note that the process responsible for the transmission is normally the user process which does the communication, but in other cases it can be a special purpose system process. This will be discussed later.

## 2.2. Compatibility with other systems

### 2.2.1. Implement BSD's socket abstraction

Since most of the other computers attached to our local networks offer the socket primitives, it is rather important to implement these also.

## 2.3. Methods for getting adequate performance

Our approach to get higher performance is based on observations discussed in [Cla85a] .

### 2.3.1. No unnecessary copying of data

It is widely known that copying data in network software should be avoided by all means. It is a sure way to construct low-performance software; we use a buffer pool and pass pointers to buffers between software modules.

### 2.3.2. Minimum number of context switches

Also well known is that one should avoid superfluous context switches in order to send or receive packets. Context switches in UNIX systems are known to be expensive.

Note, however, that this issue conflicts with the decision to have packet transmission done by a process. We prefer the simple approach, with packet transmission done by a process, and avoid other superfluous context switches.

## 2.4. Various other design issues

### 2.4.1. Transparent access to different networks

This has several implications. First, application level software should not notice any differences between several networks. If a computer is attached to several networks, and a process wants to receive data from any network, then no special precautions must be taken.

Second, our network software must define and use internally a sufficiently powerful subset of the capabilities of each network.

† Maximum Transfer Unit, is the maximum packet size that can be transmitted on a network.

### 2.4.2. No static variables in library routines

Experiences with network software that relied on state information in static variables of library routines, has revealed major drawbacks. For instance, the *exec()* system call destroys this state information. Therefore, socket descriptors were not equivalent to other file descriptors.

In order to avoid the situation outlined above, no state information should be present in libraries.

### 2.4.3. Easy implementation of a rudimentary gateway

This was one of our original goals.

### 2.4.4. Protection aspects

Since unrestricted access to a network allows one to spy on others, or even disrupt communication, some precautions should be taken.

One problem is that, if port management is not properly protected, one user can pretend to have port X on host Y, while this port is used by somebody else. By doing this, the behaviour of programs of the other user can be disturbed.

Another problem is unrestricted access to the raw network. This can easily be misused for getting other people's passwords by listening to communication with a terminal server.

Protecting port management and raw network access implies that all basic network software (*ie* everything up to the transport layer) must reside inside the UNIX kernel. Then access to the network can be controlled by setting permissions on the network device entries in the directory */dev.*

### 2.4.5. Global buffer pool

In case several networks are attached to one computer, it makes no sense to have a separate set of buffers for each network. When one network (A) is used heavily, while another network (B) is idle, network A should be able to use all buffers available. It makes no sense in that situation to split the buffer pool in two distinct parts: this would reduce performance dramatically. Therefore, the buffer pool of the networks must be shared.

### 2.4.6. Fairness

This should be understood in several ways: using one network heavily must not prohibit the use of another attached network; but also one process doing a lot of network communication should not be able to prohibit other processes to use the same network.

## 2.5. Implementation restrictions

Just as important as the design decisions are the restrictions imposed by our environment. Our goal was to add network support to a XENIX system.

### 2.5.1. No source code

The fact that we have no source code for the operating system, has several implications on the design.

First, the only way to add functionality to the kernel is using device drivers. Adding system calls to the system is impossible. All the system calls we would like to add have to be emulated by library routines calling the system calls that can be defined in a device driver: *open()*, *close()*, *read()*, *write()*, and *ioctl()*. Library routines have the disadvantage of being slower, and more vulnerable, but are easier to debug.

Another implication is the use of buffer memory: we work with the XENIX operating system on IBM AT computers. The XENIX kernel is compiled in *"medium model"* mode, which means that one can have an unlimited amount of kernel code, but the amount of kernel data is limited to 64 Kbytes. Since the default XENIX kernel already consumes most of this space, only a small number of buffers for network support can be used. We developed a workaround for this problem, which involves programming the MMU†.

† The Memory Management Unit is that part of the hardware which handles mapping of virtual to physical addresses. One entry was added to the map, in order to be able to address the network buffers.

A third implication is related to the fact that a process must always be responsible for sending a packet. This means that we need a process for implementing a protocol such as ARP [Plu82a] . This process will always run in system mode. We give more details in paragraph 3.2 .

### 2.5.2. Small computer

Adding a lot of buffers to the UNIX kernel reduces the memory available for user processes. Therefore, one should not exaggerate in the allocation of buffers.

## 3. Global design

### 3.1. Overview

Our implementation is structured into four layers. The upper layer implements the socket abstraction. Since it is implemented as a set of library routines, it can easily be replaced by another set of primitives.

The other three layers are part of the UNIX kernel. They constitute a hierarchically structured set of (pseudo-) device drivers. The layer 1 drivers directly communicate with the network adapters (except for the LB driver, which controls the pseudo network called the *loopback*). Global to all kernel-level drivers is one set of buffers. Associated with this pool of buffers is a set of routines called the Memory Manager.

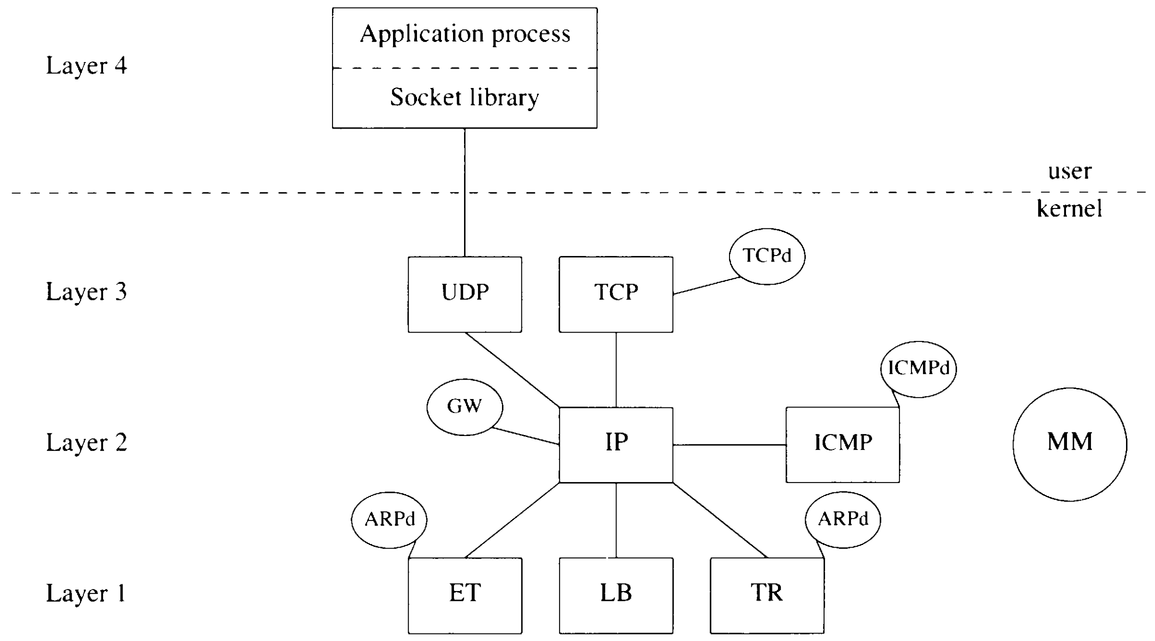


**Figure 1**: *Structure of the network software*

Each of the rectangles in figure 1 is a UNIX device driver (ET stands for Ethernet, TR for token ring). The ellipses are special processes, while the circle represents the global buffer pool.

Besides the standard set of system call routines, each driver defines additional routines for cooperation with adjacent layers. For instance, if the UDP driver wants to send a datagram, it calls *IPsend()* , which in its turn will call one of the layer 1 send routines, say *ETsend()*. This routine will try to deliver the datagram to its destination.

### 3.2. Assumptions and workarounds

In the above discussion, we referred to routines like *IPsend()* and *ETsend()*. These routines do not have the standardised names for device driver routines, nor do they obey any general parameter convention. We assume that we can add new routines to the kernel, with the parameters that we want them to have. We also assume that one device driver can call another device driver's routines freely. As a consequence, we can pass pointers to buffers as parameters to routines. These properties of XENIX are undocumented.

As mentioned above, the ellipses in figure 1 are processes. The processes drawn in the kernel are supposed to run in kernel mode only. Another process which acts this way is the traditional *swapper* process of UNIX. Since we did not have sources of the operating system, we were not able to make the kernel processes start automatically at boot time. Instead, we added another command to the *ioctl()* routines of our device drivers. Only a process executing with superuser authority is allowed to execute this command. If, for instance, a process executes the appropriate command on the Ethernet device, it will thereby indicate that it will be the ARP process. In the *ETioctl()* routine inside the XENIX kernel, the process will enter an infinite loop, in which it does whatever it is supposed to do. The process is actually started from within the boot shell script `/etc/rc`.

## 3.3. Sending and receiving data

The actual data flow will be clarified by an example. We will first consider how a UDP packet is sent, then how one is received.

The user sends a packet by calling the socket-library routine *sendto()*, which will do two system calls: an *ioctl()* to pass the destination address to the kernel, followed by a *write()* to actually send the data. Both calls are executed by the UDP device driver. The ioctl-call merely copies the destination into a temporary variable, and returns. The write-call first allocates a buffer, by calling MMalloc(), then copies the user's data into that buffer, and finally calls *IPsend()* with the destination and the buffer as arguments. After the return from *IPsend()*, the status will be returned as the result of the *write()* system call.

Inside *IPsend()*, the routing table is searched for the first hop towards the destination. Using the network table, the send routine of the appropriate network device driver is selected and called.

This send routine, say *ETsend()*, will try to deliver the data to the destination. Since the destination's physical address may not be known to the sender, this may require sending an ARP request first.

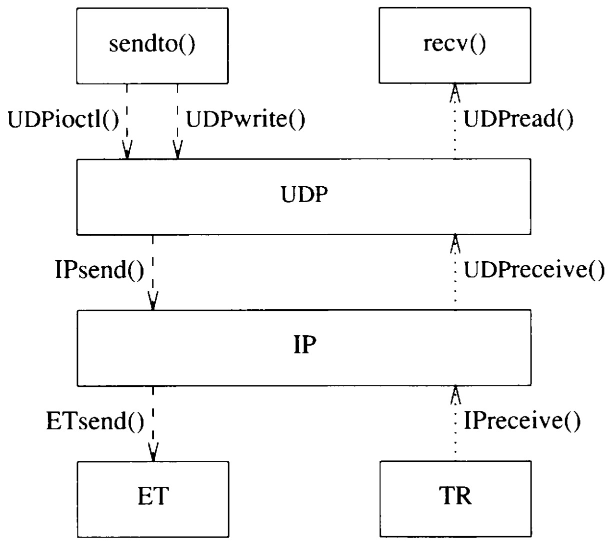


**Figure 2**: *Sending and receiving packets*

A packet being received will of course travel in the opposite direction. When the network adapter has received a packet, it interrupts the UNIX kernel. The interrupt routine *TRintr()* will allocate a kernel buffer by calling *MMalloc()*, and it reads the packet into it.

The interrupt routine will now pass the packet to the IP layer, by calling the routine *IPreceive()*. This is a so-called *upcall [Cla85a]* . This routine will first verify the checksum of the IP header. Then it examines the other fields in the header: packets with local destination are passed to the appropriate protocol module, e.g. by calling *UDPreceive()*. Other packets are put in the gateway queue, and the gateway process is awakened. If anything at all is wrong with the packet, it is thrown away by simply deallocating the buffer with a call to *MMfree()*.

Inside the routine *UDPreceive()*, the UDP header is checked. If the destination port exists, then the packet is appended to the receive queue for that port, and the process possibly waiting for it is awakened. If the port does not exist, the packet is thrown away.

Whenever either the packet is thrown away or put into a queue, the calls to *UDPreceive()* and *IPreceive()* return to the interrupt routine.

As shown, the only copy operations are the ones we can not avoid: one between the user address space (the user's buffer) and the kernel address space (the global buffer pool); the other between the kernel address space and the network adapter. The copying method may either be DMA, or a tight loop executed by the main CPU.

## 3.4. Buffer management

An important issue in buffer management is the system's behaviour when it runs out of buffers. This is especially true on a small system, since this situation will occur much more often than on a large machine.

### 3.4.1. Design decisions

In buffer management design, we followed two general rules:

- It should always be possible to accept an incoming packet. Such a packet can not blindly be ignored, because it might contain information which frees other buffers in queues. Discarding the new packet would also be unfair: if process A has a long queue of incoming but unread packets, and process B is waiting for a packet to arrive, then process B could be blocked forever if process A does not empty its queue.
- It does not make sense to allocate too many buffers for packets leaving on the same network, since a network adapter can only transmit packets with a certain maximum speed.

In general, buffers can either be assigned to a process for sending a packet, or to an interrupt routine for receiving one, or they can be linked into a receive queue.†

Buffer preemption is only possible for queues. The first buffer in a queue must not be preempted, because a process could be in the middle of a read operation. Stealing the buffer then would be disastrous. At any time, the number of buffers that can not be preempted is equal to‡:

| | |
|---|---|
| NSB | *for* send buffers |
| Nports | *for* buffers in front of queues |
| Nintr | *for* buffers for incoming packets in interrupt routines |

It is obvious that the sum of these numbers should be lower than the number of available buffers. Otherwise some critical part of our system will certainly fail. Note however that assuring this involves limiting the number of ports that is simultaneously open (Nports).

Note that we have revealed a drawback of software layering which is not at all obvious: if we could decide in the lower layer whether or not it would be wise to read a new packet from the network††, then we would have to use buffer preemption far less often. Unfortunately, these decisions can only be made in the higher layers.

### 3.4.2. Buffer preemption

There are two important questions concerning buffer preemption: when does the system have to preempt a buffer, and which buffer does it choose.

† Actually, a buffer could also be linked in the gateway queue. The gateway queue is treated like any other queue, so we do not mention it throughout the text.

‡ Assuming Nports is the number of opened UDP ports, Nintr is the number of packets reserved for the interrupt routines. For TCP, Nconns * ConnSize must be added, where Nconns is the number of TCP connections at a time, and ConnSize is the number of buffers per TCP connection.

†† E.g. does it contain a TCP acknowledgement which is higher than the previous one.

The answer to the first question is quite easy: a buffer can be preempted by any call to *MMalloc()*, which is called when a packet arrives (in the interrupt routine); and when a process wants to send a packet. This means that any send operation can cause a packet for any process to be thrown away.

For some send operations, *MMalloc()* is not called, so no buffers are preempted. These are: replies to ARP requests, packets being forwarded by the gateway process, and ICMP replies. The receive buffer is also used for sending.

Which buffer will be preempted ? As we mentioned before, only buffers in queues can be preempted, except for the first one. Buffer preemption must also be fair: the preemption algorithm must search the queues in circular order, restarting with the next queue after every preemption.

Another question is how many buffers should be preempted when the free list is empty. At least one buffer should be freed, but choices are possible: e.g. preempt the last buffer in every queue containing more than one buffer.

### 3.4.3. Limiting the number of send buffers

To implement send buffer limitation, we augment each layer 1 and 2 device driver with a *sendbfr()* and a *release()* routine. Before allocating a send buffer, a transport level driver should call *IPsendbfr()* with the destination as parameter. This routine will find out to which network the request should go, and then it calls the corresponding routine, say *ETsendbfr()*. That routine will check whether it can grant this process the permission to allocate a buffer, and return if it can do so. Otherwise it blocks, waiting for another process to complete sending. After a transport level driver gets permission to allocate a buffer, it will do so, and send the packet in the way we described earlier. Finally, it will call *IPrelease()* which will call *ETrelease()* to notify the network driver that the buffer was released.

For a process trying to transmit a packet, the limitation has the following effect: if many other processes are sending a packet **at the same time,** then this process will be delayed until some of the other processes have finished transmitting their packets.

## 4. A closer look at the layers

Now that the most important characteristics of our network software have been explained, we can take a closer look at the remainder of the system. This chapter describes the important modules that have not been discussed before. We refer to the appendices for description of the routines defined in each layer.

## 4.1. Network device drivers

### 4.1.1. ARP

ARP should conceptually be located between the lowest layer and IP, but is built into the network device drivers for simplicity.

The Address Resolution Protocol [Plu82a] provides a mapping between IP-level addresses *(internet addresses)*, and the physical addresses of network adapters. ARP was originally designed for Ethernet networks, but it is currently also used on IEEE 802.5 Token Ring networks.

If the physical address of a destination host is unknown, a so-called ARP request is broadcast, asking *"Who knows which physical address matches this internet address ?"* Usually only the host with that address replies with a ARP-reply.

We created a special ARP process to answer these requests. Incoming packets are checked in the interrupt routine, and, if necessary, forwarded to the ARP process. Since replies to ARP requests are important, we give them higher priority than normal packets.

### 4.1.2. The loopback network

The loopback network is a pseudo-network. Communication between processes on the same computer passes via the loopback network instead of a physical network.

The implementation of the loopback device driver is quite simple: a packet that is transmitted via *LBsend()* is copied into a new buffer. Then the new buffer is passed back to the upper layers via a call to *IPreceive()*. An important detail is that *IPreceive()* expects to be called within an interrupt routine only. Therefore, interrupts must be disabled before calling it, and they must be restored after returning from it.

Note that communication via the loopback network does involve a copy operation which could have been avoided. To do so, each of the transport level protocols must be modified to recognise the address(es) of the local host.

## 4.2. Internet Protocol layer

This layer is responsible for fragmentation, reassembly, and routing. The *Internet Protocol (IP) , aided by the Internet Control Message Protocol (ICMP) , performs these functions. ICMP is the protocol by which error and debugging messages are sent through the network.*

### 4.2.1. The ICMP module

Our implementation of ICMP is far from complete. Although the internet standards state the opposite, most other protocols or applications work fine without ICMP.

The ICMP module must be able to send replies to ICMP packets it receives. Examples of such packets are the ICMP ECHO REQUEST, and ICMP TIMESTAMP REQUEST. In both cases, the packet must slightly be modified, and then sent back. It is obvious that a process is needed for replying to such packets.

Another group of packets recognised are error messages to our TCP transmissions. If a TCP connection is requested with a host that can not be reached, we may get a HOST UNREACHABLE message from a gateway. TCP uses this information to respond faster to problem situations.

### 4.2.2. The gateway process

Incoming packets with a remote destination are passed to the gateway by the IP module, by calling *GWreceive()*. This routine will first check whether the gateway process is alive. If not, then the packet can not be forwarded, and is thrown away.

In the normal case, the destination of the packet is looked up. If a route towards the destination exists, the packet is appended to the gateway queue; otherwise it is thrown away. Then the gateway process is awakened. The gateway process itself is quite straightforward: whenever it is awakened, it tries to send the packets in the queue to their destination.

## 4.3. Transport protocol layer

The transport protocols currently supported are UDP and (recently) TCP.

### 4.3.1. The User Datagram Protocol

The UDP protocol [Pos80a] is very simple. When a process tries to send a packet, this is done immediately if possible. No acknowledgements are awaited. When a packet is received via a network adapter, it is put into the queue for its destination port. As mentioned before, the packet may be destroyed if the computer has a lack of buffers. The process is not notified in any way when this happens.

Since our buffering system has only one type of buffers, and fragmentation is not implemented, it is not possible to send packets larger than the size of a buffer.

### 4.3.2. The Transmission Control Protocol

The TCP protocol [Pos81c] is much more complicated than UDP. It consists basically of a finite state machine, which we will not discuss here.

Important aspects of TCP are:

- it must be able to send replies to other computers (*ie* to send acknowledgements for the data received);
- it must also be able to retransmit data for which no acknowledgement arrived;
- timeouts must be implemented in order to recover from losses on the network, and start retransmissions.

For all of these aspects, a special process is required.

Sending and receiving data is not as simple as with UDP. When a user wants to send data, this data is first put into a queue for outgoing data; then it is also sent. If no acknowledgement arrives in time, then the contents of the queue will be retransmitted by the TCP process. When data arrives, and it falls within the "window", it is put into a queue. The data will not be removed by the buffer preemption strategy. The TCP implementation will remove all buffers as soon as possible.

One unusual optimisation is built into TCP, due to the fact that the system's buffers are so large: if a buffer containing a small number of bytes is inserted into a queue, and the buffer ahead of it is not full, then they are merged into one. Admittedly, this is a copy operation, but it is only executed if the number of bytes to copy is very low.

### 4.4. The socket library

The socket layer is not implemented inside the XENIX kernel for a technical reason only: because we had no sources of the operating system, it was impossible to add new system calls. The standard system calls of device drivers can not be used, as the socket primitives such as *sendto()* and *recvfrom()* have more parameters than we can pass with the *read()* and *write()* system calls.

Therefore, these primitives had to be implemented by mere library routines, which do several system calls, to pass all the parameters. For instance, *sendto()* is implemented as an *ioctl()* to pass the destination to the kernel, followed by a *write()* to transfer the buffer and start the actual transmission.

This situation has implications on the robustness of the system. Consider a program which opens a socket, and then forks. After the *fork()* , there are two processes having access to the socket. If both processes attempt to send at the same time, then both system calls for process A should precede both system calls for process B, or the other way around. All the other possibilities have another effect.

It is possible to build an algorithm into the kernel which prevents the wrong order of system calls. The algorithm uses the process id to distinguish different processes. If process B attempts to do the *ioctl()* between the system calls of process A, then process B is temporarily blocked. The algorithm is very complex due to the necessity to handle the possible sudden death of process A, and because of the fact that an unknown number of processes could be executing it simultaneously.

## 5. Current state of the work

Although the system, as it is described here, has been implemented, many other features are not (yet) included. We will list the most important omissions briefly:

- Subnets are not implemented. Only internet addresses of classes A, B, and C are recognised.
- ICMP is far from complete. Most notably, it does not send redirects, nor does it listen to them.
- No reassembly or fragmentation is done.
- Ethernet trailers [Lef84a] are not recognised, so communication with VAXen running 4.2 BSD can only happen if the VAX doesn't send trailers. Communicating with VAXen running 4.3 BSD works fine.
- It is possible to add a network interface while the system is up, but it is not yet possible to remove one.
- Changing the routing tables while a process is sending a packet may upset the counters indicating the number of send buffers per network.

We have device drivers for the following networks:

Ethernet (3COM501 and 3COM505 adapters)

IEEE 802.5 Token Ring (IBM Token Ring adapter)

Loopback pseudo-network.

## 6. Testing and performance measurements

Our implementation was programmed by various people, and also tested with various means.

### 6.1. Porting programs

An important aim of our networking project was to provide a wide range of networking programs which support communication with other computers.

The first program ported to our system was *tftp*, which is a program that provides a simple user interface to the Trivial File Transfer Protocol, and which only requires the UDP protocol. As this port was possible without modification of the source, it also shows our conformance to the BSD socket primitives.

On top of UDP, a part of Sun's RPC package was implemented. It was tested using programs such as *rpcinfo, showmount,* and *rusers*. Finally, a client implementation of NFS in library routines was recently finished.

Useful programs based on the TCP protocol have not yet been completed, since our implementation of TCP was finished only recently.

## 6.2. Performance

A number of conclusions from performance measurements will be discussed. In our measurements, several systems were used: a SUN-3/280 with an Intel Ethernet adapter, a MicroVAX II with an Ethernet adapter, an IBM RT (6150) with an Ethernet and a Token Ring adapter, and an IBM AT running our software, with an Ethernet and a Token Ring adapter.

It is almost impossible to tell why one system is faster than another. Differences may be due to the speed of the CPU, the network adapter used, the physical network itself, and of course the network software. In order to be able to compare the relative speed of several systems, we first list the time needed to transmit a large amount of data (32000 packets of user data length 512) regardless of whether or not they are received. The information in table 1 should only be used in order to compare the relative speeds of the systems mentioned.

| Host | time | KB/s | ratio |
|---|---|---|---|
| Sun | 68.5 | 233 | 4.1 |
| IBM RT (TR) | 85 | 188 | 3.3 |
| VAX | 102.4 | 156 | 2.7 |
| IBM RT (ET) | 107.1 | 149 | 2.6 |
| AT (ET) | 271 | 57 | 1 |
| AT (TR) | 285.5 | 57 | 1 |

**Table 1**: *Sending a large amount of data*

Table 2 shows the number of packets that an AT can receive when another host sends 32000 packets each 512 bytes long at full speed. The columns labeled *transport* indicate the number of packets that were received by our system in case the destination port did not exist (the packets were dropped immediately). The column labeled *process* contains the number of packets that a process can actually receive (the process contains a loop which continually reads packets).

| Source host | transport | | process | |
|---|---|---|---|---|
| Sun | 16,086 | 50% | 11,604 | 36% |
| IBM RT (TR) | 29,137 | 91% | 5,877 | 18% |
| VAX | 31,954 | 99.8% | 21,730 | 68% |
| IBM RT (ET) | 31,984 | 99.9% | 21,961 | 68% |
| AT (ET) | 31,981 | 99.9% | 31,460 | 98.3% |
| AT (TR) | 32,000 | 100% | 31,999 | 100% |

**Table 2**: *Worst case packet throughput*

The lines are ordered according to the sender's speed. In both tests, we can see that a fast sender can cause severe packet loss due to saturation at the receiver. In the first test, packets were lost because the software could not keep up with the hardware; whereas in the second test more packets were thrown away by the buffer preemption code.

Some of these figures are quite low. Please keep in mind that we are dealing with worst case situations here. It is obvious that more packets are lost with faster senders. The curve plotting packet loss versus sender speed has the form of a knee, as with most saturation problems.

In an attempt to measure not only worst case situations, a modified version of the TFTP program was used: disk accesses were eliminated. What is left is a simple stop-and-wait protocol. All measurements for this test were done between two identical machines. The figures in table 3 indicate the number of seconds needed to copy a packet of 512 bytes back and forth 32000 times. The ratios do not differ much from the ones in table 1.

| Host | Time | Ratio |
|---|---|---|
| Sun | 272 | 3.1 |
| VAX | 417 | 2.0 |
| IBM RT (ET) | 384 | 2.2 |
| AT (ET) | 845 | 1 |

**Table 3**: *Stop and wait protocol*

Another series of measurements indicate the time needed to process packets in several layers†:

2.62 ms
: to receive a packet, pass it up to UDP, and then dispose of it.

2.94 ms
: to run the buffer preemption code, receive a packet, pass it up to UDP, and put it in a queue.

9.98 ms
: to receive a packet, pass it up to UDP, put it in a queue, and let the process read it.

10.26 ms
: to receive a packet, put it in the gateway queue, and forward it.

We also did experiments with the buffer preemption algorithm. Different algorithms revealed no noticeable performance differences. The figures above confirm that this would be very hard to measure: the time needed for preempting a buffer is one thirtied of the time needed to receive a packet.

Finally, we also list the size our new XENIX kernel occupies, as output by the standard UNIX program *size*.

| | |
|---|---|
| xenix.orig | 171 Kbytes |
| xenix.udp | 188 Kbytes |
| xenix.tcp | 220 Kbytes |

**Table 4**: *XENIX kernel sizes*

## 7. Conclusion

The original aims of our project have been accomplished. We have integrated network software into a small UNIX system. The implementation has been done using the layered structure defined by the OSI reference model. Nevertheless, several comparative tests have shown that an acceptable level of performance is reached.

Finally, we have built a framework which, because of its simple layered structure, is easy to maintain and to extend. It cost us one day to integrate an existing device driver into our system.

## Acknowledgement

We thank IBM for donating us the hardware and operating system which we used for this project.

## References

Cla85a. D. D. Clark, ''The Structuring of Systems Using Upcalls,'' *Operating systems review*, vol. 19-5, pp. 171-180, Dec. 1985.

Lef84a. Samuel J. Leffler and Michael J. Karels, *Trailer Encapsulations,* RFC 893, Network Information Center, Stanford Research International, April 1984.

† For these measurements, a process doing a very long calculation was run first, and the time the process took to complete was measured. Then the same program was run again, but a large number of packets (each 512 bytes long) were sent from another host, so the computer running the calculation needs more time to complete the calculation. The other figures were obtained by making slight variations in the setup.

Plu82a. D. C. Plummer, *Ethernet address resolution protocol*, RFC 826, Network Information Center, Stanford Research International, November 1982.

Pos80a. Jon Postel, *User Datagram Protocol*, RFC 768, Network Information Center, Stanford Research International, August 1980.

Pos81a. Jon Postel, *Internet Protocol*, RFC 791, Network Information Center, Stanford Research International, September 1981.

Pos81b. Jon Postel, *Internet Control Message Protocol*, RFC 792, Network Information Center, Stanford Research International, September 1981.

Pos81c. Jon Postel, *Transmission Control Protocol*, RFC 793, Network Information Center, Stanford Research International, September 1981.

Zim80a. H. Zimmermann, ``OSI Reference Model - The ISO Model of Architecture for Open System Interconnection,'' *IEEE Transactions on Communications*, April 1980.

## Appendix 1: Network device drivers

The network device drivers do the actual interfacing with the hardware. In order to hide differences between the physical network architectures, a very simple model has been chosen. In fact, the only functional routine defined is *XXsend()*. Its purpose is quite straightforward. Note, however, that this routine must be able to perform broadcast transmissions, if possible on the network. Analogously, for receiving, the interrupt routines *XXintr()* need only call *IPreceive()* which will deliver the packet to its destination if possible.

As mentioned before, our implementation requires that all packet transmissions are initiated by a process (not by an interrupt routine). A consequence of this requirement is that the packet transmission code is a recognizable part of the routine *XXsend()*, whereas, in other implementations, it would be scattered throughout that routine and the interrupt routine.

The routines *XXsendbfr()* and *XXrelease()*, which were added for limiting the number of send buffers, are quite straightforward. *XXsendbfr()* decrements a counter, and returns if the counter is still positive, but blocks otherwise. *XXrelease()* simply increments the same counter. There is one such counter per network interface, and it is initialised at system boot time.

Other routines defined by the network device drivers are the interrupt routine, and the standard system call routines. The interrupt routine *XXintr()* is absolutely necessary since the communication protocol between the XENIX system and the network adapter relies heavily on interrupts. In order to use any system call, *XXopen() and XXclose()* must also be defined. *XXread()* and *XXwrite()* can be used to send and receive raw network-level packets. The packets received by reading a raw Ethernet device, for instance, contain not only user data, but also TCP, IP, and Ethernet headers. It is obvious that access to the raw network devices must be suitably protected.

Finally, the *XXioctl()* has an important role in our network software: it serves to do all the system calls we could not implement otherwise. The lowest level *ioctl()* is used for

- setting and modifying system parameters (e.g. this interface's internet address);
- obtain statistics about traffic;
- using a process as the ARP daemon;
- examine and modify the ARP table.

## Appendix 2 : The IP module

The Internet Protocol module is really a switch. Based on the contents of the IP header in a packet just received, the module decides whether the packet is destined towards this host, or to another host. In the latter case, it is handed to the gateway module. In the former case, other fields in the header are checked, and the packet is given to the correct protocol module (ICMP, UDP or TCP).

In the other direction (for sending a packet), the IP must first try to find a route towards the destination. The packet's destination is used to search the *routing table*, and find the internet address of the first hop towards the destination. This address is then used to search the *network table*, which resembles the classic *bdevsw* table in UNIX. The network table thus provides the addresses of the routines which should be called for sending the packet.

| network | routines | | |
|---|---|---|---|
| 192.31.23.0 | ETsendbfr | ETsend | ETrelease |
| 192.31.27.0 | TRsendbfr | TRsend | TRrelease |
| 127.0.0.0 | LBsendbfr | LBsend | LBrelease |

**Table 5**: *The network table*

Fragmentation and reassembly are not implemented yet.

The routines defined by the IP module are *IPsend(), IPsendbfr(),* and *IPrelease()* for sending packets, and *IPreceive()* for receiving them. Essentially, *IPreceive()* hands the packet to one of ICMP, UDP, TCP, and the gateway. The other routines search the routing and network tables, and then call a routine in one of the lower level device drivers.

Another special routine is defined for the IP module: *IPsetroutines().* This routine is used by the network device drivers for initialisation. A network adapter's internet address is set by a command in /etc/rc, which will do that by performing an *ioctl()* system call on the network device driver. The kernel routine executing that system call will save the network address specified, and it will also call *IPsetroutines()* with the appropriate parameters to initialise its entry in the network table.

The normal system calls must be defined for the IP driver. The *read()* and *write()* system calls are not necessary, but *IPioctl()* is necessary for setting up the routing table.

Since the IP device driver has no associated hardware, the *IPintr()* routine is not defined.

## Appendix 3 : The transport protocol layer

The transport protocols only define one special routine: *XXreceive(pkt).* It will be called by IP to pass incoming packets to the transport layer. Since there is no hardware associated with the transport protocols, the interrupt routine need not be defined.

Since the transport layer is the highest layer inside the kernel, it is obvious that the system calls must all be defined now. What the system calls *open(), close(), read(),* and *write()* do is quite clear. The function of *ioctl()* depends highly on the protocol.

# Striping Network Device Driver for TCP/IP

*John A. Pew*

GE Aerospace
NASA Ames Research Center
*pew@orville.nas.nasa.gov*

*ABSTRACT*

This document describes a project to design a UNIX network device driver that uses multiple paths across a network. The goal of this project is to increase the performance of the network between a Cray-II supercomputer and an Amdahl 5880 mainframe. Both systems run a version of UNIX with TCP/IP. Remaining interoperable with other hosts on the network is a chief concern; therefore, modifying the TCP/IP software is not a viable alternative. The network medium being used between these systems is the Network Systems Corporation (NSC) HYPERchannel.

The limiting factor in the speed of this network is the speed of the IBM-compatible channels on the Amdahl. The HYPERchannel is rated at 48 Mbits per second, however we are unable to take full advantage of this speed because the channel connections on the Amdahl are rated at only 1.5 Mbytes (12 Mbits) per second. This 1.5 Mbyte limit makes it impossible to use the full bandwidth of the HYPERchannel. To circumvent this limit, a striping driver that will simultaneously direct traffic over multiple paths to multiple HYPERchannel addresses is proposed. In order to accomplish this striping, one IP address must map to multiple HYPERchannel addresses. However, in order to adhere to IP standards, a one-to-one mapping of IP address to network interface is maintained.

## 1. Introduction

The dissimilar I/O rates at which various computers transfer data from host to network interface create a problem in throughput. Even the fastest network is underutilized if the host computer is unable to transfer data to and from the network hardware at rates equal to the I/O rate of the network. Such is the problem at NASA Ames Research Center where Network Systems Corporation (NSC) HYPERchannel is used to communicate between supercomputers, mainframes, minicomputers and workstations. The I/O rate of the Cray-II supercomputer allows network data to be transferred at a high rate between the channel and the HYPERchannel adapter. At the other end, on the Amdahl 5880, the HYPERchannel adapter is able to receive the data at network speeds but the transfer to host – via IBM-type channels – severely limits the overall throughput.

## 2. Background

A Mass Storage Subsystem (MSS) is under development at NASA to provide high speed archival of data from a Cray-II computer to an Amdahl mainframe. The Amdahl runs UTS, a version of UNIX provided by Amdahl. The Cray-II runs UNICOS, Cray's version of UNIX. Both systems run TCP/IP networking software. The MSS project at NASA involves modifying UTS to accommodate the huge amounts of data produced by the Cray-II. One of the components of the MSS, the high performance file system, will provide users with high speed disk access, but the high speed file system is of little value if the data cannot be transferred to the MSS at similar speeds.

The speed of the Cray I/O processor allows data to be transferred at rates over 100 Mbits/sec. The bottleneck on the Amdahl is related to the speed of the IBM-compatible channel. The current model HYPERchannel adapter attaches to the channel in interlock mode and is therefore limited to an I/O rate of 1.5 Mbytes/sec, which means that the highest data transfer rate between the Cray and the Amdahl is limited by the channel speed of the Amdahl.

## 3. Goals

The following objectives were established to increase the overall throughput between these two systems.

- Use the available bandwidth of the HYPERchannel hardware
- Maintain current TCP/IP networking software
- Achieve a transfer rate of 85 Megabits per second

A chief concern is to remain interoperable with other hosts, so changing the TCP/IP software or using other protocols is not an alternative.

## 4. Proposed Solutions

To achieve a rate of 85 Megabits per second would be impossible with the current A-series hardware because of the limiting factor of the channel speed. Even if the channel speed was limitless, the network speed would still impose limits far below the 85 Mbit/sec goal. Figure 1 shows the current configuration with existing hardware.

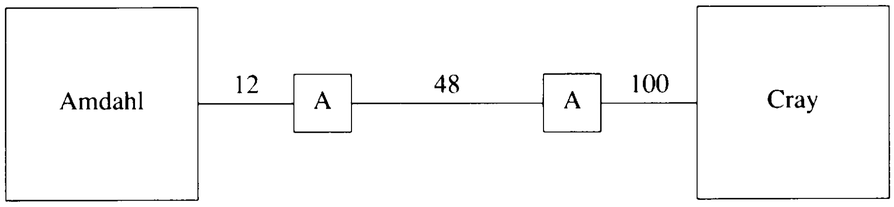


**Figure 1**: *Current Configuration*

Network Systems has been promising newer, faster hardware for some time. The new N-series hardware promises a doubled network speed and a possible 4.5 Mbyte/sec channel connection. If these two specifications are realised, then a theoretical rate of 36 Mbits/sec could be achieved (see Figure 2). However, this rate still falls significantly short of the 85 Mbit/sec goal.

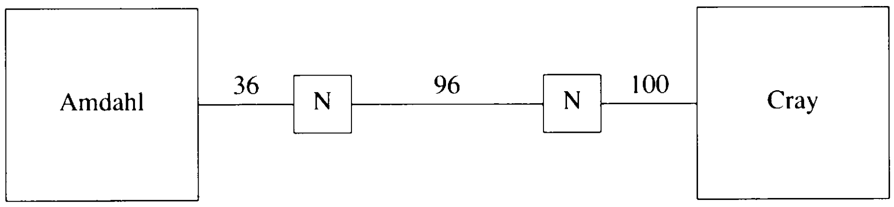


**Figure 2**: *Current Configuration with New Hardware*

The goal of 85 Mbits is still 50 Mbits/sec away and the bottleneck is still the channel speed.

The solution to the channel speed problem is to stripe data from the adapter to the host across multiple channels, but current A-series hardware provided by NBSC does not allow multiple channel connections from a single adapter. However, new N-series hardware is designed to connect to multiple channels, which could provide the solution to the channel speed bottleneck. If connections could be made from the adapter to three seperate channels at 4.5 Mbytes each, then an aggregate bandwidth of 36*3 = 108 Mbits/sec between the adapter and the host could theoretically be attained. At this rate the bottleneck would become the rate of the network as shown in Figure 3.

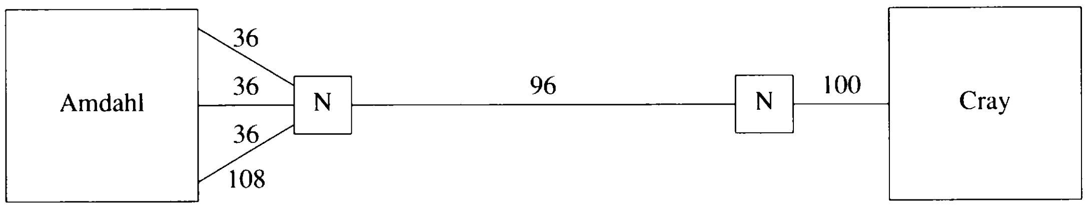


**Figure 3**: *Proposed Configuration with New Hardware*

When traffic originates from the Amdahl, some mechanism needs to direct traffic across the different paths between the host and the adapter. This task is rather trivial: the HYPERchannel device driver is designed to start I/O on different channel addresses. Some configuration must be done to notify the driver of the channels that are available to send on, but sending the message poses no problem to the upper layers of the networking code because only the driver need be aware that packets are being sent out on multiple addresses. The following code from the device driver implements the striping on traffic originating from the Amdahl.

```
hystart(unit)
int unit;
{
     register struct hy_softc *hyc;

     hyc = &hy_softc[unit];
     if(hyc->hy_state == ST_CHAN_BUSY) {
          hyc = &hyc_softc[unit+1];
          if(hyc->hy_state == ST_CHAN_BUSY) {
               return;
          }
     }
}
```

The HYPERchannel header control field is set with the `I_LIED` bit, which notifies the receiver that the source addresses in the HYPERchannel header may actually not be the address at which the packet was written. The receiving host generally throws away the HYPERchannel packet header after examining it.

A more difficult question is: How is networking traffic addressed to different channels? Directing packets to different addresses implies that the remote host understand that there are multiple hardware destination associated with a single internet address. The source and destination address in the HYPERchannel packet header is specified by an adapter number and subchannel address. The subchannel address is analogous to a hardware port number on a given adapter. The standard paradigm in the TCP/IP protocols is that an IP address maps to one and only one hardware address. Such is the case with Ethernet where the arp table maintains the mapping of Internet address to Ethernet hardware address.

A possible solution to the mapping of a single Internet address to multiple hardware addresses would be to require the firmware on the receiving side (Amdahl) to distribute packets destined for a single network address to multiple channels. The firmware distribution of packets would eliminate the need to change any software, either TCP/IP or driver level, on the remote system (Cray). Outgoing packets from the Cray would simply be addressed in the normal way. When a packet was received by the adapter on the Amdahl at the specified address, the firmware would then notify the host that a packet had arrived (by asynchronous interrupt) and that it was available to be read at a given address. The specified address would be determined by the firmware. The algorithm could either use a round robin approach or dynamically determine which channel was available. The receiving device driver on the Amdahl would be required to understand that packets arriving on any of certain specified addresses were associated with a given network interface.

The firmware approach is preferred for the application at NASA Ames. The changes required to implement striping via software would be completely contained in the driver on the Amdahl. No software would have to be modified on the Cray.

Unfortunately, the current version of the N-series hardware from NSC does not meet the specification to attain 85 Mbits/sec: the firmware does not support striping, and only two channel connections are available. Due to these limitations a software striping solution is also proposed.

## 5. Software Striping

Software striping would allow packets to be directed to multiple adapters connected to the same host, but it would require the sending host to address each packet appropriately so the sending host would require software that understood where to send the packets.

The existing HYPERchannel driver resolves an Internet address to a HYPERchannel hardware address via a table lookup. A table stored in the kernel associates a HYPERchannel hardware address with an Internet address. This static table is installed when the system is brought up, after the interface is configured. It is not dynamically updated and aged like the arp table used for Ethernet. The HYPERchannel driver makes a call to the `hyroute` function each time the output routine (`hyoutput`) is called from the IP layer.

Hyroute fills in the HYPERchannel header with the appropriate information including destination HYPERchannel address and trunks to try.

The modifications necessary are limited to the structure of the hyroute function in the driver. The hyroute table is an array of hy_hash structures as shown below.

```
#define HYRSIZE 256

struct hy_hash hyr_hash[HYRSIZE];

struct hy_hash {
     u_long hyr_key;       /* destination              */
     u_short hyr_flags;
     u_short hyr_dst;      /* adapter number and port */
     u_short hyr_ctl;      /* trunks to try            */
     u_short hyr_access;   /* access code              */
};
```

To accommadate striping a few additions are made to the HYPERchannel driver. A new flag, HYR_STRIPE, is added; the size of the table is increased by HYSTRIPESIZE entries; and a new field is added to the hy_hash struct, hyr_next. There are now two regions of the table that will be referred to as the "standard region" (the first HYRSIZE entries) and the "striping region" (the next HYSTRIPESIZE entries). The following shows the structure of the table with the changes.

```
#define HYRSIZE          256
#define HYSTRIPSIZE      12
#define HYR_STRIPE      0x20      /* striping entry           */

struct hy_hash hyr_hash[HYRSIZE+HYRSTRIPESIZE];

struct hy_hash {
     u_long hyr_key;       /* destination              */
     u_short hyr_flags;
     u_short hyr_dst;      /* adapter number and port */
     u_short hyr_ctl;      /* trunks to try            */
     u_short hyr_access;   /* access code              */
     u_short hyr_next;
};
```

When the hyroute table is installed in the kernel, all the non-striping entries are in the standard region; however they now have an additional field which is unused (hyr_next). The entries in the standard region are guaranteed to reside in the first HYRSIZE entries in the table. The additional HYSTRIPESIZE entries at the end of the table are reserved for the new striping entries. The algorithm used to hash into the table guarantees that a standard entry will never encroach into the striping region of the table.

An address that is to be striped still has a primary entry in the standard region of the table; however, it contains no pertinent information except in the hyr_next field. The hyr_next field contains an index into the striping region of the table where the address information can be referenced. The striping entries associated with one primary entry are circularly linked by the hyr_next field.

The hyroute function first identifies the primary entry in the standard region. If hyr_flags is set to HYR_STRIPE then the hyr_next field is referenced. Each time referencing occurs, the hyr_next field in the primary entry is updated to point to the next address in the stripe.

```
if(rhash->hyr_flags == HYR_STRIPE) {
     nrhash = &rt->hyr_hash[rhash->hyr_next];
     rhash->hyr_next = nrhash->hyr_next;
     rhash = nrhash;
}
```

Each packet destined for an Internet address that is marked as striped will be sent to one of the addresses in the striped region.

## 6. Current Status

The new N-series hardware has been delivered and installed, but all the functionality has not been integrated into the N-series hardware. The channel connections are still at 1.5 Mbytes/sec and the firmware to distribute packets to different channels is not available. Some software striping has been tested for functionality but not for performance.

If the N-series hardware does not achieve the anticipated results, then an alternative is to keep the A-series hardware (which NASA already owns) and implement the software striping across multiple adapters as shown in figure 4.

The software striping via A-series hardware would not meet the desired objective of 85 Mbits/sec, and if the N-series hardware does not meet specifications, then the 85 Mbits/sec goal will have to be reevaluated.

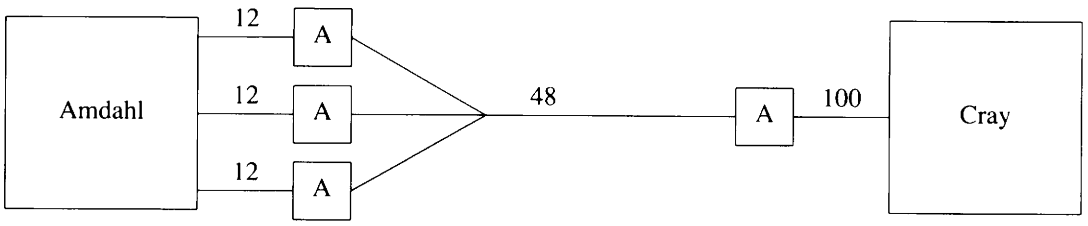


**Figure 4**: *Possible A-series Configuration*

Testing the striping driver for performance has yet to be done. This test environment has not yet been established.

Error recovery may also require additional work. If striping is implemented in software, what happens if one of the channels becomes inoperable ? Packets sent to the broken channel will be lost, but, when retransmitted by the upper layers of the networking software, may be successfully received over one of the good channels. This may not completely bring down the network but cause it to limp rather severely.

## 7. Conclusion

Striping via firmware in the HYPERchannel adapter is the preferred solution: it would require less software modifications and would be faster. If striping firmware is not available, then software striping would provide the solution though hardware capabilities may require adjustment to the original goals.

# More Haste, Less Speed

*David S. H. Rosenthal*

Sun Microsystems
2550 Garcia Ave,
Mountain View, CA 94043
*dshr@sun.uucp*

*ABSTRACT*

In the good old days, six years ago, I was teaching at the Universiteit van Amsterdam and sharing a 1MIPS, 4Meg machine with up to 60 other users. At times, it was irritatingly slow. Now, I have a 1.5MIPS, 4Meg machine all to myself, and at times it's irritatingly slow. Where did all the extra power go? Are we doomed always to be frustrated at the performance of our Unix systems? What can we do to improve the performance of the systems we use and the code we write?

"What we created
has separated
a past we hated
from what we are feeling today
'cause everything's different now."

'Til Tuesday – *Everything's Different Now*

## 1. Introduction

Six years ago I was teaching Computer Science in the Universiteit van Amsterdam. The department had lots of students, and not much money, so we configured a low-cost, high capacity UNIX timesharing system. It was a second-hand PDP11/70 with 4 Megabytes of memory, two Massbuss disk controllers with 5 SMD disks, and 64 lines of DH11 DMA asynchronous line multiplexers, running 2BSD.

The system proved very popular, and I frequently found myself sharing it with around 60 other users. I won't claim that the performance was great, in fact at times it was irritatingly slow, but I could get my work done at the cost of some frustration.

Now, I have a Sun 3/75 with 4 Megabytes of memory and a SCSI disk on my desk*. It has the same amount of memory, and a processor that is around 50% faster than the PDP11/70. I have it all to myself instead of sharing it with others. I won't claim that the performance is great, in fact it's irritatingly slow at times, but I can get my work done at the cost of some frustration.

A long time ago, C. Northcote Parkinson observed a similar phenomenon and enunciated Parkinson's Law to describe it:

*Work expands to fill the time available for its completion.*

It seems as if an analogue of this law operates for UNIX

*The hardware is never powerful enough to keep the user happy.*

Other have made the same observation, [Chr88a] but they normally blame it on the growth of applications. I contend that there are non-linearities in the behaviour of UNIX systems and their users that interact together to make the statement true, reinforcing the effect of application growth. It isn't enough to adopt a reactionary attitude, and complain that networking, window systems, virtual memory and all the other things that have been added to UNIX since the golden age of V7 burn more CPU cycles than they're worth. If that were true, at some point the machines would be fast enough to afford all these goodies. I contend that there are fundamental mechanisms that mean that simply adding MIPS will *never* keep the user happy.

* The fact that this is hardly a state-of-the-art machine doesn't, alas, affect my conclusions. I have observed the same phenomena on much more powerful machines, they're just easier to see on my machine.

This is not a "scientific" paper. I have only fragmentary evidence for my hypothesis, and no clear idea of how to test it. I am simply presenting some of the ideas I'm spending some of my time thinking about.

## 2. An Experiment

Figure 1 shows the results of a simple performance experiment. The SunView user interface toolkit, and the underlying SunWindows window system, were modified to allow a script of user actions to be recorded and played back as fast as the system would accept input. The object was to simulate an interactive user limited by overall system performance. The time to complete the script was recorded in a series of runs; on each run the UNIX kernel's memory self-sizing routine was faked to return a different amount of system memory.

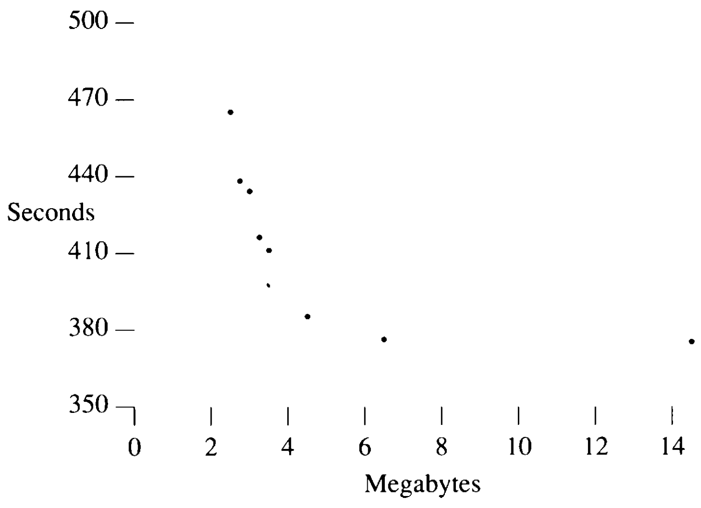


**Figure 1**: *Task duration* vs *available memory*

The exact numbers aren't important, its the shape of the graph that is telling us something. What it is telling us is hardly unexpected. As you add memory to a system with very little memory, the time to complete the script falls very rapidly. But after a certain point is reached, adding more memory has little effect. In this case, the knee in the curve is around 4.5M of available memory.

## 3. What is Going On?

Like fluid flow and other dynamical systems, operating systems with multiple processes and virtual memory exhibit transitions between ordered and chaotic states [Gle87a] . As the system is driven harder and harder, response time gets gradually worse and paging gradually increases. At some point, response gets sharply worse and paging increases dramatically as the system starts *thrashing*. Thrashing is inevitable; introducing virtual memory provides the system with a feedback loop that necessarily allows chaotic behaviour and introducing multiple processes allows the user an easy way to modulate system load.

Another way of looking at this is that the system oscillates between a state in which it is saturating the CPU, and a state in which it is saturating the disk. The system is stable in the CPU-saturated state, and unstable in the disk-saturated state, in the sense that small changes in the load in the ordered state produce small changes in response time, whereas in the chaotic state small changes in load can produce large changes in response.

## 4. What Does It Mean?

Since it appears that the shape of this graph is a consequence of multiple processes and virtual memory, two of the reasons why we want to use UNIX rather than MS-DOS or the Mac "OS", spending some time thinking about what the shape implies looks like a good idea. So far, I have come up with the following implications:

- Is the goal *good* performance or *predictable* performance? They aren't the same thing.
- The goal is obviously to keep the system operating in the ordered region. Adding memory helps but there may be other things just as good.

- Virtual memory is *not* just like real memory, only slower. We need VM-aware design for good performance.
- It is impossible to discuss the performance of a UNIX workstation without discussing how the user is driving it. To keep the system in the ordered region, we need to affect what the user does as well as the way the system responds.
- Simplistic benchmarks can do more harm than good.

## 4.1. Good Performance or Predictable Performance?

Comparisons between the performance of workstations and "personal computers" such as the IBM PC and the Mac are often unflattering to the workstations despite their much greater horsepower. The difference is largely due to the fact that PC operating systems have only a single process, and do nothing unless the user asks them to. The workstations, on the other hand, have multi-process systems and perform many activities in parallel autonomously.

As these processes compete against each other for resources such as CPU and memory, they reduce the *predictability* of the response of the workstation. On a PC, a given operation will always take exactly the same time. On my workstation, the time a given operation will take depends on whether, at the time I invoke it, someone else is sending me mail, or one of the file servers I use is heavily loaded, or there is a

```
        make vmunix
```

underway in another window.

If the machine is thrashing, its response will be much less predictable. This makes the contrast with the PCs even more marked. Not merely is the system inherently less predictable, but the more you use it the less predictable it gets!

And, worse still, workstations typically use network facilities. Sharing network bandwidth and resources with other workstations adds to the unpredictability, and there's little that can be done to reduce this.

Given a multi-process system, users use it to get more work done. I can still remember the amazing productivity boost I got when I switched from Control Data timesharing systems to 6th Edition UNIX and I discovered the effects of '&'. The effect of this, and even more so of multiple windows, is that users are free to load up the system until the response drops to a point they deem unacceptable. For obvious reasons, the shape of the curve means that users will end up close to the knee of the curve.

Most users will spend most of their time on the flat part of the graph close to the knee. Every so often, they will push the system over into the chaotic state and experience a long pause. The system will exhibit a kind of *hysteresis*; once it gets into the chaotic region it will take some time to recover. The spectrum of responses the user sees will be bi-modal, many short pauses and a few very long pauses. Since the users cannot predict the results of their actions, the long pauses will be unexpected, and thus memorable. The combination of the shape of the curve and the natural way to use a multi-process, virtual memory workstation means that the user will be unhappy.

There's another aspect of the user's behaviour that makes the transitions to the chaotic region more unpleasant. What do you do when the system stops responding for a while? You prod it in some way to see if it's still there, perhaps by moving the cursor, hitting return in some shell window, or popping a menu. All these things make the problem worse, because they cause processes to be scheduled and increase the demand for pages.

## 4.2. Adding Memory is Good, but ...

The natural first reaction to the graph is to add memory immediately. Adding memory not merely improves performance, it improves predictability as well. Although memory is wonderful stuff, it is expensive (in low-cost workstations the memory is often worth more than everything else in the box put together) and tends to come in large chunks (the minimum memory increment in current technology is typically 4 megabytes and it will soon be 8).

Notice that the graph is labelled "Task duration *vs* available memory". What is the memory being used for, and why isn't some of it available? When Berkeley Unix is booted, it normally reports two numbers; on my machine they are:

```
        mem = 4096K (0x400000)
        avail mem = 3325952
```

The second of the numbers doesn't mean what it says. In the good old days on the PDP11 kernel memory was statically allocated, but nowadays the kernel allocates much of its memory dynamically through a version of *malloc()*, and this makes it more difficult to work out how many non-pageable (locked-down) pages the kernel is really using. Table 1 shows the kernel's typical use of locked-down pages on my machine;

| Used for | 8K Pages |
|---|---|
| text + data | 77 |
| heap | 35 |
| mbufs | 13 |
| UFS bufs | 10 |
| valloc space | 10 |
| u structs | 17 |
| locked-down total | 162 |
| + "avail mem" | 406 |
| total | 568 |

**Table 1**: *Non-pageable memory usage*

if you believe these numbers my machine has grown an extra half megabyte of memory! The discrepancy represents the memory the kernel has dynamically allocated and thereby stolen from the pageable page pool. In fact, my machine normally has about 350 pages in the page pool.

## 4.3. Saving Memory is Much Cheaper

A 4M machine with 8K pages not merely has very little memory, but it also has very few pages in the page pool. So:

- Adding even a few pages to the page pool will have a big effect.
- Using the pages in the pool effectively is important.

The best thing you can do to add pages to the page pool isn't to buy more memory, it's to configure your kernel carefully. We find that people often continue to run the (huge) generic kernel we supply to get the boot process started. This is like beating your head against the wall; its wonderful when you stop.

But even following the manufacturer's instructions isn't always enough. The *SDST160* SunOS4.0 kernel Sun advised its customers to run on a 3/75 had only around 290 pages in its page pool on my machine. By carefully eliminating all unneeded options and resizing kernel data structures downward, I was able to free up a lot of pages, and since then the advice has been changed [Mic88a] .

On my machine, the kernel is wiring down around 1.25M. The data points on the left of the graph are 0.25M apart, so they correspond to 20% of the kernel memory. Relatively small changes to the configuration of a typical workstation kernel can make a 20% difference to the amount of memory it uses, and thus a significant performance difference.

Other good things you can do include:

- Using shared libraries. Experiments measuring the age of pages (the length of time they have survived without being paged out) in my system show that the oldest pages almost always come from the shared *libc*. Old pages are being used effectively, normally because they are shared by several processes.
- Reducing your program's demand for space. Excessive *malloc()*-ing is bad for you; it creates pages that are non-shared by definition. On the other hand, although the stack is equally private, its pages are almost always in memory and tend to be re-used more effectively.
- Thinking about locality of reference for text. Re-ordering modules in libraries can pay big dividends. So can moving infrequently used code, such as error reporting, onto other pages, or even into other libraries.
- Thinking about locality of reference for data. Putting a layer over *malloc()* that arranges that data objects that will be referenced together appear adjacent in memory can make a big difference.

### 4.4. Adding Disk Bandwidth is Important Too

Why could my PDP11/70 support 60 times as many users as my Sun 3/75? I admit that these days I'm much more of a load on the system than I was then, since I now expect my machine to paint on a mega-pixel display, track a mouse every 1/30 second, support a window system, network file access, and, of course, *sendmail*. But a factor of 60? The key difference between the systems is I/O. The PDP11's SMD disks were faster than the Sun's SCSI disk, and above all it had several disks and used them effectively. With 5 drives on two controllers, it often had two transfers in progress *and* another two drives seeking, while the CPU was running a user process. Simply comparing the transfer rates of the two drives gives no idea of the PDP11's huge advantage in overall disk bandwidth.

We ran a blind taste-test experiment, in which users were asked to complete a complex task using the window system and a number of applications three times each, comparing the subjective performance. All the systems appeared identical, with the same software, display and keyboard. Although all the machines had 4M of memory, unknown to the subjects the machines had different CPU architectures, MIPS ratings from 3 to 5, different versions of the operating system, disks, and so on.

In general, users were unable to detect any difference between the systems. The only difference they could reliably detect was between disks; some of the systems had a 70M SCSI drive and some a 140M SCSI drive with twice the transfer rate. The systems were all operating in the disk-saturated region of the curve, and adding disk bandwidth was the *only* thing that made any difference. It is very easy for a customer to compare MIPS ratings of workstations, but very difficult to compare disk bandwidth. Given the increase in program size, and thus the increasing demand for disk bandwidth, it is clear that the effective disk bandwidth of typical Unix systems has declined in recent years while their MIPS rating have increased.

### 4.5. VM-aware Design

When they have something to do, processes compete with each other for the pages in the pool. By being constantly aware of this, we can design systems that use the page pool more effectively by scheduling fewer processes less often.

For example, there is long-running debate among the users of UNIX window systems between the advocates of *click-to-type* focus management and the advocates of *focus-follows-cursor* mode. Note that this debate doesn't happen among Mac users; the only window that is really active is the front one, and focus-follows-cursor simply isn't a concept. Users want some indication of the window that has the input focus, so each application that gets the focus wakes up and makes some subtle change to its appearance to indicate that it now has the focus.

Take a focus-follows-cursor system and a screen crowded with windows, scrub the mouse around without typing anything and watch the performance monitor's paging activity graph. It will rise sharply even though the system isn't really doing anything, because of all the processes waking up and saying "I've got the focus". In fact, the situation is even worse than that. Suppose the screen is being driven by an X server [Scha, Sch87a] . Each time the mouse moves from an *xterm* window (client A) to another *xterm* window (client B):

- The X server gets the CPU, generates the window crossing and focus transfer events, sends them to the clients (making both A and B runnable), and goes to sleep.
- Client A (or B) gets the CPU, examines the event, and sends the protocol requests to change its cursor from solid to hollow (or vice versa) to the server (making it runnable), and goes to sleep.
- Client B (or A) gets the CPU, examines the event, and sends the protocol requests to change its cursor from hollow to solid (or vice versa) to the server, and goes to sleep.
- The X server gets the CPU, reads the two sets of requests, adjusts the screen accordingly, and goes to sleep.

For one mouse movement, we have a best case involving four schedulings of three separate processes. The worst case is much worse, since the window manager process may also get involved, and the server may get the CPU and process one client's requests before the next client gets it. It appears that in a VM-aware user interface, click-to-type focus management is the way to go.

Another feature of many UNIX window systems is the need to wake up processes to repair damage to their windows. When the window tree is changed in such a way as to damage some windows, the X server sends Expose events to all the clients which need to repaint some of their image. They all become runnable together, and start competing for the pages in the page pool. The sudden spike in demand for pages may well be enough to push the system into the chaotic region of the graph.

We did an experiment in which we modified the kernel-based SunWindows system to request a repaint only from the process with the front-most damaged window, and to wait until the repaint was complete before requesting a repaint from the next window back in the stack. This was a big improvement:

- Even with surplus memory, the visual impact of having the windows repainted in a predictable order, and having each window repaint completely before another window painted, was much better.
- Even with surplus memory, on average more pixels were updated sooner. This is because, on average, the front window has more pixels exposed than windows deeper in the stack.
- On limited-memory machines, the page traffic generated by each re-arrangement of the screen was much reduced.

Reducing the impact of a given set of damage to windows isn't the only thing we can do. We can also change the user interface to reduce the damage-prone-ness of the window layout. An extreme example of this is the *tiling* window layout used, for example, in Carnegie-Mellon's *Andrew* system [Mor86a] . Windows are arranged in columns; they never overlap and no window layout change can cause more than one window to repaint. In the worst case for an X tiling window manager, only the window server, the window manager, and a single client would be competing for the pages in the pool.

There is an index of damage-prone-ness; the average number of processes that are woken up by a single to-level window manipulation. It is possible for a window manager to estimate this:

```
index = 0
for (each mapped top-level window A) {
     if (A is at least partly visible) {
          for (each mapped top-level window B that A overlays) {
               if (there is at least one pixel in B which would
                    be visible if A were unmapped) {
                         index++
               }
          }
     }
}
```

Window managers might use this estimate as input to their window layout policies, striving to minimise the index. Even in an overlapped window environment, there are things that can be done. For example, the window manager might automatically iconify windows that are deep in the stack and do not have the input focus.

## 4.6. The User is Part of the System

In the good old days of timesharing, the system load was statistical in nature. It resulted from a process of averaging the actions of a large number of users. In a workstation, the system load is largely the result of the actions of a single user, and is highly correlated with the actions of the system.

While the system is operating in the flat part of the graph, load is additive. Adding a small constant amount of load will degrade response by a small constant amount. Users can build a reliable mental model of system performance, and can predict the effect of their actions. Once you get to the knee of the curve, load is no longer additive. Adding a small constant amount of load will degrade response by wildly different amounts depending on where on the curve the system is. Users cannot predict the effects of their actions.

Because the load that the user imposes on a workstation is closely related to the system's behaviour, unpredictability in the system's response will lead to variations in the imposed load as the user's estimates of the load it can absorb prove wrong. These spikes in demand will push the system into the chaotic part of the graph, causing abnormally long delays, reinforcing the unpredictability, and the user's unhappiness.

It's a common observation that experienced workstation users often run a performance monitor, an EEG-like trace of system parameters such as CPU load and disk traffic. Why do they add to the system load in this way? It prevents the need to make the response worse in order to obtain reassurance that response hasn't stopped completely.

The performance monitor also allows the user to find the knee in the curve more accurately, and predict the systems behaviour better. Interpreting its output takes skill, and it may well be that the information available could be presented in a more effective way. A better presentation should have measurable effects on throughput as the user manages the workload to stay closer to the edge of the ordered region; experiments are needed to investigate this effect.

A VM-aware system should also attempt to prevent the user getting into trouble. Systems that fork sub-tasks might monitor the load before attempting the fork, and putting up an alert to warn the user if the load is heavy enough that adding the sub-task would go over the knee. We did an experiment in which the *suntools* program's startup phase was modified to delay starting new applications if the paging load caused by earlier applications was too heavy. The paradoxical result of delaying the start of some of a set of applications was that the set as a whole got started sooner. The effect is similar to that of the repaint-front-to-back strategy.

Once again, this experiment shows that expectations based on linear behaviour are wrong. In fact, we are faced with another paradox. Adding memory (or reducing kernel memory demand) to a system that is around the knee of the curve (and most will be) will probably increase throughput, but the user will probably not be any happier. They will simply add load to the system to bring it back to the knee in the curve, and as a result will continue to see occasional long pauses.

### 4.7. Simplistic Benchmarks Are Harmful

As we saw earlier, MIPS ratings may have no correlation with perceived performance. But they are easy to measure, and they have driven the workstation market for the past few years. In order to achieve better satisfaction from their systems, users (customers) need a benchmark measure that correlates better with perceived performance.

Unfortunately, we have also seen that the user is an essential part of the feedback loop that determines system performance. Building a benchmark that modulates system load as a real user would is very hard. It seems improbable that one could be developed in the near future; much more research into the nature of the relationship between system performance and user actions is needed before understanding sufficient to do this would be available.

In the meantime, using a script-style interactive benchmark to draw the task duration *vs* available memory graph for a range of systems would allow a somewhat better prediction of likely perceived performance than MIPS or the (only marginally more sophisticated) benchmarks in current magazine product reviews [Wil88a] . Some of the facilities needed in X servers to do this are being agreed under the auspices of the group looking at X testing and validation.

## 5. Conclusion

Virtual memory systems running script-style interactive tasks exhibit a task duration *vs* available memory graph with a characteristic shape. This combined with the freedom a multi-process system gives the user to modulate system load, and the tight correlation between system and user actions in a single-user workstation environment, leads to a region of unpredictable and thus irritating system response around the knee in the curve.

Normal user responses to systems that are far from the knee in the curve move them closer to the knee, by adding load or memory as the case may be. Thus, most users spend most of their time close to the knee, experience unpredictable response, and feel dissatisfied. There are two possibilities to explore in an attempt to improve the user's satisfaction with the system:

- Improve the system's behaviour close to the knee. Among the tactics that can be employed are eliminating spikes in the demand for memory, changing the scheduling and paging policies, and exporting to the applications more relevant measures of system load (such as the number of processes in short-term I/O wait).
- Improve the user's behaviour close to the knee. Among the tactics that can be employed are performance monitors displaying more detailed and relevant system load parameters, applications that estimate the load that user commands will impose and warn the user before executing commands that are likely to push the system over the knee.

It is difficult to develop benchmarks that show this problem; because it is directly related to the way a user uses the system. The task duration *vs* available memory graph can be used as a rough indication of likely system behaviour, but much more work is needed to develop benchmarks that model the user's behaviour properly.

## Acknowledgements

Special thanks are due to Mark Liu, Marty Itzkowitz and the other members of Sun's Software Performance Engineering Group, who did some of the experiments. Thanks are also due to Sue Knapp, Joe Provino and Sun's East Coast system software team for doing most of the other experiments. Above all, thanks to Gary Sager for giving me a diskless 4 megabyte Sun 3/75 for my new office and thereby re-awakening my interest in performance.

## References

Chr88a. Steven M. Christensen, "Monstrous Performance," *Unix Review*, vol. 6, no. 12, pp. 61-67, December 1988.

Gle87a. James Gleick, *Chaos: making a new science*, Viking Penguin, New York, 1987.

Mic88a. Sun Microsystems, "SunOS4.0 Read This First, Appendix E," 800-1737-17, Rev. A., November 1988.

Mor86a. James H. Morris and others, "Andrew: A Distributed Personal Computing Environment," *Comm. Assoc. Comput. Mach.*, vol. 29, no. 3, pp. 184-201, March 1986.

Rod88a. Robert Rodriguez and others, *A Dynamic UNIX Operating System*, pp. 305-319, USENIX Conference, San Francisco, CA, June 1988.

Sch87a. Robert W. Scheifler and Jim Gettys, "The X Window System," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1987.

Scha. Robert W. Scheifler, James Gettys, and Ron Newman, *X Window System: C Library and Protocol Reference*, Digital Press, 1988.

Wil88a. David Wilson, "Tested Mettle," *Unix Review*, vol. 6, no. 12, pp. 87-95, December 1988.

# Authentication in a Unix Network Using Smart Cards.

*Dr S. T. Jones*†,
*simon@ese.essex.ac.uk*

*Dr M. J. Clark,*
*markc@ese.essex.ac.uk*

Department of Electronic Systems Engineering
University of Essex
Wivenhoe Park
Colchester
Essex
England
CO4 3SQ

*ABSTRACT*

The security of the UNIX login mechanism has been found to be satisfactory for individual systems. Implementations of remote logins for a network of UNIX systems have been less satisfactory. A system that improves the security for remote login has been developed based on the use of intelligent or smart cards. This paper presents an extension to the mechanism developed that allows the card to be used for authentication by application programs.

First the operation of the login authentication system is reviewed, then an analysis of the possible extensions and problems are discussed. Finally a detailed description of an experimental implementation of a card based authentication system is presented.

The system developed stores the credentials for a user within the smart card, these are accessed by an extension to the UNIX login mechanism, this is used for both local and remote access. Then a communications path is established between the card and the user's login shell. This shell verifies each command with the card and then executes the command which inherits the communications path to the card for its own use.

## 1. Introduction

The login mechanism used within the BSD4.3 version of the UNIX operating system is satisfactory for locally connected terminals, however, it has been shown to be weak for network logins from other systems or workstations. This paper presents a mechanism whereby intelligent cards are used to improve the security of user access control within UNIX An analysis of the problems associated with network logins is presented, possible solutions are discussed, a description of an experimental implementation of a card based UNIX login system is presented, this system is then extended to allow a user's shell and subsequent user other processes to be able to access the card.

## 2. Operation Of The UNIX Login System

The login security of UNIX can be viewed from two points, that of locally connected terminals and that of remote logins using **rlogin**†† and **telnet**‡. A typical network environment consisting of a local host with local and remote terminals and workstations is shown in figure 1.

† British Telecom Research Laboratories, Martlesham Heath, Ipswich, Suffolk, England, IP5 7RE.

†† **Rlogin** includes **rsh**.

‡ **Telnet** uses similar procedures to local logins.

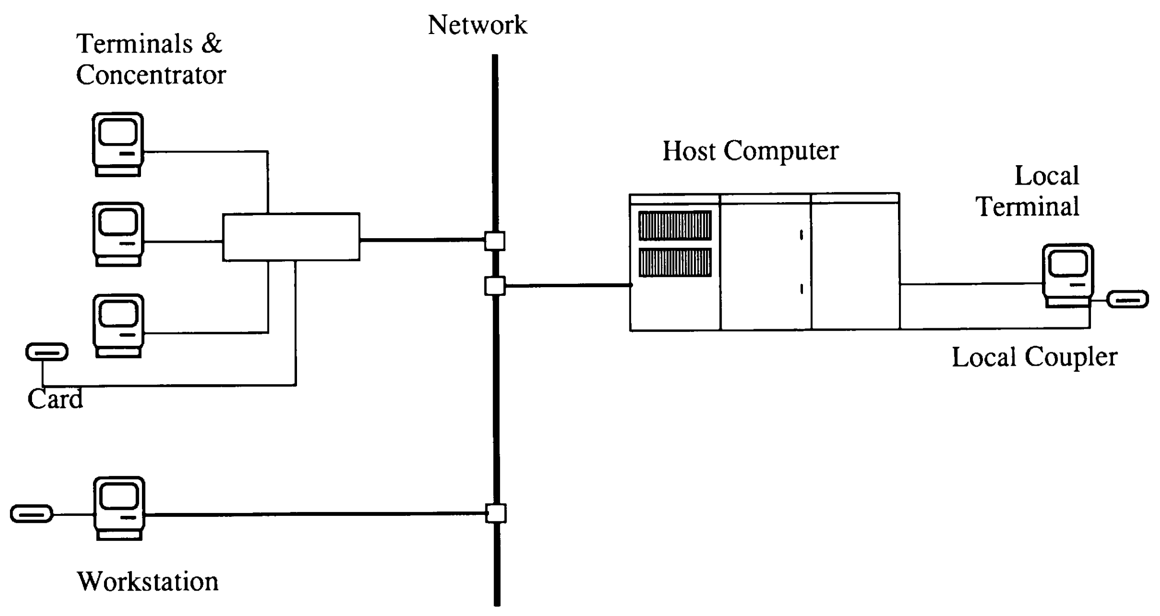


**Figure 1**: *A Typical Network Environment*

Remote logins can originate from three types of user terminal.

1. A conventional terminal connected to another network host.
2. A terminal connected to a terminal concentrator.
3. Workstations with their own network interface.

Cases 1 & 3 have a processor that can perform tasks, where as in case 2 the terminal concentrator's processor is normally dedicated to the connection of terminals to the network. In the following discussion the term *remote terminals* is used to include terminals in case 1 & 3. Solutions are possible for case 2 but are not considered in detail.

## 2.1. Locally Connected Login Requests

Login security for locally connected terminals is based on a password that is entered by the user, used as the key to encrypt a fixed data pattern and the encrypted result compared with an expected result. This procedure is fully described by Morris and Thompson. [Mor79a] The operation of this system is satisfactory provided simple precautions are taken in the choice of passwords.

## 2.2. Remotely Connected Login Requests

Login security for remotely connected terminals is based on the idea of equivalent hosts, these are hosts that have the same management, and share common user names.

When a remote login request is received a check is made to determine:

1. If the source host is considered to be equivalent and the two user names are the same.
2. If the receiving user allows the originating user access from the source host.

If either condition is satisfied then password checking is suspended. If a check fails the incoming user is prompted for a password which is checked in the manner used for local login.

There are problems with each of these two scenarios; with and without a password. When a password is requested, character echo is suspended at the destination machine but the characters are passed across the network unencrypted. It is a simple task, with an network monitor, to trace packets sent to the **rlogin** daemon and collect passwords of remote users.

For a login without a password the system assumes that identical user names on the equivalent machines refer to the same user. This is acceptable in domains using common password files, as with SUN's yellow pages, [Sun86a] but can cause problems where the files are administrated and edited manually and where users do not have accounts on all of the equivalent machines.

The equivalence of hosts has to be defined for most networked services, eg. printing, to successfully operate. But to facilitate one remote operation, access has to be given to all remote operations.

## 3. Motivation And Objectives

The motivation for the investigation of the use of intelligent cards within UNIX login arose from an interest in the use of intelligent card technology for the storage and transfer of access control capabilities. The first step in accessing an application that may require the provision of capabilities is usually to login to that system. Therefore if the card is to be used by the application it could also be used for the login authentication.

The major decision to be made is how to incorporate the facilities that using a intelligent card provides into the existing login procedure ensuring that the technique could be simply extended to all applications.

### 3.1. Secondary Objectives

During the consideration of the aims and motivations the following secondary objectives were formulated.

1. To make as little distinction as possible between local and remote login. This is to simplify the code required and to make for a uniform communication between the login processes and the intelligent card.
2. To establish a communication path between the application process and the card which would be common for both local and remote login. This connection saves the application from determining if the user is local or remote when the connection to the intelligent card is established.
3. To make the techniques developed portable between versions of UNIX requires that no changes should be made to the system programs as the source code for these may not be available.

## 4. Intelligent Card Hardware Description

A generalised view of an intelligent card is a processor and non-volatile storage packaged in a format that has the same physical dimensions as a standard *credit* card, see Svigals [Svi85a] for details. Communication to the card is normally performed serially, either by direct physical contacts or by a contactless system, eg. an inductive loop.

In use the card is connected, or located near, a coupling device that provides the serial communication and the power to operate the processor on the device. Communication to the coupler is generally performed over a standard RS232 interface. In the case of some contactless systems only half-duplex communications can be performed. The memory on the card is either intrinsicly non-volatile or is maintained by an on-card power source.

### 4.1. Communications Restrictions Imposed by the Card

The card system used in the investigation is manufacturered by GEC Card Technology. [GEC86a] The card is based on a 8048 processor with 8k bytes of battery supported RAM. Communication to the card is contactless using an inductive loop system that also provides the power for the operation of the microprocessor. Access to the coupler is via a standard RS232 connection. The inductive loop communication technique avoids the problems associated with physical contacts to the card but restricts the channel to half-duplex operation.

With the GEC card the amount of RAM storage directly available to the card's processor is limited to 32 Bytes. Therefore messages are restricted to this total length. The card's processor can access the main memory but this requires a 'op_system' call per byte which results in a lower performance.

## 5. Using an Intelligent Card for Login Authentication

The approach adopted for login authentication aims to provide a uniform method for both local and remote logins. The conventional UNIX login procedure involves three processes that run before the user's shell is executed. These are, in order, **init, getty** and **login**. The **init** processes is the principle system start up process and creates a **getty** process for each serial line, or display device, on which logins can be performed. It also waits for a user's shell to terminate at logout and then restarts the procedure by creating a new **getty** process. The **getty** process controls the the line parameters and accepts the user's name which it passes to the **login** process that it executes. The **login** processes controls the user authentication and establishs the user's **shell** and environment after successful authentication.

To modify any of these processes to include smart card authentication would require access to the UNIX sources on all types of machine that might be used. This is obviously undesirable and in some cases impossible. The approach adopted is to introduce an additional process, called **auth**, between the **login** process and the user's shell. This is shown in figure 2.

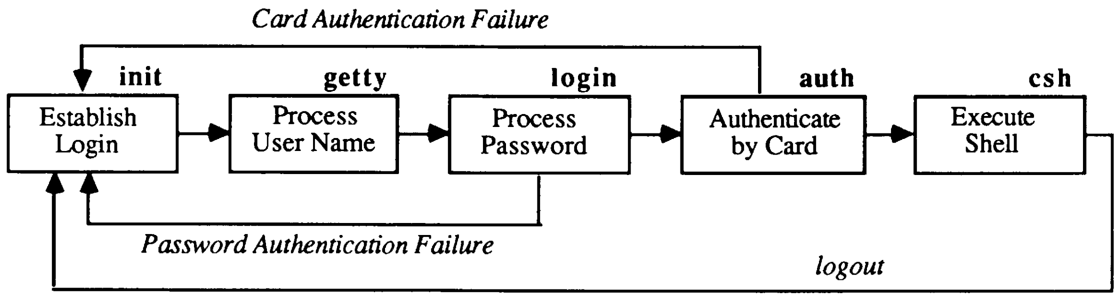


**Figure 2**: *Position of* **auth** *process*

This is achieved by changing the shell entry in the system's password file. The **auth** process performs the additional user authentication before establishing the user's **shell** and environment.

This approach removes some of the flexibility available for selecting the user's **shell**†, however, this can be overcome by creating different named links to the **auth** program specifying the required shell. This can be extended for any process in the system command directory.

A feature of using an additional processes is the flexibility to chose between the two methods of user authentication, traditional UNIX or intelligent **card**. In practice none, either or both can be specified.

## 5.1. Card Authentication Procedure

The following sections describe the procedures required to establish communication with the card and perform authentication checks. The checks used n the implementation are primative but it would be a straight forward task to extend them to include more rigorous forms of user authentication.

The initial stage is for the **auth** program to establish communication to the card. The methods used vary for local or remote logins and are described in the following sections.

If a communications path exists to the reader but no communication occurs to the card the user is prompted to place the card on the reader. If communication can not be established then the authentication is deemed to have failed and the login request is rejected.

Once communications has been established the authentication procedures can be performed. In the implementation communication establishment initialises the card. The user's name is sent, if this matches a name known by the card it is positively acknowledged else it rejected and authentication deemed to fail. If the name is known the user is prompted for a Personal Identification Number, PIN, which is sent to the card and checked against the one held for the user name previously received. If they match the authentication succeeds, if not, two retries are attempted to allow for user error.

If authentication fails the **auth** program exits. This is detected by the **init** process which restarts the login sequence. If the authentication is successful the **auth** program executes the user's **shell**.

† The ability of a user to change their shell with **chfn** must be removed.

## 5.2. Card Communications

The following sections describe how communication is established to a card. Initially this varied between local and remote cards as the approach for the local card was developed first. Subsequently the two approachs were combined to provide a uniform method for both local and remote cards.

### 5.2.1. Communications to a Local Card

When the authentication process is executing on the host to which to card is directly connected communication can be performed by direct access to the coupler via its serial line. There are two different configurations that have to be considered:

1. Host with one card coupler.
2. Host with more than one coupler.

In case 1 the **auth** process determines the device name of the serial connection. In the second case it has to determine which of the couplers is associated with the terminal that requires authentication. Both configurations are solved by the use of a data file that associates terminals and couplers. If there is no coupler available then card based authentication is denied.

### 5.2.2. Communication to a Remote Card

For a remote login connection it is not possible for the local authentication process to directly access the remote coupler. The communication path used is the over same network that was used by the in-comming login request. This requires an intermediate process to be created on each host with a coupler. The remote process, or *card server*, listens for requests from the network, establishes communication to the card, passes the message to the card and returns any reply from the card back to the authentication process.

The communication used over the network needs to mirror the message based request/reply transaction used when communicating with the card. The development environment consisted of a VAX 11/750, VAX11/730 and SUN 3/60 connected by an Ethernet. The protocol used has to provide for the request and its expected response. The choice available was either the Internet UDP/TCP protocols [Lei85a] or SUN's Remote Procedure Call, RPC, package. [Sun86b]

The User Datagram Protocol, UDP, [Pos80a] approach is the simplest for network communication entailing low overheads, but does not provide the facilities to handle the expected response. This would have to be provided within the server and authentication process. The TCP protocol [Pos80b] provides a reliable bi-directional communication path but carries a high overhead during connection establishment and when maintaining the communication path even if no messages are being sent.

The RPC mechanism supports transaction bassed communication and provides timeout and error handling mechanisms. With the RPC approach the 'remote procedure' that is actually executed is the operation performed by the card. The server acts as an agent, passing the request message to the card and returning the reply message to the authentication process.

For the above reasons the SUN RPC package was chosen. The choice has the advantage that the package is available for many different machines, SUNs, VAXen, PCs etc. and is portable to most systems providing a BSD like TCP/IP networking package e.g. Apollos. Any machine dependent data representation differences are hidden within the XDR mechanisms, [Sun86c] allowing software portability between the different machine architectures.

### 5.2.3. Generalised Communication

To provide a common approach the two communications procedures developed were compared to identify common elements. Few common elements exist as a server is essential for remote operation. However the server can also respond to RPC requests comming from processes on the same machine. Therefore the approach for local logins was changed to match that for remote logins by restricting all communication to the card to be via its server.

A side effect of this decision is that the security of communication to the card can be improved. By having a card server on each machine it is possible to protect the connections to the couplers so that they are only accessed by the server. Under UNIX this can be achieved by having the server process and the coupler device share a unique *user id*. Having one server per system reduces the number of processes that would be required on a system with many couplers. The data file containing the association between terminal and coupler is protected by read only access to the servers uid.

## 5.3. Card Message Structure

All communication to the card is based on message exchange.

Classes of message exchanges;

| Type | Action | Response |
|---|---|---|
| Status | Status determination or change | ACK or NACK |
| Information | Store Information | ACK or NACK |
| | Retrieve Information | NACK or Information |
| Processing | Process Information | NACK or Result |

Message transfer is based on a simple character orientated protocol with the following structure;

SOH msg_code msg_len STX message chk_sum EOT

Where SOH, STX and EOT are the standard ASCII control codes.

The **msg_code** identifies the type of message sent and is used to index a table of message handlers. The **msg_len** specified the number of data bytes in the message field. The **message** contains the body of the information sent between the card and the system. The **chk_sum** is identified as being the character immediately preceding the EOT character.

## 6. Using Smart Card For Application's Authentication

The technique developed above provides a facility for access to a smart card for authentication. This can be extended from the authentication of the login procedure to authentication by applications. The important facility developed is the mechanism of communication between the login process and the card. For use by an application this has to be passed onto the application. This procedure is described below with its inclusion into a sample application to determine the operational performance.

### 6.1. Inheritance of Communications Paths

The communication path is established between the **auth** program and the card for the purpose of initial user verification. For the card to be used by subsequent processes such as applications, command processors etc. this path must either be maintained and inheritied or re-established by the child process. If the communications path is to be re-established, it requires that the application *knows* all about the connection to the card. In the case of a network connection this may not be a simple task. It requires that the communications path can be established by a user process, where as the **auth** program may be a system process.

To pass the communications path between processes requires that information can be maintained between the **auth** program and its grandchildren applications. Few secure methods exist as all existing process data is destroyed when a new application process is executed. The method used† is to pass an open file descriptor, with known file_id, between the processes. This file descriptor is then used to pass messages to and from the card.

To maintain transparency of communication for an application process it is necessary for any distinction between local and remote connection to be hidden and, to some extent, to hide the communication protocol used between the card and it's server process. These two aims can be achieved by using an intermediate process between the card server and the application. In practice the intermediate process is the **auth** program.

After successfully completing the login verification the **auth** process will set up a socket pair communication, with a known file identifier, to its child. The auth process will fork and execute the command processor. The command processor can then use this socket pair via the file descriptor to communicate with the card to perform any authentication as required. The file descriptor is maintained and passed on to all of the command processor's child applications†. Figure 3 shows the communication paths between the application and the card.

† Assuming a shell as the original child process.

† With the BSD4.3 **csh** this requires a minor modification to the code to prevent the socket descriptor from being closed.

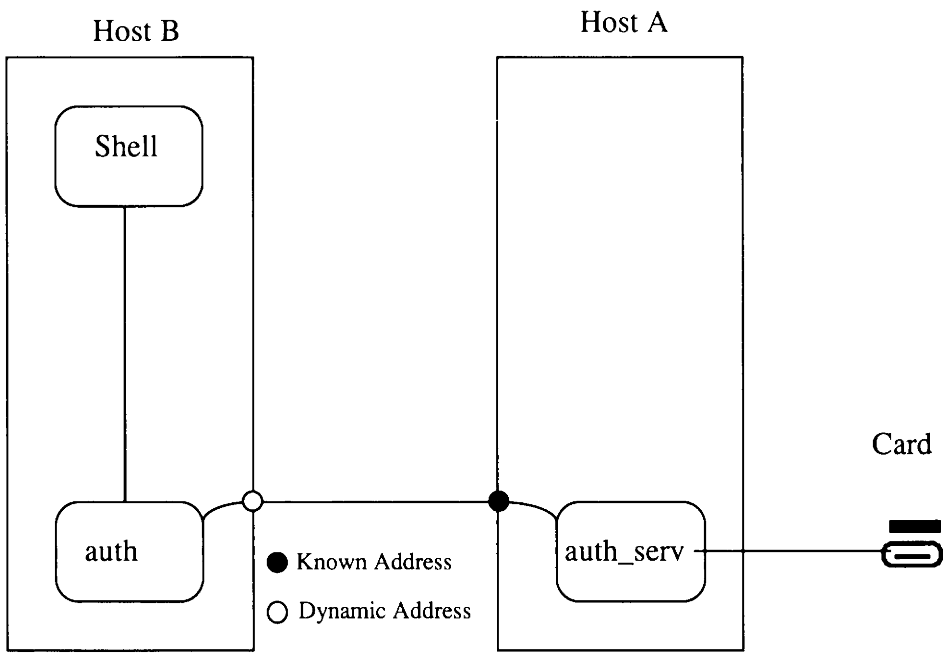


**Figure 3**: *Communication Paths for Application Authentication*

## 6.2. Application – User Authentication

The login authentication procedures described above establish a mechanism whereby any process that is a child of the user's login shell has a communications path to the card. The path can be used by the application to verify the user's rights relating to it. The application can send a request to the card by writing a message structure to the socket connected to the 'auth' program. This message is sent to the appropriate local or remote card server and then by the server to the card. The message is processed by the card and the result is passed back to the application by the reverse route.

The types of information that can be provided are only limited by the capacity of the card and the authentication requirements of the application. By looking at a number of sample applications it was found that two classes of authentication emerge.

1. Application startup authentication.
2. Repeated authentication.

Most applications that require authentication will perform some transactions at startup to determine the vaildity and access rights of the user. For many applications this is sufficient. However, for applications providing many facilities or those that need to ensure that the same user is still present, repeated authentication, per operation or time interval, will be required.

The effect of the authentication system's performance during startup authentication will be a slight increase in the application's startup delay. Unless a large volume of data is required this increase in delay is unlikely to be significant. For repeated authentication the performance of the system is more important. The user will not accept a noticeable decrease in the system's performance, this will be particularly true for interactive applications.

## 6.3. Authenticated Shell

To investigate the performance effect of repeated authentication to a card on a remote system, card authentication was added to a version of the UNIX csh program. The authentication added was a simple user verification requiring the card to acknowledge that the user's identification was one that was registered to use the shell. This requires a message containing the application's identification and the user's identification to be passed to the card. The card checks this information against its stored tables and the success/failure message returned to the application. This authentication was added into the main command processing loop of the csh so that it was performed before any actions were executed.

The actions authenticated include internal csh operations like **alias**, external commands like **vi** and also any commands from shell scripts including **.cshrc**.

### 6.3.1. Authentication Rules

The following rules were used in the implementation of the authenticated shell.

1. If the authentication was successful the command was performed.
2. If the authentication was rejected the command was not performed.
3. If the authentication errored it was retried two times and if still in error the command was not performed.
4. If three successive commands were not executed the shell exited.

### 6.3.2. Authentication Performance

Procedures to measure the performance were investigates but not performed in depth as the time take for the authentication was found to be appreciably less than the variances in command executions times due to changes in system loading. From this it can be concluded that if authentication is required then it can easily be added by the above method without noticeably effecting the performance of the application.

## 7. Future Developments

The are two areas of further development that are currently being considered, to improve the security of the current system and to increase the facilities offered for application authentication.

Providing that sufficiently secure authentication is actually performed between the applications and the card the greatest weakness of the existing systems is the network communication. The security of the original SUN RPC mechanisms is poor as no encryption is used and only weak checking performed. These problems have been tackled with the introduction of a secure RPC mechanism. Therefor the security of the card based system would be noticeably increased by using a secure RPC mechanism.

Improvements to the facilities offered by the card to the authentication by applications within a distributed system are currently being performed. The approach adopted is for the card to store, in encrypted form, the capabilities of its user. These capabilities are retrieved by the application, either on startup or as required.

To support a more general approach to application authentication and to allow the system to be used by users with conventional logins, access to the card server is allowed by secure RPC direct from the application rather than through the auth process.

## 8. Conclusions

The results of the experimental implementations show that intelligent card can be easily incorporated into the existing UNIX login system and that the resulting configuration potentially offers a greater level of security.

The performance of local logins is acceptable but adding an intelligent card allows greater control with improved authentication methods. The performance of remote logins is greatly enhanced as the existing weEkness, of unencrypted passwords on the network, is removed. The choice of the authentication technique used between the systems and the card is the subject of further research, however the method used in the implementation is effective but weak to a masquerade attack.

The performance of the system was very satisfactory, the increase in login time was far less than the variation in login times due to changes in system and network loadings.

Extending card based authentication to applications allows a greater range of facilities to be provided. With the use of a secure RPC mechanism an application will be able to perform complex authentication and access control operations in a secure and efficient manner without the need for interaction with the user.

## References

GEC86a. GEC, *IC Card Programmer's Guide*, GEC Card Technology, Chelmsford, Essex, England, 1986.

Lei85a. Barry Leiner, Jon Postel, Robert Cole, and David Mills, "The DARPA Internet Protocol Suite," *INFOCOM'85*, pp. 28-36, IEEE, 1985.

Mor79a. R. Morris and K. Thompson, ''Password Security: A Case History,'' *Communications of the ACM*, vol. 22, pp. 594-597, ACM, Nov. 1979.

Pos80a. J. Postel, *Internet; User Datagram Protocol,* RFC 768, USC/Information Sciences Institute, Aug 1980.

Pos80b. J. Postel, *Internet; Transmission Control Protocol,* RFC 761, USC/Information Sciences Institute, Jan. 1980.

Sun86a. Sun, *Network Services,* Sun Microsystems Inc, Mountain View, CA, 1986.

Sun86b. Sun, *Remote Procedure Call Programming,* Sun Microsystems Inc, Mountain View, CA, 1986.

Sun86c. Sun, *External Data Representation Protocol Specification,* Sun Microsystems Inc, Mountain View, CA, 1986.

Svi85a. Jerome Svigals, *Smart Cards, The Ultimate Personal Computer,* Macmillan Publishing Company, New York, 1985.

# Into the Padded Cell

*S. J. Hinde, D. Colman, A. McPherson, A. P. Standford.*

Information Technology plc, UK

*ABSTRACT*

Information Technology plc produce secure UNIX products for the military and commercial markets. The secure development team reveal some of their experiences in securing UNIX systems.

Following the principle that the highest fences of the protective measures should be placed at the outer defensive limits, the most obvious and cost effective method of reducing the threat to system security is to strengthen the UNIX login process.

Once users are accepted by the system, a strict "need-to-know" principle is enforced. Users are only permitted access to information required for their work. Users are compartmentalised in a "cell" of information. A user attempting to infringe the system security is isolated by the system into a "padded cell", without privilege or access to useful information. Coupled with the requirements for enforcing a security policy is a need to monitor the action of the users – particularly as regards security sensitive events. An audit capability allows this form of surveillance to be achieved.

In standard UNIX, the system is managed by a single omnipotent superuser. Attacks using superuser privilege are potentially the biggest threat to the security of a UNIX system. This privilege can be sensibly partitioned between users of lesser privilege, greatly reducing the risk.

## 1. The Security Threat

The security of information held on computer systems is a growing concern of large and small users alike. Each week, the antics of hackers and other lapses and breaches of computer security make headline news. System managers and prospective purchasers of computing equipment may be justified in feeling uneasy about their own fairly rudimentary knowledge of the strengths and weaknesses of the system they run compared to the in-depth familiarity many of the reported practitioners seem to display. The growing acceptance of standard operating systems such as UNIX as a means of achieving vendor independence is a two-edged sword. On the one hand, it allows large purchasers of computers, such as government departments, defence establishments, or large commercial conglomerates, to maintain a high level of competition in the procurement of computing equipment where common standards are agreed. On the other hand, the very standardisation of the features and facilities mean that shortcomings, particularly in the area of security, are replicated on many of the systems in use. The current emphasis on networking of computer systems exposes any inherent weaknesses to a wider audience.

The purchaser of any secure computer system will generally require the system to provide some measure of protection against

- the unauthorised disclosure of sensitive information.
- the reduction of the integrity of the stored information, by unauthorised creation, alteration, or deletion of data.
- the unintended denial of services offered by the system.

Attacks on a computer system may be active or passive, or a combination of both. Active attacks involve the use of the system resources in some unauthorised way. For example, one user may attempt to masquerade as another. A hacker may attempt to masquerade as an authorised user. Valid use of a system terminal may be illegally recorded and replayed at a later date. A user may interfere with the system software used by other system users, planting a virus or trojan horse. An intelligent workstation or personal computer may be substituted for one of the system terminals allowing sensitive information displayed on the terminal to be copied to magnetic media and taken outside the security perimeter of the system, beyond the reach of physical controls applied at the site.

Passive attacks rely on exploiting leakage paths for the extraction of sensitive information, including discarded printed material or consumables, discarded equipment and storage media, emitted electromagnetic radiation, traffic analysis, wire-tapping, and so on. Attacks of this nature are particularly difficult to detect.

Counter measures in a computer system which address these threats are generally a combination of physical security measures and computer mediated measures. The emphasis in the remainder of this paper is on the computer mediated, self-protective features built in to a secure version of the UNIX operating system.

## 2. The Standardisation of Security Features

The widespread use of computers in government and defence has led to the establishment of national authorities for the setting of standards for the use of computers holding information affecting national security. The US National Computer Security Centre (NCSC) sets standards for secure systems used by the US Department of Defense and Intelligence agencies, and performs evaluation of systems against these standards. In the UK, the Communications and Electronics Security Group (CESG) perform a similar function for UK government and defence systems.

Another government department in the UK, the Department of Trade and Industry, has established a Commercial Computer Security Centre (CCSC) to address the security of products used in commercial systems, to set standards and perform the security evaluation of products in isolation.

Secure systems offered by vendors who wish to sell the same product in both the defence and commercial market places will generally have to conform to the minimum criteria in the US DoD Trusted Computer System Evaluation Criteria (known as the Orange Book) at some chosen level. The levels identified in the Orange Book are widely referred to outside the US in both military and commercial contexts. Individual systems will frequently require additional features, specific to a site, which reflect the physical environment in which the system is to be used.

Level B1 in the Orange Book is the lowest security level which permits information at different security levels to be held on the same computer system. The B1 level is entitled "Labelled Security Protection" since the storage items must be labelled with the security level of the information they contain. The separation of multiple security levels is a necessary feature of systems used in government which may not have much relevance in commercial circles.

The security level below B1 in the Orange Book is level C2 entitled "Controlled Access Protection". At this level, all information is assumed to have the same security level corresponding to the most sensitive data held on the system. Most of the functional requirements in the C2 level have equal applicability to commercial and military systems. C2 systems are therefore attractive to vendors looking for wide markets for their secure products.

## 3. Layers of Protection

Figure 1 is a diagramatic representation of the onion-skin approach to system security adopted on ITL's secure version of the UNIX operating system called ITALIX. Each layer of the onion-skin offers some degree of protection to the information held on the system. The major layers are an enhanced user authentication mechanism, confinement to a restricted environment called a cell, Mandatory Access Control, and the provision of trusted functions. Access to an item of information is granted only if all of the protection mechanisms permit. Bypassing one of the protection mechanisms does not guarantee access. Security in depth is achieved through the placement of a number of barriers of different types such that the compromise of any one type of barrier does not yield unrestricted access to anything on the system.

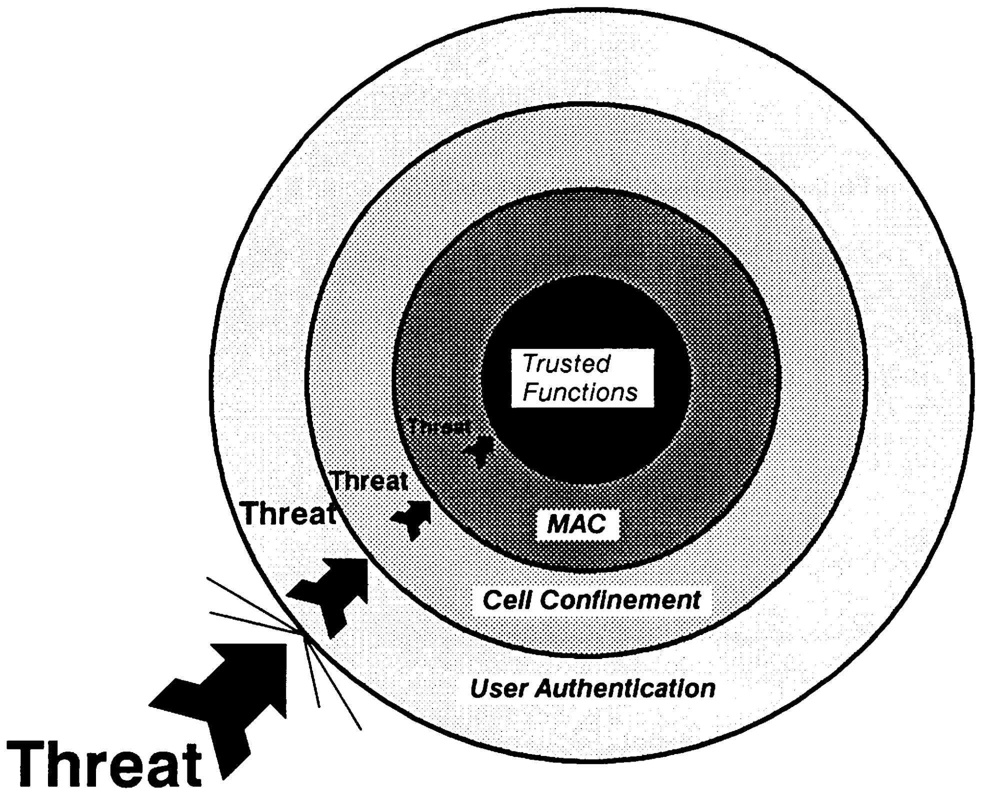


**Figure 1**: *The onion-skin approach to system security*

## 4. User authentication

User Authentication is the first and most important security measure beyond the physical controls. If this barrier is effective, the remaining security measures need only restrict and monitor the activities of authorised users. In ITL terminology it can be seen as the door to the cell.

There are two aspects to user authentication, prevention of fraudulent access and notification of attempts to violate the security. Prevention of fraudulent access must apply equally to all methods of access to the system, from directly connected terminals and LAN connections, to dialup lines and remotely entered jobs from other machines.

Before accepting an attempt at login, the system must ensure that the terminal connected to that path is known to the system and is permitted to perform the login. This measure is intended to prevent the unauthorised introduction of intelligent terminals capable of mounting a systematic password attack, or copying to some removable medium data that must not be moved from the system. The devices which are authorised for use on the system are identified in a list, created and maintained by the system administrator like other system privileges.

Protection against the substitution of one terminal for another of a different type can be achieved by arranging that each terminal is capable of responding to a request from the host system with a unique identity, checked each time a connection from the host to the terminal is established.

If the terminal is valid, the login may be allowed to proceed. The user must then respond to prompts, supplying a valid login name and password pair. The password can be seen as the key to the door of the cell.

Enhancements to the password management features provided on standard UNIX systems are frequently required. The names of wives, children and car registration numbers tend to be chosen with depressing regularity in the average human-generated password file. To avoid the problem of such predictable passwords, all user login passwords should be machine generated. This feature is supported on secure ITALIX systems.

Repeated failure to supply a valid password causes that user name to become "locked out", and the security officer is notified. Repeated login failures at a specific terminal results in that access path becoming locked out. The locked out status of the user name or terminal is not made apparent to the user; a login prompt will be continue to be presented, thereby allowing the security officer an opportunity to investigate the problem.

For particularly sensitive roles or commands, logging in as the system administrator or the security officer for example, 2-man control may be required. In this case, two or more independent user/password combinations are required to authorise the access.

A further level of security can be achieved by restricting the use of individual terminals to a pre-defined list of authorised system users. The list can be made as general or as restrictive as the situation demands. Attempts by a user to login at a terminal for which he is not authorised is a security relevant event which results in immediate notification to the security officer. This notification takes the form of an ITALIX warning message displayed and periodically refreshed on the security officer's terminal until explicitly cancelled.

## 5. Cell Confinement

An important aspect of secure systems is the support of a need-to-know policy in which users of the system are permitted access only to the information they need to carry out their authorised tasks. Many of the computer system resources can be allocated on the basis of project groups, the members of which have a similar need-to-know requirement since they all work towards a common goal. A system should ideally support a number of projects concurrently, each isolated from the others. The system itself should ensure the separation between project groups, and deny access by members of one group to information belonging to another. One scenario might be in an Automated Office environment where the need-to-know groups would correspond to the company departments, for example the Personnel Department, the typing pool, the Managing Director's Office etc.

ITL's secure ITALIX operating system provides a separation mechanism called "cells". A cell is a self-contained UNIX environment to which a group of users with a similar need-to-know requirement are assigned. To these users, the system appears to be a UNIX system dedicated to the users within the same cell. The cell has the standard UNIX directories necessary for conformance with the standard interface definitions. The users within the cell can communicate with each other using the normal UNIX mechanisms using shared memory and IPC facilities. Items of electronic mail may be passed freely between members of the cell. Individual users can manipulate the normal set of permission flags to restrict access to private files on the basis of owner, group, or public controls.

The cell structure is illustrated in Figure 2. The pathnames quoted by users within a cell are interpreted by the operating system kernel as relative to the cell root. Users therefore have the capability of only addressing file system objects within the confines of the cell. Objects outside the cell cannot be accessed since they cannot be named.

In fact, the cell allows access only to a restricted portion of the overall directory structure. A node identified as the cell root, defines the upper addressing limit for users assigned to that cell. All pathnames quoted within the cell are assumed relative to the node defining the cell root. Multiple cells may be allocated on the system. The allocation process ensures that the various cells are distinct and do not overlap.

The presence of users operating within a cell is not generally made apparent to users in other cells. Utilities such as "who" and "ps" present only information pertaining to other users in the same cell, further reinforcing the concept that each cell appears to be a dedicated machine.

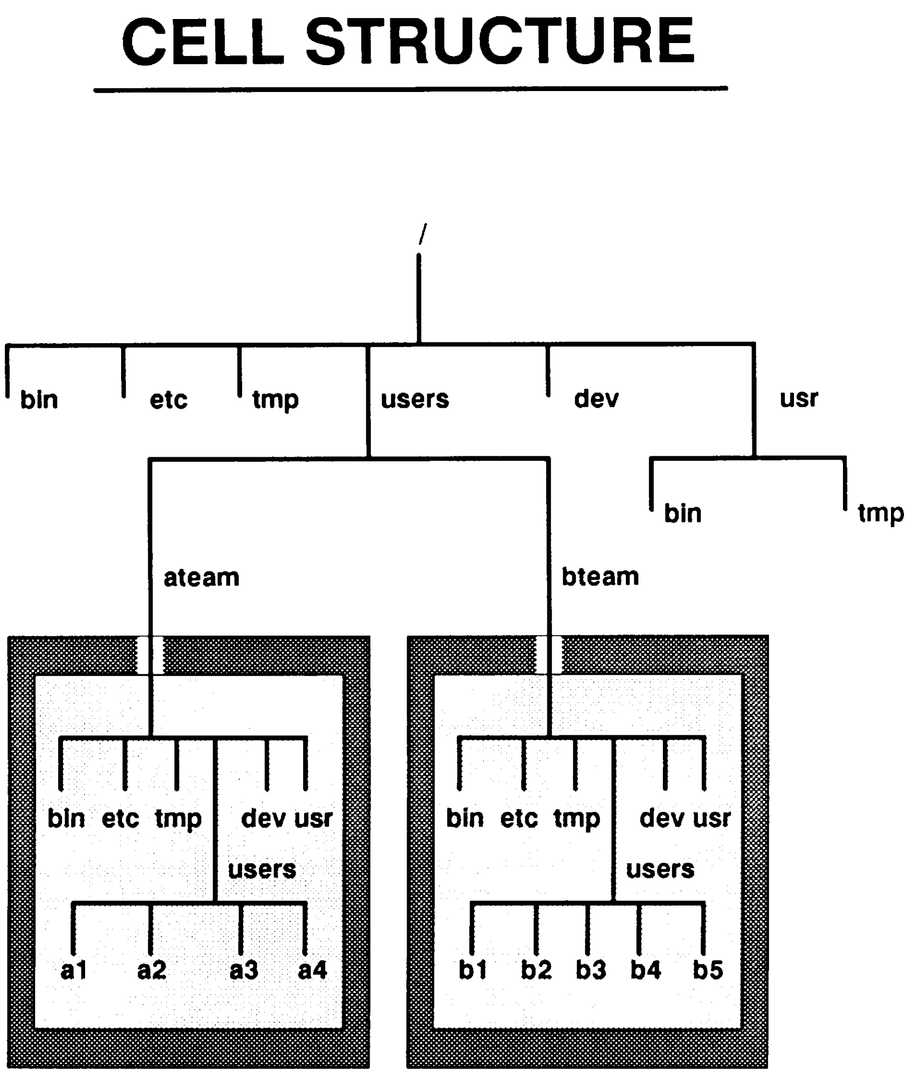


**Figure 2**: *The "cell" structure*

## 6. Access Control

The control of access to the information held on a secure system takes two main forms – Discretionary Access Control and Mandatory Access Control.

Discretionary Access Control allows the owner of an object holding information to determine the level of access to that information permitted to other system users. The familiar set of access permission flags assigned to self/group/public categories are typical of the expected form of discretionary controls. In some respects however, those features provided in the standard UNIX product do not cover the more stringent requirements of a highly secure system. In such systems, the originator of a file may require to maintain discretionary access control over a file even though the file owner may change through copying or distributing. The three permission flags indicating read, write and execute provide only a basic level of control. Desirable extensions include print, copy, delete, annotate, and distribute permissions.

Mandatory Access Control places restraints on accessing information enforced by the secure system itself. This form of access control is relevant to systems which hold information at different security levels using security labels as a basis for deciding whether access to a file by a user should be granted. The separation between the various security levels is enforced by the system itself and in general cannot be overridden by the normal system user. The Orange Book defines a set of rules to be adopted in applying Mandatory

Access Controls, and reflect the concern of the military establishment to maintain confidentiality and prohibit downgrading of the information to a lower security level. The Orange Book security policy is a statement of the wishes of a single, albeit large, customer and does not necessarily satisfy others, particularly in the commercial field.

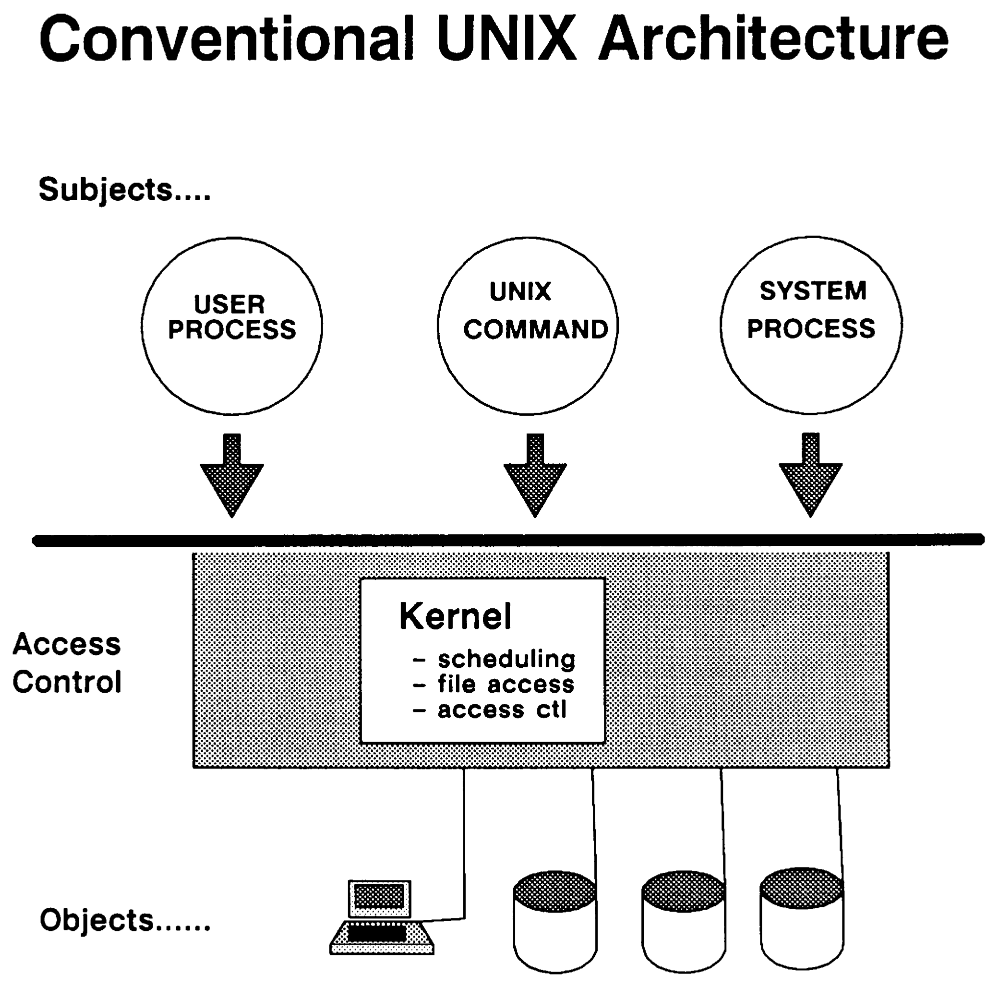


**Figure 3**: *Conventional UNIX Architecture*

## 7. Trusted Software

The "Trusted Computing Base" is that part of the system software responsible for enforcing the system security policy and protecting itself against unauthorized tampering.

In a UNIX system the trusted software is not confined to a small nucleus of the system, nor is it particularly well protected. The Trusted features of the system are split between the UNIX Kernel and functions that run in user space as application programs. These user space based programs consist of "daemons" running at all times performing system management functions such as system accounting, and a set of privileged utilities for managing the system. The names of these privileged utilities and some of their global files are visible to ordinary users of a UNIX system. For example any user of a UNIX system can see the encrypted version of all the passwords in the system and the login names of all authorised users. Figure 3 shows the standard UNIX system.

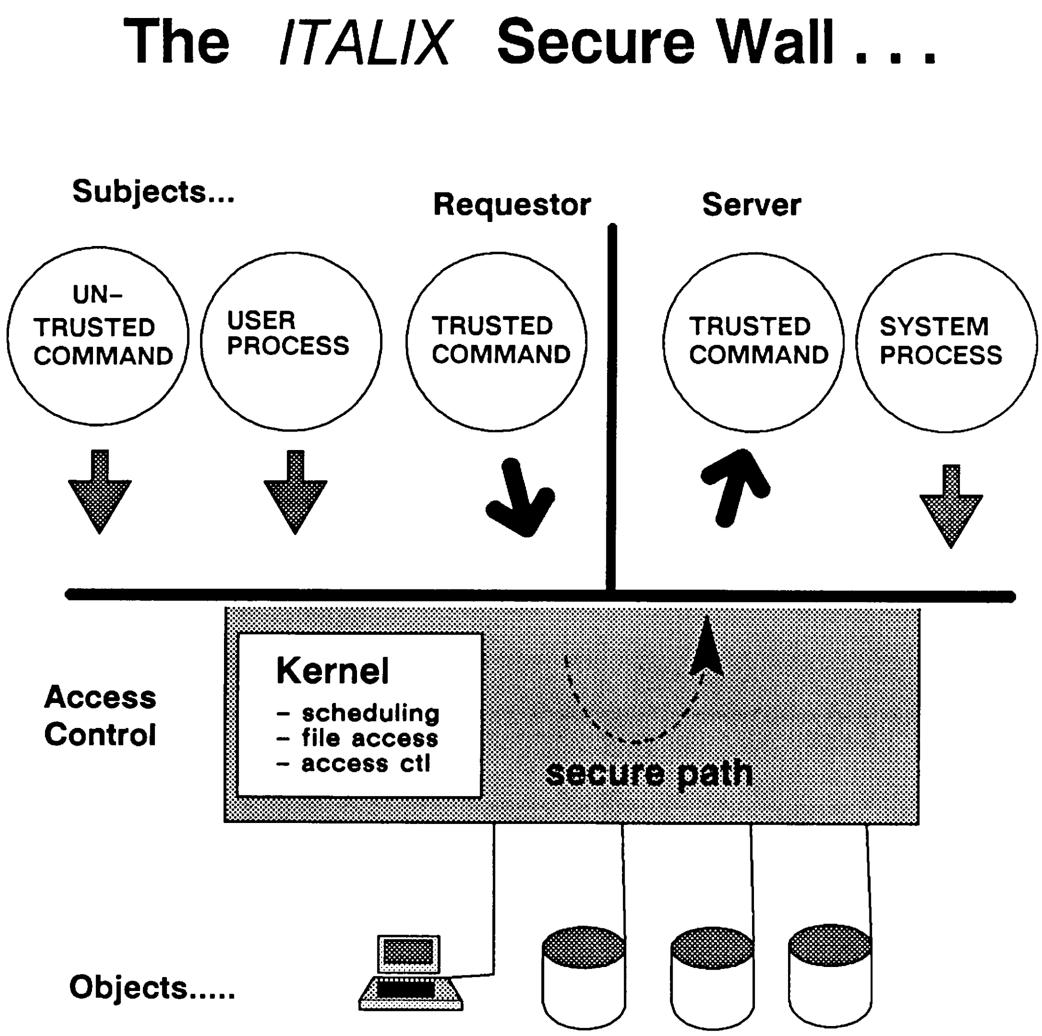


**Figure 4**: *The ITALIX Secure Wall*

A better approach is to encapsulate all trusted software in a domain of the system that is not accessible to the normal system users. Obviously a user interface to the trusted software is required but this should be carefully guarded. Figure 4 shows a the approach to guarding the trusted software in a UNIX system, adopted by the secure ITALIX system. Users who have been granted the privilege to access system management functions do so using a new interface. Requests to the trusted software are passed from a "client" to a proxy "server" process via a "secure path". The UNIX kernel attaches information to the request giving the clients credentials for making the request before forwarding the request to the "server".

IPCs (Inter-Process Communications) allow the exchange of information between two processes. This exchange of information needs to be restricted to two processes at the same security level. This may be achieved by either labelling the IPC channel, or by labelling the users and ensuring that the users at each end of the IPC have the same security label. The secure ITALIX system adopts the labelling of users approach.

## 8. Into the Padded Cell

In addition to the software enforcing the security policy, there is also software in the system that could potentially subvert the security of the system simply because it runs in a trusted environment. Many attacks on UNIX systems are made by planting a Trojan Horse in the domain of a privileged user, for example by substituting the "cat" program by a program that performs a subversive activity and then performs the actions normally associated with "cat". Once the superuser has logged in and typed "cat" the security of the system is compromised.

## 9. Administrative Roles

The standard UNIX superuser is an omnipotent system user with responsibilities for all aspects of system management. This concept is not consistent with the requirements for high levels of security. On such systems, the management responsibilities must be divided among a collection of users who play specific roles. The ITL secure ITALIX system allows three administrative roles to be defined, the default roles being that of System Manager, Security Officer, and System Operator.

The System Manager is the only one of the three roles who must login as root. His responsibilities are limited to the management of cell configuration and 2-man control features. The low level of activity related to this role limits the exposure the root password.

The Security Officer is responsible for the examination and destructive manipulation of audit trails, and for re-enabling user access to the system. The System Operator has responsibility for day-to-day loading and archiving of the system, and the restriction of access to the system by the imposition of lockout status if necessary.

Each of these administrative roles can be assumed by a one of a nominated group of special users. As such, each role is implemented as a unique reserved name. The nominated members for each role must choose the role name at login time, and further qualify the login request with his own login name and password. Each user has therefore only a single password to remember irrespective of whether he wishes to login in an administrative role, or as a normal system user.

## 10. System Monitoring

One much publicised method of attack on computer systems is the "virus" for which monitoring is an effective defence. A virus is a malignant self-reproducing code corruption that propagates itself throughout the system. By running a periodic integrity check on all the code that enforces the security of the system, a virus can be quickly diagnosed and eliminated. This type of integrity check could checksum all code and compare it against a bonded checksum, or alternatively compare all the code against a bonded copy. This bonded copy of the system code could be held on a "WORM" to eliminate the possibility of tampering.

The ability to monitor the activities of a secure system is paramount. As well as protecting the system with various defences and countermeasures, it is necessary to monitor all activity on the system that are security sensitive.

The primary method of monitoring system security is to record events pertinent to system security in an indelible, permanent record, termed the system "audit trail". In the event of a security violation or anomaly, the cause can be diagnosed from the record.

Many pitfalls lie in store for the implementor of a security audit trail. At high levels of security, the quantity of information gathered is the source of most of the problems. More specifically the following problems exist:

(i) An audit trail imposes a performance overhead on the system. Particularly affected are applications that are I/O intensive. Most audit regimes cite I/O activity as an important contributor of audit information.

(ii) The storage, management, and archiving of the audit files can demand a lot of attention from the system administrator. What archiving media is appropriate? How long should the record be kept for? What archiving regime should be used? What is the procedure to be followed if and when the audit trail uses up all the available disc space on the system? How is this wealth of information to be analysed?

(iii) Without effective analysis, the capture and storage is a futile activity. Security violations could remain unnoticed while the system administrator sits buried under megabytes of audit file.

There are no definitive answers to these questions which cover every system. The regime appropriate depends very much on the level of security required. The quantity of audit information gathered and the length of time it is kept is a matter of balancing risk against cost. At high levels of security the cost of media and loss of performance is insignificant compared with the consequences of not identifying security violations. In such systems, a regime must be imposed that saves all audit information for all time identifying all actions performed by the system. In a standard office environment with a much lower security profile, the threat may be the occasional attack from the local hacker or disgruntled employee. It may be sufficient to retain each audit trail file for a couple of weeks while the system administrator investigates the problem. There are however a number of well understood techniques for easing the problem.

(i) Optical media are ideal for this type of application. They have high capacity and are available in write-once form. Also, optical media allow fast random access to the audit trail for analysis purposes.

(ii) The audit trail can be held in encoded binary form, using compression techniques to conserve space on the storage media.

(iii) The audited event types can be selected to meet the security needs of the individual system, which may change from week to week. If the auditing can be dynamically tuned and tailored, the size of the audit file can be kept to a minimum compatible with the security requirements.

(iv) The analysis of the audit trail should be carried out using tools that can scan for particular combinations of events. Tools based on Artificial Intelligence techniques are now emerging, the more sophisticated of these build up a knowledge base of how audit information correlates to security threats. A sophisticated analysis tool could run at all times in a system and dynamically tune the resolution of the system auditing to match the perceived threat.

Although audit trail accumulation and analysis is an essential facet of secure system monitoring, there are other monitoring techniques which can complement it. By careful design, it is possible to build "trip wires" in the system software that trigger as security sensitive events occur. This technique will allow the system administrator to be informed immediately as potential security penetration occurs.

## 11. The Padded Cell

The security requirement that accountability can be maintained to identify the actions of individual users of the system implies that each authorised system user must be assigned a unique login name and associated password. The sharing of login names between users or between system administrators is a feature permitted in a standard UNIX system which is not acceptable on a secure system.

Having established a regime in which each user can be identified individually, the operating system is in a position to detect and prevent some types of system penetration attempts. If a user attempts to login quoting a login name and password corresponding to a currently logged in user, something must be wrong. The user authorised to use the login name may be either the one currently logged in or the one attempting to login. Either way, one or other of the users is using a name for which he is not authorised. Since the login name and password form a valid pair, it is most unlikely that the event has occurred by innocently mis-typing the reply to the login prompting sequence. Rather, a deliberate attempt to gain unauthorised entry to the system is indicated.

The action taken by the system in these circumstances has to take account of the uncertainty of which of the two users is the one attempting to masquerade as the authorised user. At the same time, the record of events in the audit trail should distinguish the actions of the user already logged in from the new user attempting to gain entry. Furthermore, the new user should not be permitted access to the files and data belonging to the authorised user.

A possible defensive approach adopted by the secure ITALIX system is to raise an alarm to attract the attention of the system administrator whilst concealing as far as possible the detection of the event from the two users concerned.

The existing first user is allowed to continue his session unaware of the actions taking place. The system has however flagged the login name as "Locked Out", so that once the user terminates the current login session, no further sessions will be accepted for that user until positive action is taken by the system administrator to release the lock.

The second user is positioned in an isolated dummy cell in which no write access to any data files is permitted, and only a benign set of system commands is made available. There are no private data files in the cell, and the user is prohibited from creating new files. The system has accepted the login request but placed the user in a "padded cell" in which no access to information held on the system is permitted, and communication with other system users is prohibited. The system administrator is alerted that a security violation has occurred and can then investigate while the two possible suspects remain logged in, the rationale being that the second user logged in as the padded cell will be delayed sufficiently by this deception to enable his apprehension.

## 12. References

[1] US Department of Defense Trusted Computer System Evaluation Criteria, DOD 5200.28-STD (The Orange Book).

[2] Dr. D. H. Barnes et al, *The DTI Commercial Computer Security Centre at RSRE Malvern*, DP027/3.3, System Security 1987.

[3] *ITL ITALIX-40 Security Option System Administrator's Guide*, Doc. No. 30894.

# UNIX Standardisation: An Overview

*Cornelia Boldyreff*

Department of Computer Science
Brunel University
Uxbridge UB8 3PH
United Kingdom
*corn@cs.brunel.ac.uk*

*ABSTRACT*

A short history of UNIX standardisation is given here together with an assessment of the current state of affairs vis a vis UNIX standardisation. An outline of future developments of the UNIX related standards is given; and some general trends in standardisation are discussed with respect to the development of UNIX standards.

## 1. Background

With the success of the UNIX Portability Project [JOH78] in the late seventies came a more portable edition of the UNIX operating system as well as tools developed to support UNIX and C portability. The goal of the project was to support application portability; there was an interest in making the software tools developed on the UNIX operating system more widely available on other operating systems. At first it was thought to be sufficient to have a portable version of the C compiler; however, the UNIX software tools made use of UNIX features not provided on other host operating systems, so a radical solution was adopted: to develop a portable version of the UNIX operating system as well as the C compiler. This project laid the groundwork for the promulgation of the UNIX operating system and the C programming language beyond the PDP-11 hardware with which they were originally developed onto a host of hardware and software platforms.

Not only was UNIX ported onto other hardware systems as an alternative operating systems; it was in some cases provided as a subsystem within another operating system. As these more portable versions of UNIX and C emerged simultaneously with a number of new microprocessor based architectures, UNIX was seen by many as providing a tailor-made solution to the problem of operating system provision.

Whilst C provided the basis for UNIX portability, the C language also took on an independence from UNIX with the provision of C implementations on other operating systems and as stand-alone systems.

Some of the credit for the popularisation of UNIX and C beyond Bell Laboratories and the academic and research institutions which constituted the UNIX and C user base in the 1970s must go to the user groups which played an active role in furthering UNIX and C development as well as being a source of UNIX and C expertise. As the user base of UNIX and C grew, the necessity for standards became more apparent, and it was through the efforts of one of these groups, /usr/group, that work on UNIX standardisation was first initiated.

## 2. The Work of /usr/group

The Standards Committee of /usr/group had as its goal the development of a standard that

> *"...describes the external characteristics of computer operating systems that are functionally compatible with the UNIX operating systems With the success of the UNIX Portability Project [USR84]*

This work was motivated by the need to ensure portability of applications across the range of emerging UNIX implementations. As UNIX was ported to other architectures, various enhancements were introduced. The result for users was unfortunately that the uniformity of UNIX was compromised and application portability could no longer be assured. Thus, the case for standardisation was apparent; particularly the need to define the interfaces used by applications as these were recognised as the key to ensuring application portability. The early pioneering /usr/group Standard work has been subsumed by IEEE P1003's POSIX Standard and ANSI's X3J11 C Standard work described below. This group has continued to play an active role in the formation of the POSIX standard; their work on popularising the standard and educating management regards the significance of standards in this area has been particularly useful.

## 3. The AT&T Development of the SVID

Even within AT&T a diversity of UNIX implementations had developed as result of its widespread use within the AT&T organisation and release of UNIX products, such as System III and later System V. With System V, AT&T sought to define the source-level interfaces that are consistent across all System V environments; the result was the SVID - the AT&T System V Interface Definition, Issue 1 [ATT85]. From the start, this was an AT&T product definition; the first issue carries the following footnote:

> "* *This document is an official publication of AT&T and its content is wholly based on material from AT&T documentation.*"

This edition of SVID defines the functionality of the AT&T UNIX System V Release 2.0 product with the additions of advisory record and file locking. It contains an acknowledgement to the Standards Committee of /usr/group recognising their contribution to *the effort to define an interface standard for operating systems*; and acknowledges that through participation in the /usr/group standardisation work, ideas were exchanged which were reflected in the SVID.

In their continued support of UNIX standardisation, AT&T have been major contributors to the development of the POSIX standard whilst at the same time maintaining their own product definition, the SVID [ATT86].

## 4. The X/OPEN Approach to UNIX Related Standards

Originally the X/Open Group was a consortium of European UNIX System suppliers who sought to define a Portability Guide for UNIX software developers based on the SVID and other higher level application interfaces such as programming languages, graphics, database functions, etc. They termed this collection of defined interfaces, the Common Application Environment (CAE); and their aim was to provide the means whereby applications could be developed which would compile and run on a number of UNIX systems without change.

With the development of the POSIX standard work, X/Open has replaced the base of their CAE with the POSIX standard. They have also actively participated in the formulation of the POSIX standard. With their goal of establishing a package of related standards to underlie portable applications, X/Open has become an organisation separate from its original founders companies; and they offer a portable application *kite-marking* service: allowing applications found to adhere to the standards and definitions specified by X/Open to carry the X/Open symbol.

## 5. The IEEE POSIX Initiative

As the interest in UNIX standards grew, it was recognised that an official standard for UNIX was required. Consequently work on UNIX standardisation was initiated within the IEEE, one of the professional bodies recognised by the American National Standards Institute as an accredited standards making organisation. The earlier work of /usr/group, the *'1984 /usr/group Standard'*, was taken as the base document by the working group formed within the IEEE to address UNIX standardisation, 1003. This committee is sponsored by the IEEE Computer Society's Technical Committee on Operating Systems; and it has from the start been open to any member of the public who wishes to participate in its standardisation work. The remit of the 1003 working group is to to standardise a Portable Operating System Interface for Computer Environments (colloquially known as POSIX) based on the UNIX operating system documentation. Initially its focus has been the C language operating system interface required to support portable applications at source code level.

There have been a number of guiding principles adopted by the developers of the POSIX standard; these are given in the Rationale which accompanies the standard [IEE88]. They are as follows:

Application Oriented.

Interface, Not Implementation.

Source, Not Object, Portability

The C Language and X3J11. (Co-ordinate with related standards work)

No Super-User, No System Administration.

(Limit scope initially)

Minimal Interface, Minimally Defined.

Broadly Implementable.

Minimal Changes to Historical Implementations.

Minimal Changes to Existing Application Code.

IEEE Consensus Process.

IEEE Balloting Process.

WeirdNIX *or destructive QA of a standard*

The last principle was an interesting idea implemented during the evaluation of the trial-use standard; people were offered prizes for *the best new and technically legal interpretation of the POSIX standard which nevertheless violates the intuitive intent of the POSIX standard.*

The first draft trial-use standard published by the IEEE for comment and criticism was issued in April 1986 with the proviso that its distribution for comment would not extend beyond one year. In order to facilitate wide-spread distribution, the draft was available from the IEEE and ANSI as well as Wiley-Interscience [IEE86].

The IEEE Working Group formulating the POSIX standard includes staff from all the major US computer companies in the UNIX market.

UNIX user groups (USENIX, /usr/group) and US government and military users are also represented on the Working Group; and there has been some participation from outside the USA notably from the X/Open Group. In the Acknowledgements concluding the P1003.1 text, over 200 organisations are thanked for their contributions to the Working Group.

The IEEE through ANSI opened their work to the standards making bodies of other nations by proposing that the International Organisation for Standards (ISO) initiate a New Work Item based on the P 1003.1 effort.

Their aim was to facilitate international participation in this work leading to its adoption as an international standard. In the spring of 1986, approval was given for ISO work on POSIX. Since its formation the ISO Working Group on POSIX has worked in parallel with the IEEE working group with the aim of developing a single standard. When the POSIX work was first discussed at ISO level, there were some reservations concerning its proper place and relation to existing work within ISO [MEE87]. Earlier work on an Operating System Command and Response Language (OSCRL) by another group within ISO was recognised as having a contribution to make in the further development of the POSIX standard. It was also recognised that POSIX could benefit from a more abstract, language independent, definition with associated language bindings similar to the way in which the Graphical Kernel System (GKS) [HOP83] standard has been framed. Work in progress with respect to an Ada binding for POSIX is described in [BOL88d].

The Trial Use Standard with several rounds of revisions became an IEEE standard in the autumn of 1988. Detailed reports on the progress of the standard from the time it was proposed as a New Work Item at ISO level to date can be found in the EUUG Newsletter and the British Standards Institution (BSI) Information Technology Services Newsletter (BITS) [BOL87a, BOL87c, BOL88, BOL88b, BOL88c, BOL88f]. The IEEE approved POSIX standard P1003.1 has gone forward for approval as an ANSI standard; and for registration as a Draft International Standard (DIS) at ISO level. During 1989, there will be a formal six month ISO ballot of the POSIX standard; and with the approval anticipated, the first part of the POSIX standardisation process will be completed.

## 6. Complementary Work on C Standardisation

The POSIX standard is closely related to the Draft Proposed ANSI Standard for C formulated by the CBEMA X3J11 Committee in parallel with the formulation of an ISO C standard [BOL 87, BOL 88, BOL 88a, BOL 88c, BOL88e]. Work on the standardisation of C was initiated in 1983 within CBEMA - the Computer and Business Equipment Manufacturers Association, another body which is accredited as a standards making organisation by ANSI. IEEE P1003 has left the definition of library functions required for a C implementation in any environment to X3J11; that is POSIX refers to the C Standard for these. The C standard in turn does not define operating-system-specific functions leaving these as the province of the POSIX P1003 Standard. There is active liaison between the P1003 committee and the X3J11 committee who over the past few years have developed a good working relations; and liaison between the corresponding ISO Working Groups as well. In the standardisation of C as with POSIX, the goal is to have a single standard agreed at international level which each of the national member bodies of ISO will endorse. Ultimately, the POSIX standard will simply refer to ISO C where relevant; and in the language independent version of the standard, the C language binding will have the status of any other language binding.

## 7. Form of the POSIX Standard and Proposed Supplements

There are three major components to the base POSIX standard, IEEE P1003.1:

Definitions - this initial chapter deals with terminology used through the standard, general concepts are described informally, and various symbolically named variables and constants are defined.

System Interface and Functions - this forms the core of the standard. These chapters define a C Language Binding for Process Primitives and the Process Environment; Files, Directories and File Systems; Input and Output Primitives; Device- and Class-Specific Functions; and Password Security.

Other Interface issues - Portability; Media Formats; and Error Handling and Recovery.

Within IEEE P 1003, several new subgroups have been formed to develop supplements to the base POSIX standard, identified as P 1003.1. These groups cover the following areas:

Shell and Utilities

Realtime

Secure/Trusted Systems

Network Interface/Distribution Services

System Administration

Ada Language Bindings

FORTRAN Language Bindings

Language-Independent Service Descriptions

A subgroup addressing Verification Testing has also been formed to ensure that once the base standard and its supplements are approved, a means of validating conformance to them will be in place. A subgroup has also been formed to formulate guidelines relating POSIX-based standards and related Open System standards.

## 8. The Need to a Language Independent Specification of POSIX

In its present form, POSIX does not provide a functional specification of a portable operating system independent of any specific language, *ie* in this case C, binding.

Initially because POSIX was seen as a much needed effort to standardise an existing applications environment interface based on UNIX systems and complementary to the C standard work described above, it was recognised that the definition of the standard would be heavily biased towards the C language.

In the longer term, the goal is to abstract the principal concepts of UNIX from their realisation in any particular programming language; thus, in the next version of the POSIX standard, the definition of the interfaces will have a more abstract form independent of any programming language. The rationale for a language independent specification is discussed by Meek [MEE87].

## 9. Future Directions of POSIX Standardisation

As well as developing the supplements listed above, the POSIX work has come to the realisation that the user interface and particularly its support for processing natural language representations must be addressed.

Within the ISO Working Group on POSIX, a rapporteur group on Internationalisation has been formed to co-ordinate work on this important aspect of application interfaces. The problems here are very difficult; not only must the problem of handling multi-byte characters within a single language be solved, in some countries, support for multiple languages is required within a single application.

There is also interest is defining a high level application graphical user interface: a POSIX application interface style guide, so-called *look and feel*; so that application adhering to the POSIX standard all present a common user interface and mode of operation to the application user. Such work could be based on the current proposal to standardise X-Windows with CBEMA's X3H3.

## 10. General Trends in Standardisation and their Implications for UNIX Standards

Within the standards community working on Information Technology, there are three key questions which are being debated:

What are the proper objects of standardisation?
How can standards be most effectively formulated?
Who should formulate the standards?

These questions provide the measure against which new standards work is considered. The usual method of resolving any conflicting views is to give consideration to all points of view and then progress towards agreed common view by consensus. This is necessarily an iterative process; and where consensus cannot be achieved progress is held up.

It is generally agreed that there is a place for both product driven standards such as programming languages traditionally have been, and functional standards such as the Open Systems Interconnection (OSI) programme. In the latter case, the standards resulting from the OSI programme have been reflected in product implementations, but the standards themselves are not based on any implementation. The POSIX work with its long term goal of having a language independent definition and its current goal of being independent of any particular implementation though based on descriptions of the functionality of current implementations falls somewhere between these two classes of standards.

In many standards, an unnatural form of natural language has been used in formulating the standard. The successful use of formal definition techniques within the OSI programme has led to a growing interest in more widespread use of these methods. The case for using one such method, LPG, in the definition of the UNIX File System to obtain a precise standard has been made at an earlier EUUG conference [DEC88]. A benefit of more formal specifications of standards is that they assist in automatic verification of implementations claiming conformance to the standards [FMIS89]. There is a growing interest in applying formal definition techniques in formulating the next version of the POSIX standard.

In discussing **who** should formulate standards, there are the pragmatic aspects to be to considered such as resources available as well as general principles such as ensuring the standardisation process is open to all interested parties. In the past before electronic communication was widely used, it was assumed that standard formulation was best done by groups meeting face-to-face. In practice, formulation of a particular standard was often undertaken by a single national member of ISO to whom the responsibility had been delegated by ISO. This is the way both the POSIX standard and the C standard have been developed with the USA member body of ISO, ANSI, in turn delegating the work to the IEEE and CBEMA respectively. In future, to ensure more direct input from the international community, it is likely this practice of delegating standards work to national member bodies will cease. With respect to POSIX, this is unlikely to make a significant difference as the IEEE has operated an open policy throughout the development of the POSIX standard.

## 11. References

[ATT85] *AT&T System V Interface Definition*, Issue 1, AT&T, Spring 1985.

[ATT86] *AT&T System V Interface Definition*, Issue 2, AT&T, 1986.

[BOL87] C. Boldyreff, "Progress of ANSI/ISO C Standardisation" in *EUUG Newsletter*, Vol 7, No 2 (1987).

[BOL87a] C. Boldyreff, "Review of IEEE Trial-Use Standard Portable Operating System for Computer Environments POSIX" in *EUUG Newsletter*, Vol 7, No 2 (1987).

[BOL87b] C. Boldyreff, "Status Report on the Draft Proposed ANSI/ISO C Standard" in *EUUG Newsletter*, Vol 7, No 3 (1987).

[BOL87c] C. Boldyreff, "POSIX Progress at ISO and BSI Level" in *EUUG Newsletter*, Vol 7, No 3 (1987). (Reprinted as "POSIX - PROGRESS AT THE ISO AND BSI LEVEL" in in *BITS - BSI Information Technology Services Newsletter*, Vol 1, Issue 5 (Aug 1987).)

[BOL88] C. Boldyreff, "Draft Proposed ANSI/ISO C Standard and POSIX Standards Developments" in *EUUG Newsletter*, Vol 8, No 1 (1988).

[BOL88a] C. Boldyreff, "Macro Expansion as Defined by the ANSI/ISO C Draft Standard" in *EUUG Newsletter,* Vol 8, No 2 (1988).

[BOL88b] C. Boldyreff, "Report on POSIX Meeting: 2-4 March 1988, London" in *EUUG Newsletter,* Vol 8, No 2 (1988).

[BOL88c] C. Boldyreff, "Draft Proposed ANSI/ISO C Standard: Developments and *POSIX Standardisation Developments" in BITS Newsletter,* Vol 2, No 5, August 1988. (Reprinted in *EUUG Newsletter,* Vol 8, No 3, Autumn 1988.)

[BOL88d] C. Boldyreff, "ADA/POSIX marriage plans" in *Government Computing,* October 1988.

[BOL88e] C. Boldyreff, "ANSI C Standard and Progress towards an ISO C Standard" in *EUUG Newsletter,* Vol 8, No 4, 1988.

[BOL88f] C. Boldyreff, "The POSIX Standard and Its Future Development" in *EUUG Newsletter,* Vol 8, No 4, 1988.

[DEC88] O. Declerfayt, B. Demeuse, F. Wautier, P-Y. Schobbens, E. Milgrom, "Precise Standards through Formal Specifications: A Case Study: the UNIX File System" in *Proceedings of the EUUG Autumn 1988 Conference,* Cascais, Portugal, 1988.

[FMIS89] "Formal Methods in Standards (FMIS) Working Party" in *FMIS Report,* British Computer Society, London, 1989.

[HOP83] F. R. A. Hopgood, D. A. Duce, J. R. Gallop, D. C. Sutcliffe, "Introduction to the Graphical Kernel System (GKS)" in *APIC Studies in Data Processing,* No 19, Academic Press, 1983.

[IEE86] *IEEE Trial-Use Standard Portable Operating System for Computer Environments,* The Institute of Electrical and Electronic Engineers, Inc, IEEE Std 1003.1, April 1986.

[IEE88] *IEEE Portable Operating System for Computer Environments,* The Institute of Electrical and Electronic Engineers, Inc, IEEE Std 1003.1, 1988. (IEEE Order Number SH12211. Cost $32.00.)

[JOH78] S. C. Johnson and D. M. Ritchie, "Portability of C Programs and the UNIX System" in *The Bell System Technical Journal,* Vol 57, No 6, July-August 1978.

[MEE87] Brian Meek, "UNIX Standardisation: A Bystander's View" in *EUUG Newsletter,* Vol 7, No 3, 1987.

[USR84] *1984 /usr/group Standard,* /usr/group Standards Committee, Santa Clara, California, 1984.

# The Next Generation of UNIX System V

*Bob Duncanson*

AT&T UNIX Europe
+44 1 567-7711
*ukc!uel!bob*

## 1. Introduction

Good afternoon. In this presentation we are going to look at some of the features of the forthcoming release of UNIX System V but before we start on this let us consider a picture of the future.

In the 1990s there will be several major operating environments each with a different approach to the market place.

## 2. Major Computer Industry Platforms in the 1990s

IBM with a closed range of computer architectures spanning micro to mainframe systems. Portability across this range of architectures is clumsy.

Apple with another single vendor closed architecture available only on micro and super microcomputers.

DEC with an integrated architecture with good portability in the minicomputer environment. This is the third closed environment and is only available from DEC.

DOS/OS2 is an open software environment but is limited to the 80x86 Intel processor architecture and is a single user system thus it is only applicable to micro and super microcomputers.

UNIX System V is the open computing environment available across the complete range of hardware and from all major vendors. Software portability is available from micros to super computers via recompilation. Binary software portability is available within given processor architectures. Networks containing equipment from multiple manufactures will be easy to set up and maintain.

To support this picture let me give you some facts and figures about the the UNIX Systems market of today.

UNIX System V derived systems are available from some 285 vendors and across the whole spectrum of computer systems from microcomputers to super computers. In 1987 UNIX System V share of the computer market was 6% and forecasts predict an annual growth of 30% with a 20% market share in 1991.

So we can see that UNIX System V is already the open systems environment of today and is idealy placed to be the open systems environment for the 1990s.

## 3. Agenda

After that bit of future predicting we can go on to look specifically at the up coming release of UNIX System V, Release 4.0. The major thrust of this product is to merge all the versions of the UNIX Operating System into one standard system and we will look at how this is happening.

Secondly we will look at some of the specific features within the new product in the areas of Networking, Real Time, Internationalisation and Operations, Administration and Maintenance (OA&M).

Lastly we will look at two other activities which are going on in parallel with development of UNIX System V Release 4.0 to improve software portability on UNIX System V.

## 4. Formation of a Standard

Before talking about standards let us look at a process by which standards can be formed. All technology starts with some research for which by definition there are no standards. This pure technology if it has a practical application is then used to implement products and if these products are successful, industry groups form to take advantage of and support the product. This leads to formation of de facto Standards. Finally after several years of product availability the de facto Standards begin to be used as the basis to form formal standards and at this time the product becomes a commodity. It can be seen that this picture fits well with the life cycle of the UNIX Operating System.

## 5. System V Interface Definition

Several years ago AT&T saw the need for standards in the computer industry. To facilitate this, AT&T published the System V Interface definition or SVID. This was our guarantee that the System V interface would be preserved in future releases of the system and we have kept this guarantee through several major releases of the system and will continue to keep this guarantee for future releases. AT&T also developed the System V Verification Suite or SVVS which evolves to track the SVID. The SVVS can be used by vendors to check the compliance of their systems to the SVID.

## 6. System V Standardisation

The SVID was used by X/OPEN, /usr/group and IEEE-POSIX as the basis for there standardisation efforts and hence there is a great deal of commonality between these efforts. This means that AT&T can easily track the standards efforts with the UNIX System V product in an upward compatible way thus giving backward compatibility for existing users and adherence to standards for the future.

## 7. UNIX System V Unification

Now let us take a look at how the various UNIX System products fit with these standardisation efforts.

There are 3 major products in the UNIX System market, UNIX System V, Xenix and BSD/Sun OS, these 3 represent more than 75% of the installed base of all computer systems derived from the UNIX System. The goal over the last 2-3 years has been to consolidate the features of these 3 systems into one standard release of UNIX System V. Firstly, with Microsoft, we incorporated Xenix compatibility as part of the UNIX System V/386 Release 3.2 product released this year. Secondly we are folding the Berkeley and Sun OS features with the System V Release 4 product, scheduled for availability mid 1989. Throughout this process AT&T has been committed to standards and open systems through licensing of its products to other computer vendors. This means that UNIX System V Release 4.0 will contain the functionality of Xenix and BSD.

## 8. UNIX System V, Release 4.0 – Objectives

We have talked a lot about standards lets now look at the goals for System V Release 4. There are 4 major objectives for System V Release 4.0 these are:

- To facilitate standardisation by adopting features from other UNIX System implementations.
- Enhance data networking with the addition of RFS and Remote Procedure Calls (RPC).
- Penetrate new application areas and environments such as Real Time.
- Strengthen existing markets through improved internationalisation and administration.

## 9. Standards – Proposed Functionality

POSIX conformance. There are 3 new features in UNIX System V Release 4 to ensure conformance to the POSIX standard.

- Enhancements to signal handling
- Multiple groups and ownership
- Job control.

UNIX System V Release 4 will conform to the X/OPEN CAE which is achieved via some minor adjustments of system parameters.

In the merge of Xenix and BSD into UNIX System V Release 4 some specific areas which will be merged are: both the BSD and Xenix application programming interfaces, BSD commands, a new high-performance file system type, and an enhanced file system architecture derived from File System Switch (FSS) and VNODES.

## 10. Networking Evolution – Proposed Functionality

There are many enhancements to the networking capabilities which were the main thrust of UNIX System V Release 3 these include much wider use of the STREAMS facility, support of both RFS and NFS remote file sharing facilities and remote procedure call facilities.

## 11. Real-Time Support – Objectives

One long standing criticism of UNIX System V was its lack of support for real time processing. To enhance the usability of UNIX System V in this area it was identified that the UNIX System V Release 4 would need to be deterministic and more responsive to applications designated as real time.

## 12. Real-Time Support – Proposed Functionality

This was achieved through several new features, user controlled process scheduling, high resolution timing services and Non-Blocking Asynchronous I/O.

## 13. Internationalisation – Objectives

It was identified some years ago that to be applicable world wide the UNIX System must be able to operate in languages other than English and in all releases since 3.0 work has been done to facilitate this goal.

## 14. Internationalisation – Proposed Functionality

Several major features have been added to UNIX System V Release 4 to advance this internationalisation work.

## 15. Operations, Administration and Maintenaince – Objectives

UNIX System V Release 4 will have new functionality in the area of Operations, Administration and Maintenance (OA&M). This is to increase the suitability of UNIX System V for use in commercial computer centers, especially with respect to remote operations in a heterogeneous environment.

## 16. Operations, Administration and Maintainence – Proposed Functionality

In parallel with the development of UNIX System V Release 4 AT&T has two other areas of work which are Application Binary Interface or ABI and Application Operating Environment or AOE. These two are designed to enhance the portability of applications software. The ABI for binary portability and AOE for high level source portability. Let us first look at the goals of an ABI.

## 17. Application Binary Interface – Goals

The goals of the ABIs are:

- to provide binary compatibility within a specific computer architecture for instance Motorola 68030, SPARC, INTEL 80386 much as DOS does with the 80x86.
- to support any operating system implementation as long as that implementation conforms to the ABI.
- there will be several ABIs one for each major computer architecture.

## 18. Applications Binary Interface – Compared to SVID

Let us expand on the concept of ABI by first looking at the portability provided by the SVID. The SVID is a subset of what is defined in the ABI and ensures easy portability for applications across systems with different architectures. The ABI ensures portability within a processor architecture at the binary level.

ABIs are split into two parts the first being generic to all ABIs specifying such things as system call numbers, the second part is specific to each computer architecture being a super set of the processor reference manual. The outcome of this is that software vendors need only one binary version of a software product for a given computer architecture. For instance one for all Intel 80386 machines running UNIX System V one for all Motorola 68030 machines running UNIX System V and so on. AT&T are putting processes in place to work with other computer/processor manufactures to define ABIs.

## 19. ABI – Benifits

The final outcome for end users is that they will be able to go into a computer store and buy application x for processor y UNIX System V and be sure that it will run immediately. Users will also have a wider range of software packages to choose from given that the application developers are attracted by the large market potential offered by the ease of portability. Lastly due to the software portability across architectures users will be able to buy bigger computer systems while still using the same application software, thus saving on software conversion and retraining.

## 20. Applications Operating Enviroment – Objectives

The ABI does not replace the need for open system interfaces across architectures and AT&T addresses this area firstly with the SVID to define the operating system interface and the AOE, our implementation of the X/OPEN CAE.

The AOE expands on the system interfaces defined in SVID by specifying among other things compilers, user interface, data management and networking interfaces. In this way the AOE builds on the SVID to provide a richer application development environment and a higher level computing platform. The AOE is not a static definition it will evolve to make use of new developments.

## 21. UNIX System V Release 4.0 – Product Plans

The technology to support System V Release 4 will be made available to our customers as soon as possible and to facilitate this we have taken a phased approach as can be seen on the slide.

## 22. AT&T Committment

During this presentation it has become clear that AT&T is committed to Open Systems and is committed to working with the industry to provide the Open Computing platform for today and for the 1990s.

Thank you.

# Current Research by The Computer Systems Research Group of Berkeley

*Marshall Kirk McKusick*
*Michael J Karels*
*Keith Sklower*
*Kevin Fall*
*Marc Teitelbaum*
*Keith Bostic*

Computer Systems Research Group
571 Evans Hall
University of California, Berkeley
Berkeley, CA 94720
USA
*kfall@okeeffe.berkeley.edu*

*ABSTRACT*

The release of 4.3BSD in April of 1986 addressed many of the performance problems and unfinished interfaces present in 4.2BSD [Leffler84] [McKusick85]. The Computer Systems Research Group at Berkeley has now embarked on a new development phase to update other major components of the system, as well as to offer new functionality. There are five major ongoing projects. The first is to develop an OSI network protocol suite and to integrate existing ISO applications into Berkeley UNIX. The second is to develop and support an interface compliant with the P1003.1 POSIX standard recently approved by the IEEE. The third is to refine the TCP/IP networking to improve its performance and limit congestion on slow and/or lossy networks. The fourth is to provide a standard interface to file systems so that multiple local and remote file systems can be supported, much as multiple networking protocols are supported by 4.3BSD. The fifth is to evaluate alternate access control mechanisms and audit the existing security features of the system, particularly with respect to network services. Other areas of work include multi-architecture support, a general purpose kernel memory allocator, disk labels, and extensions to the 4.2BSD fast filesystem.

We are planning to finish implementation prototypes for each of the five main areas of work over the next year, and provide an informal test release sometime next year for interested developers. After incorporating feedback and refinements from the testers, they will appear in the next full Berkeley release, which is typically made about a year after the test release.

## 1. Recently Completed Projects

There have been several changes in the system that were included in the recent 4.3BSD Tahoe release.

### 1.1. Multi-architecture support

Support has been added for the DEC VAX 8600/8650, VAX 8200/8250, MicroVAXII and MicroVAXIII.

The largest change has been the incorporation of support for the first non-VAX processor, the CCI Power 6/32 and 6/32SX. (This addition also supports the Harris HCX-7 and HCX-9, as well as the Sperry 7000/40 and ICL machines.) The Power 6 version of 4.3BSD is largely based on the compilers and device drivers done for CCI's 4.2BSD UNIX, and is otherwise similar to the VAX release of 4.3BSD. The entire source tree, including all kernel and user-level sources, has been merged using a structure that will easily accommodate the addition of other processor families. A MIPS R2000 has been donated to us, making the

MIPS architecture a likely candidate for inclusion into a future BSD release.

## 1.2. Kernel Memory Allocator

The 4.3BSD UNIX kernel used 10 different memory allocation mechanisms, each designed for the particular needs of the utilizing subsystem. These mechanisms have been replaced by a general purpose dynamic memory allocator that can be used by all of the kernel subsystems. The design of this allocator takes advantage of known memory usage patterns in the UNIX kernel and a hybrid strategy that is time-efficient for small allocations and space-efficient for large allocations. This allocator replaces the multiple memory allocation interfaces with a single easy-to-program interface, results in more efficient use of global memory by eliminating partitioned and specialized memory pools, and is quick enough (approximately 15 VAX instructions) that no performance loss is observed relative to the current implementations. [McKusick88].

## 1.3. Disk Labels

During the work on the CCI machine, it became obvious that disk geometry and filesystem layout information must be stored on each disk in a pack label. Disk labels were implemented for the CCI disks and for the most common types of disk controllers on the VAX. A utility was written to create and maintain the disk information, and other user-level programs that use such information now obtain it from the disk label. The use of this facility has allowed improvements in the file system's knowledge of irregular disk geometries such as track-to-track skew.

## 1.4. Fat Fast File System

The 4.2 fast file sytem [McKusick84] contained several statically sized structures, imposing limits on the number of cylinders per cylinder group, inodes per cylinder group, and number of distinguished rotational positions. The new "fat" filesystem allows these limits to be set at filesystem creation time. Old kernels will treat the new filesystems as read-only, and new kernels will accommodate both formats. The filesystem check facility, `fsck`, has also been modified to check either type.

## 2. Current UNIX Research at Berkeley

Since the release of 4.3BSD in mid 1986, we have begun work on several new major areas of research. Our goal is to apply leading edge research ideas into a stable and reliable implementation that solves current problems in operating systems development.

## 2.1. OSI network protocol development

The network architecture of 4.2BSD was designed to accommodate multiple network protocol families and address formats, and an implementation of the ISO OSI network protocols should enter into this framework without much difficulty. We plan to implement the OSI connectionless internet protocol (CLNP), and device drivers for X.25, 802.3, and possibly 802.5 interfaces, and to integrate these with an OSI transport class 4 (TP-4) implementation. We will also incorporate into the Berkeley Software Distribution an updated ISO Development Environment (ISODE) featuring International Standard (IS) versions of utilities. ISODE implements the session and presentation layers of the OSI protocol suite, and will include an implementation of the file transfer protocol (FTAM). It is also possible that an X.400 implementation now being done at University College, London and the University of Nottingham will be available for testing and distribution.

This implementation is comprised of four areas.

1) We are updating the University of Wisconsin TP-4 to match GOSIP requirements. The University of Wisconsin developed a transport class 4 implementation for the 4.2BSD kernel under contract to Mitre. This implementation must be updated to reflect the National Institute of Standards and Technology (NIST, formerly NBS) workshop agreements, GOSIP, and 4.3BSD requirements. We will make this TP-4 operate with an OSI IP, as the original implementation was built to run over the DoD IP.

2) A kernel version of the OSI IP and ES-IS protocols must be produced. We will implement the kernel version of these protocols.

3) The required device drivers need to be integrated into a BSD kernel. 4.3BSD has existing device drivers for many ethernet devices; future BSD versions may also support X.25 devices as well as token ring networks. These device drivers must be integrated into the kernel OSI protocol implementations.

4) The existing OSINET interoperability test network is available so that the interoperability of the ISODE and BSD kernel protocols can be established through tests with several vendors. Testing is crucial because an openly available version of GOSIP protocols that does not interoperate with DEC, IBM, SUN, ICL, HIS, and other major vendors would be embarrassing. To allow testing of the integrated pieces the most desirable approach is to provide access to OSINET at UCB. A second approach is to do the interoperability testing at the site of an existing OSINET member, such as the NBS.

## 2.2. Compliance with POSIX 1003

Berkeley became involved several months ago in the development of the IEEE POSIX P1003.1 system interface standard. Since then, we have been participating in the working groups of P1003.2 (shell and application utility interface), P1003.6 (security), P1003.7 (system administration), and P1003.8 (networking).

The IEEE published the POSIX P1003.1 standard in late 1988. POSIX related changes to the BSD system have included a new terminal driver, support for POSIX sessions and job control, expanded signal functionality, restructured directory access routines, and new set-user and set-group id facilities. We currently have a prototype implementation of the POSIX driver with extensions to provide binary compatibility with applications developed for the old Berkeley terminal driver. We also have a prototype implementation of the 4.2BSD-based POSIX job control facility.

The P1003.2 draft is currently being voted on by the IEEE P1003.2 balloting group. Berkeley is particularly interested in the results of this standard, as it will profoundly influence the user environment. The other groups are in comparatively early phases, with drafts coming to ballot sometime in the 90's. Berkeley will continue to participate in these groups, and move in the near future toward a P1003.1 and P1003.2 compliant system. We have many of the utilities outlined in the current P1003.2 draft already implemented, and have other parties willing to contribute additional implementations.

## 2.3. Improvements to the TCP/IP Networking Protocols

The Internet and the Berkeley collection of local-area networks have both grown at high rates in the last year. The Bay Area Regional Research Network (BARRNet), connecting several UC campuses, Stanford and NASA-Ames has recently become operational, increasing the complexity of the network connectivity. Both Internet and local routing algorithms are showing the strain of continued growth. We have made several changes in the local routing algorithm to keep accommodating the current topology, and are participating in the development of new routing algorithms and standard protocols.

Recent work in collaboration with Van Jacobson of the Lawrence Berkeley Laboratory has led to the design and implementation of several new algorithms for TCP that improve throughput on both local and long-haul networks while reducing unnecessary retransmission. The improvement is especially striking when connections must traverse slow and/or lossy networks. The new algorithms include "slow-start," a technique for opening the TCP flow control window slowly and using the returning stream of acknowledgements as a clock to drive the connection at the highest speed tolerated by the intervening network. A modification of this technique allows the sender to dynamically modify the send window size to adjust to changing network conditions. In addition, the round-trip timer has been modified to estimate the variance in round-trip time, thus allowing earlier retransmission of lost packets with less spurious retransmission due to increasing network delay. Along with a scheme proposed by Phil Karn of Bellcore, these changes reduce unnecessary retransmission over difficult paths such as Satnet by nearly two orders of magnitude while improving throughput dramatically.

The current TCP implementation is now being readied for more widespread distribution via the network and as a standard Berkeley distribution unencumbered by any commercial licensing. We are continuing to refine the TCP and IP implementations using the ARPANET, BARRNet, the NSF network and local campus nets as testbeds. In addition, we are incorporating applicable algorithms from this work into the TP-4 protocol implementation.

## 2.4. Toward a Compatible File System Interface

The most critical shortcoming of the 4.3BSD UNIX system was in the area of distributed file systems. As with networking protocols, there is no single distributed file system that provides sufficient speed and functionality for all problems. It is frequently necessary to support several different remote file system protocols, just as it is necessary to run several different network protocols.

As network or remote file systems have been implemented for UNIX, several stylized interfaces between the file system implementation and the rest of the kernel have been developed. Among these are Sun Microsystems' Virtual File System interface (VFS) using **vnodes** [Sandburg85] [Kleiman86], Digital Equipment's Generic File System (GFS) architecture [Rodriguez86], AT&T's File System Switch (FSS) [Rifkin86], the LOCUS distributed file system [Walker85], and Masscomp's extended file system [Cole85]. Other remote file systems have been implemented in research or university groups for internal use, notably the network file system in the Eighth Edition UNIX system [Weinberger84] and two different file systems used at Carnegie Mellon University [Satyanarayanan85]. Numerous other remote file access methods have been devised for use within individual UNIX processes, many of them by modifications to the C I/O library similar to those in the Newcastle Connection [Brownbridge82].

Each design attempts to isolate file system-dependent details below a generic interface and to provide a framework within which new file systems may be incorporated. However, each of these interfaces is different from and incompatible with the others. Each addresses somewhat different design goals, having been based on a different version of UNIX, having targeted a different set of file systems with varying characteristics, and having selected a different set of file system primitive operations.

Our effort in this area is aimed at providing a common framework to support these different distributed file systems simultaneously rather than to simply implement yet another protocol. This requires a detailed study of the existing protocols, and discussion with their implementors to determine whether they could modify their implementation to fit within our proposed framework. We have studied the various file system interfaces to determine their generality, completeness, robustness, efficiency, and aesthetics and are currently working on a file system interface that we believe includes the best features of each of the existing implementations. This work and the rationale underlying its development have been presented to major software vendors as an early step toward convergence on a standard compatible file system interface. Briefly, the proposal adopts the 4.3BSD calling convention for file name lookup but otherwise is closely related to Sun's VFS and DEC's GFS. [Karels86].

## 2.5. System Security

The recent invasion of the DARPA Internet by a quickly reproducing "worm" highlighted the need for a thorough review of the access safeguards built into the system. Until now, we have taken a passive approach to dealing with weaknesses in the system access mechanisms, rather than actively searching for possible weaknesses. When we are notified of a problem or loophole in a system utility by one of our users, we have a well defined procedure for fixing the problem and expeditiously disseminating the fix to the BSD mailing list. This procedure has proven itself to be effective in solving known problems as they arise (witness its success in handling the recent worm). However, we feel that it would be useful to take a more active role in identifying problems before they are reported (or exploited). We will make a complete audit of the system utilities and network servers to find unintended system access mechanisms.

As a part of the work to make the system more resistant to attack from local users or via the network, it will be necessary to produce additional documentation on the configuration and operation of the system. This documentation will cover such topics as file and directory ownership and access, network and server configuration, and control of privileged operations such as file system backups.

We are investigating the addition of access control lists (ACLs) for filesystem objects. ACLs provide a much finer granularity of control over file access permissions than the current discretionary access control mechanism (mode bits). Furthermore, they are necessary in environments where C2 level security or better, as defined in the DoD TCSEC [DoD83], is required. The POSIX P1003.6 security group has made notable progress in determining how an ACL mechanism should work, and several vendors have implemented ACLs for their commercial systems. Berkeley will investigate the existing implementations and determine how to best integrate ACLs with the existing mechanism.

A major shortcoming of the present system is that authentication over the network is based solely on the privileged port mechanism between trusting hosts and users. Although privileged ports can only be created by processes running as root on a UNIX system, such processes are easy for a workstation user to obtain; they simply reboot their workstation in single user mode. Thus, a better authentication mechanism is needed. At present, we believe that the MIT Kerberos authentication server [Steiner88] provides the best solution to this problem. We propose to investigate Kerberos further as well as other authentication mechanisms and then to integrate the best one into Berkeley UNIX. Part of this integration would be the addition of the authentication mechanism into utilities such as telnet, login, remote shell, etc. We will add support for telnet (eventually replacing rlogin), the X window system, and the mail system within an authentication domain (a Kerberos *realm*). We hope to replace the existing password authentication on each host with the network authentication system.

## 3. References

Brownbridge82

D. R. Brownbridge, L. F. Marshall, B. Randell, "The Newcastle Connection, or UNIXes of the World Unite!," *Software– Practice and Experience*, Vol. 12, pp. 1147-1162, 1982.

Cole85

C. T. Cole, P. B. Flinn, A. B. Atlas, "An Implementation of an Extended File System for UNIX," *Usenix Conference Proceedings*, pp. 131-150, June, 1985.

DoD83

Department of Defense, "Trusted Computer System Evaluation Criteria," *CSC-STD-001-83*, DoD Computer Security Center, August, 1983.

Karels86

M. Karels, M. McKusick, "Towards a Compatible File System Interface," *Proceedings of the European UNIX Users Group Meeting*, Manchester, England, pp. 481-496, September 1986.

Kleiman86

S. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Usenix Conference Proceedings*, pp. 238-247, June, 1986.

Leffler84

S. Leffler, M. K. McKusick, M. Karels, "Measuring and Improving the Performance of 4.2BSD," *Usenix Conference Proceedings*, pp. 237-252, June, 1984.

McKusick84

M. K. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems 2*, 3. pp 181-197, August 1984.

McKusick85

M. K. McKusick, M. Karels, S. Leffler, "Performance Improvements and Functional Enhancements in 4.3BSD," *Usenix Conference Proceedings*, pp. 519-531, June, 1985.

McKusick86

M. K. McKusick, M. Karels, "A New Virtual Memory Implementation for Berkeley UNIX," *Proceedings of the European UNIX Users Group Meeting*, Manchester, England, pp. 451-460, September 1986.

McKusick88

M. K. McKusick, M. Karels, "Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel," *Usenix Conference Proceedings*, pp. 295-303, June, 1988.

Rifkin86

A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, K. Yueh, "RFS Architectural Overview," *Usenix Conference Proceedings*, pp. 248-259, June, 1986.

Rodriguez86

R. Rodriguez, M. Koehler, R. Hyde, "The Generic File System," *Usenix Conference Proceedings*, pp. 260-269, June, 1986.

Sandberg85

Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network File System," *Usenix Conference Proceedings*, pp. 119-130, June, 1985.

Satyanarayanan85
M. Satyanarayanan, *et al.*, "The ITC Distributed File System: Principles and Design," *Proc. 10th Symposium on Operating Systems Principles*, pp. 35-50, ACM, December, 1985.

Steiner88
J. Steiner, C. Newman, J. Schiller, "*Kerberos:* An Authentication Service for Open Network Systems," *Usenix Conference Proceedings*, pp. 191-202, February, 1988.

Walker85
B. J. Walker, and S. H. Kiser, "The LOCUS Distributed File System," *The LOCUS Distributed System Architecture*, G. J. Popek and B. J. Walker, ed., The MIT Press, Cambridge, MA, 1985.

Weinberger84
P. J. Weinberger, "The Version 8 Network File System," *Usenix Conference presentation*, June, 1984.

# Guidelines for an Informatics Architecture

*William de Backer*

Commission of the European Communities
Jean Monnet Building – Room C2-84
L-2920 Luxembourg

*ABSTRACT*

As a user of the information technology (IT) for its own administration the Commission of the European Communities (CEC) has been – as it has to be – a forerunner and an example in applying a procurement policy based on standards. The CEC adopted this policy in 1980.

In order to share its experience with other customers, the IT industry and the standard-making bodies, the CEC is publishing this third edition of its guidelines for the implementation of a vendor-independent architecture.

These guidelines will be revised regularly to respond to changes in the market place. Since the first edition in February 1985, there has been considerable progress in the area of standardisation and a significant shift among major customers towards adopting standardised products.

This edition incorporates further developments to the architecture with a particular emphasis on simplicity, economy, timing and end-user services. Options left open for the future in the last edition have now been settled: inter-institutional cooperation (INSIS and CADDIA), ISDN, LAN, cabling, addressing, security, interactive communication, file transfer, applications architecture. Parts of the previous edition have been rewritten, without however changing the substance.

The next edition of these guidelines will concentrate on applications architecture. The Commission of the European Communities would be grateful to receive information and suggestions.

[This paper is taken from the introduction to a booklet called "Guidelines for an Informatics Architecture" published by the Commission of the European Communities (3rd Edition) ISBN 92-825-7945-X, 1988.]

## 1. Introduction

After decades of being locked into one or a few suppliers for all their computing needs, more and more corporate users in Europe, the United States and Japan are now switching to multi-vendor procurement policies by stepping up their own standardisation effort.

It is in the interest of all users to:

> remain free to choose the best way to integrate new technology independently of the policy of individual manufacturers.

In addition, the European institutions have to cope with the complexity caused by different languages and partners in remote geographical locations. This turns an interest into a necessity, and dictates a high degree of flexibility in implementing these technologies.

More and more organisations understand why a vendor-independent strategy based on standards is feasible, economic and necessary.

- It is feasible through an evolutionary path: proprietary interfaces can be phased out and replaced with standard ones.
- It is economic because expensive interface adaptions can be eliminated and costly conversions avoided. Once users are no longer locked into vendors, competition in an open IT market brings prices down.

- It is necessary if organisations are to communicate and interwork effectively.

### Standards contribute to market growth

The limiting factor in market growth is not technological progress: customers are often offered more than they can use. Instead, it is the ability of users with different hardware and software to interwork – this ability is only offered by standardisation.

This demonstrates that the market for standardised products is a growth market: standardisation has a multiplier effect. The more people can communicate and services can interwork, the grater the volume of information they exchange – and the large the IT market becomes.

### Industry and standard making bodies must adapt to customer priorities

Industry and the standard-making bodies should listen to the requirements of customers for the standards and understand the difficulties they have in implementing them; the risk of not doing so is to offer standards which cannot be sold, while other standards may be badly needed. In the standard-making process, this means missing windows of opportunity when these present themselves.

For these reasons, customers must coordinate their activities in formulating their standards requirements and make their voices heard so that industry will fulfill them.

The CEC sees with satisfaction a growing number of customer associations taking the lead in standards implementation. The CEC as a customer wishes to encourage their efforts, contribute with professional information and experience and support proposals of common interest.

European users should lead the world in demanding standard products. If the IT industry in Europe supplies the solutions to such a receptive market, it can become competitive on a world wide scale.

### An architecture is more than standards

Defining guidelines for an evolutionary architecture for the next decade is not an easy task. Most architectures have been designed by the computer manufacturers to match their present and future product ranges. Such architectures are, generally speaking, mutually incompatible even where standards are used. A vendor-independent architecture can only be developed by the customer – but no single customer has the power to impose a given architectural design on industry. Consequently, such an architecture must emerge from the ongoing process of supply and demand.

This is the spirit in which these architectural guidelines have been prepared. They reflect neither specific customer needs not particular design concepts; moreover, they take into account the availability of products on the market. This is an additional reason why the guidelines themselves are subject to constant review and correction.

## 2. Architecture

### 2.1. Domain Structuring

User populations of autonomous organisations, such as private companies, government administrations and European institutions, have independent informatics management structures, here called Private Domains (see figure 1). Private domains communicate via Public Communication Networks (PCN) and use services provided by Public Domains, which are managed by public telecommunication administrations (PTT).

The distinction between public and private domains implies a distinction between inter- and intra-domain architectures. The first has to be determined by common agreement between all the domains concerned, whereas the management of each domain is autonomous in determining its intra-domain architecture. The definition of the intra-domain architecture, however, is significant if end-user interworking across domains is to be achieved.

A private domain is composed of Local Domains interconnected with a Computing Center and Telecommunications Center by a Private Domain Communication Network (DCN).

A Local Domain is characterised by a close working relationship between its users and a common requirement for specific sets of services, thus justifying the cost of providing local support and management.

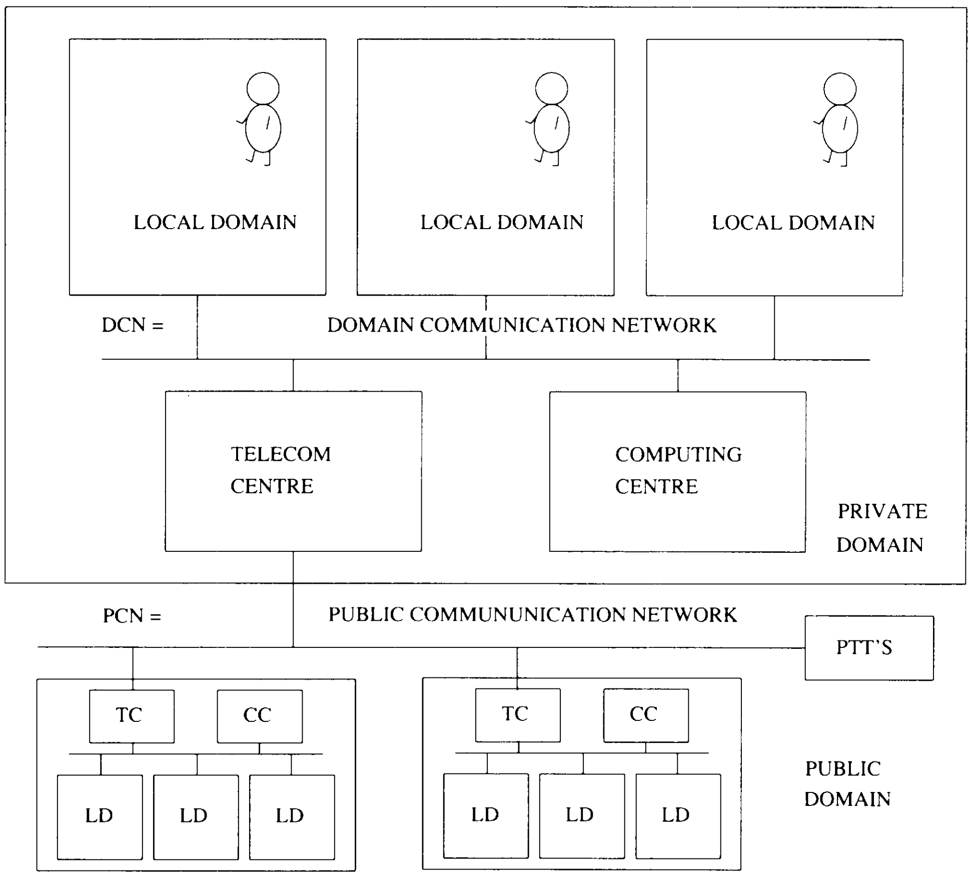


**Figure 1**

A local domain is equipped with a Local System (LS), which includes local computers, personal computers, terminals, printers, a local communication network, management and user support. A local system (or departmental system) is thus an autonomous unit dedicated to the local domain.

A Common System (CS) of interconnected computers provides remote services to one or more local domains, the entire domain or other private domains. Common systems are located only in computing centers and telecommunication centers.

The structuring of local domains does not apply to telephone networks. For further discussion of this matter see §3.2 in [1].

## Organising local domains

A local domain will generally coincide with an organisational unit such as a department at a single site. "Site" is not necessarily the same as "building". There may be more than one site within a building, or there may exist links between buildings to enable them to be treated as a unit.

A local domain should serve as large a user community as possible. If, however, a user community is divided geographically in such a way that it cannot be supported economically by a single local domain, it should be served by different local domains. Similarly, if a large user community consists of groups carrying out activities with little or no interaction, it should be served by different local domains.

The domain architecture as described above is intended for large and geographically spread organisations. Small organisations located in one building are advised to adopt an architecture similar to that of a single local domain. A gateway to the public networks could then contain a mini telecommunications center.

Other deviations from the general model may occur when members of a local user community are not in the same office area but a separated by relatively large distances. If this is an exception, a pragmatic ad-hoc solution must be found without changing the architecture guidelines. There are cases, however, where user communities are spread over different countries (e.g. internation committees), or even composed of mobile members (e.g. members of the European Parliament). Two solutions are then recommended:

- If administrative and/or secretarial support can be provided at some location for the community, this can be equipped with a local system providing a service access point. The distant workstations, preferably personal computers, are then connected by the public and private communication networks to the local system, from where all the services for the local community can be accessed.
- If such a service entry point cannot be provided, each user may then be equipped with a "single workstation local system", again based on a personal computer, or even, in some cases, simply with a terminal linked to services provided by the national PTTs (e.g. Videotex). In this solution, isolated autonomous users are considered as single-member domains.

## 2.2. Distributed Processing

One of the fundamental problems of an evolutionary architecture is to decide where to locate software and data on the computers in the network (see figure 2). In order to describe the problem more accurately, the following terminology is employed.

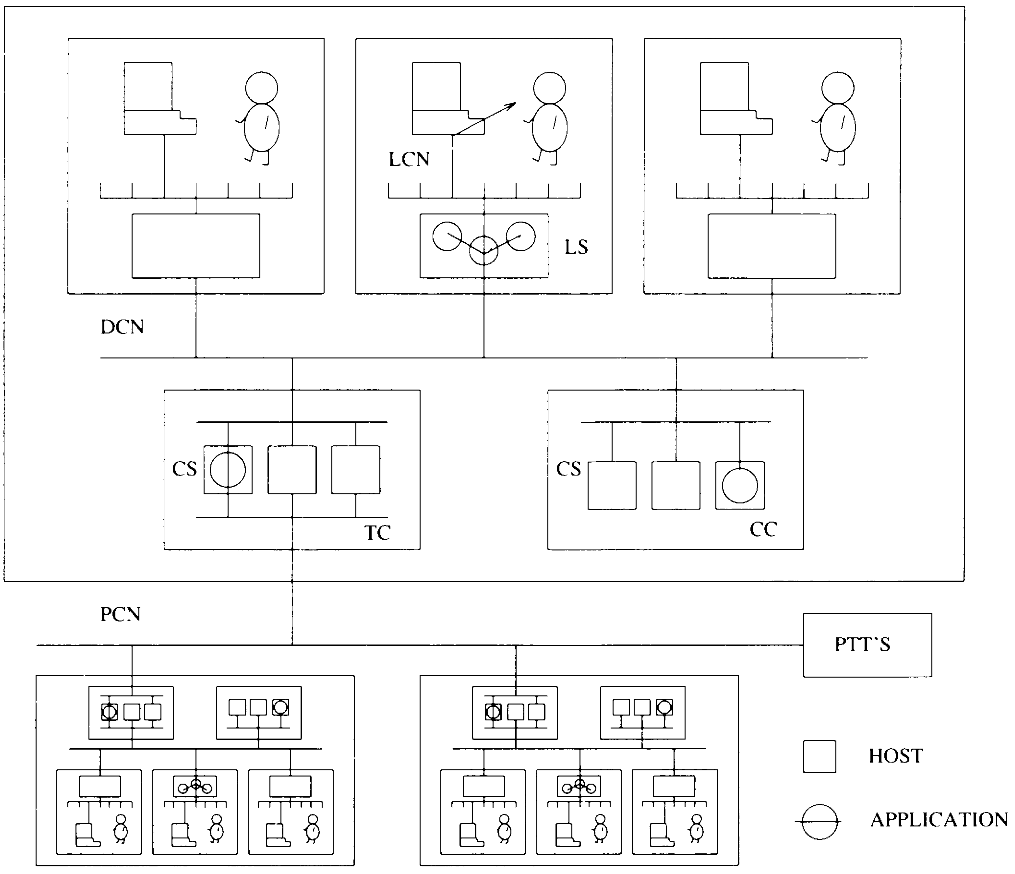


**Figure 2**

The *workstation* is either a terminal or a personal computer (including printer), although a future integration of data, text, voice and image will lead to a single multi-function workstation providing access to all user services.

A *user service* is what the user gets at the workstation when he has logged into an application. Examples of user services are: access to a database, word processing, document management, electronic mail, etc.

An *application* is a set of co-operating pieces of application software and data processed on the same host or on different hosts in the network.

A *host* is a computer with its operating system and the communication software necessary for its integration into the network. Interworking hosts not only process applications but also provide communication services such as interactive communication, file transfer and messages handling.

A *personal computer* is, in line with the definitions above, both a workstation and a personal host.

The *Open Systems Interconnection* (OSI) reference model is fundamental for the architecture of hosts and networks but does not solve all the problems.

For applications to be processed by a given host, they must technically fit together. The more hosts that offer the same interface to applications the easier it is to relocate application software on different hosts. Transportability of software is very important for an evolutionary architecture.

In order to define a distributed architecture one must define

- how user services are provided by co-operating software and data on different hosts.
- how hosts are distributed in the network.

The decision rules are organisational (see §2.1 on domain structuring), economic, functional and technical. All of them change in the course of time. As a result of decreasing hardware costs, there is a shift from central to local processing, thus avoiding the chronic congestion of central hosts and networks.

In general, the distribution of application software will follow the pattern in which data are distributed.

The natural location of application software and data is on a personal host (personal computer) when dedicated to a single user, on a local host when shared by members of a local user community, or on a common hosts when shared by several local user communities, the whole private domain, and possible user communities outside the domain. However, for technical and economic reasons, they are often installed further upstream from the user. In the past, all applications were processed by the computing center mainframes, and for large local databases, heavy batch processing and number crunching this is still economic today. Another example is where data for a single user are stored on a local host in order to implement data administration services.

It is technically possible, though undesirable, to locate common data downstream from their normal position and process them on local hosts. For reasons of responsibility, security and resource management, common user services must be provided only by the computing centers and the telecommunication centers.

## 2.3. Applications Architecture

Workstations, hosts and networks are merely the foundation that provides stability and supports the applications. Computers and communications have become a commodity. The really difficult architectural problems start when the user organisation invests its own added value.

Most applications are based on software products requiring tailor-made development. This results in as many different architectures and user interfaces as there are applications. With their growing number, this becomes an unbearable situation both for the user and those who have to maintain the applications.

*Integration* is the solution, but what is the framework to support this integration? The OSI reference model was not devised for this purpose. The so-called "client-service" model is a much more promising road to follow. Much will depend on whether this model will be followed by the designers of software products.

The problem is to assemble user services with commercially available software components such as those listed in table 1.

Many software suppliers have already succeeded in integrating several components in the same product, although too often for a single host instead of a distributed host environment. The higher the level of integration, the more the internal architecture of the product becomes part of the overall architecture adopted by the user organisation. This is progress, but in the absence of standards it can also introduce a new type of captivity. For all these reasons, software products should be selected only if they

- have an open architecture (easy to integrate other software components);
- are transportable or at least are available on as many computers as possible;
- provide a similar human interface (in the absence of standards, the best approach is to follow the most successful trends);
- are designed for distributed processing, preferable following the client-service model.

A particularly difficult problem with architectural implications is multilingualism. International language coding tables must be outside the operating systems or application software and must be processed as data.

| | |
|---|---|
| Word Processing | Formatted Database system |
| Spreadsheet | Data Dictionary |
| Data Entry | Thesaurus |
| Report Writer | Electronic Mail |
| Document Composition (Desktop Publishing) | Integration software |
| Business Graphics | Statistical software |
| Geographical Information software | Time management |
| Filing | Project management |
| Document Administration | Compilers/Interpreters |
| Documentary Database | etc. |

**Table 1**: *Commercially available software components*

User services are not independent from one another and also call on lower-level supportive services. The structuring of application software and data covers more than distributed processing and also includes switching between user services, the content and location of databases and the information flow between them. The management of the private domain has to adopt its own architecture design in these cases. Chapter 4 in [1] illustrates a possible approach.

## 2.4. Standards

Commercial products are the building blocks of the architecture. They can only fit together when there is a 100% match between

a) all the functions that have to interwork

b) the standards governing access to these functions

The only solution to this problem is to employ *Functional Standards* or *profiles* that match the product interfaces. In the field of machine interworking, CEN/CENELC/CEPT, with the support of industry (SPAG), has take the initiative in defining such profiles. The resulting EVN-EN standards must be fully complied with. These profiles are a direct application of ISO/CCITT basic standards. When ISO profiles are approved, they will supersede the EN profiles.

At present, functional standards have only been developed for the OSI model but they should be extended to all types of interworking between products (e.g. different software products on the same machine). We shall from now on use the term "standardised products" when the product interfaces conform with such standard profiles.

A customer pursuing a multi-vendor procurement policy will buy standardised products. Council Decision 87/95/EEC of December 1986 makes this an obligation for public procurement in the European Community. However, when such products are not available or have only a partial standards coverage, the customer will take what, in his judgement, is the best alternative available on the market. In association with other customers, he can handle deviations from standards through controlled procurement, put pressure on the standard making bodies and industry to get their priorities right and, if possible, introduce new standards proposals.

The introduction of standardised products is urgent, not only for economic reasons, but also because it will become more difficult the longer it is postponed. Computer applications (databases, wordprocessing, electronic mail, document administration, etc.) are growing at a rate of 25% a year, which means that if we wait 10 years, the switch-over to standards will be much more difficult, if not impossible.

For this reason, one might expect a fast growing market for standardised products, but unfortunately this will not happen tomorrow. Today, many standards are still missing, especially in the upper layers of the OSI model. Those available now are not always option-free, or do not completely cover the functionality of product interfaces (standard profiles). When these problems are solved, paper standards must be turned into *de facto* standards by industry developing and selling standardised products. But the customer, in line with his multi-vendor procurement policy, will require standardised products supplied with a conformance certificate from one of the official Conformance Test Centers now being set up in Europe. A conformance certificate will not necessarily guarantee interworking, however.

All this will take considerable time. In addition, we should not forget that the first deliveries of a new standardised product will not fit into a non-standard environment, that the customer cannot phase out equipment before the end of its economic lifetime (3 to 5 years), that industry must continue to sell the money-spinners in its non-standard product range and that the end-user wants stability and continuity in his applications.

In short, an important market of fully standardised products will not emerge before ten years from now, and this is much too late for the reasons explained earlier. Consequently, we must apply an implementation strategy long before the availability of fully standardised products. This is both possible and urgent. If we do not do this, the window of opportunity to a standardised market may be closed before we get there.

It is not possible to understand the priorities for standards and the market for standardised products without considering the customer trying to get away from a captive supplier-dependent architecture. He can already, where standardised products are not available yet, implement interim solutions that are as vendor-independent as possible and integrate them into an evolutionary and carefully planned scenario in successive steps of about 5 years, aimed at closing the gap between proprietary architectures and a standard interface architecture.

The CEC has been successfully following such a scenario since 1980 and is now benefiting from the resulting cost improvements.

In the light of this experience, the following four subjects are of prime importance for standards work.

### Open System Interconnection (OSI)

The OSI model and related standards must be applied throughout the architecture and not just to bridges between proprietary architectures. The major customers in the world, not only in Europe but also in the USA and Japan, are now putting their weight behind the implementation of OSI, to the extent of ignoring other standards.

The top priority for OSI should be remote access to databases, for which no satisfactory standards are available. This is one of the important reasons why the information market has been so slow to take off.

### Applications Architecture

The present lifetime of computers is 3 to 5 years, whereas applications survive for at least 15 years. At the same time, applications software is migrating from central to local and personal hosts for economic reasons. In order to protect investment in applications in an evolutionary environment, it is necessary to have a stable platform to build upon. In 1984, the CEC decided to adopt UNIX (or equivalent) for local systems as the best solution for this purpose.

Since then, a group of computer manufacturers with foresight (X/Open) has decided to cooperate in defining a standard interface between UNIX and applications software. This common applications environment will become more important than UNIX, as it will open up the software market.

Applications software will dominate the future but the high development costs will not translate into profits unless the products can be sold for a world-wide computer base. This is why X/Open is so important, especially for Europe. Independent software vendors with good ideas will at last find an open market.

The CEC, as a customer, fully supports the adoption of the X/Open Portability Guide as an international standard, hopefully harominised with the POSIX specifications in the USA.

The more the evolution of applications can be uncoupled from the evolution of hosts and networks, the easier it will be to speed up the integration process explained in §2.3. But this makes all the more urgent the standards work on applications architecture, including the further development of the ECMA and ISO model for Distributed Office Applications (client-service model).

### Human Interworking

In the decade to come, users will spend more and more time at their workstations with access to a growing number of user services, they will no longer accept

- software products employing a different terminology for the same functions
- different languages and screen styles for each different application
- different keys on the keyboard for the same function
- different office procedures for work of a kind that is the same all over the world

On the other hand, once users have been trained to master the complexity of a given user interface, they will resist further changes. This shows the urgency of international standards for human interworking, but today their preparation has not advanced beyond a preliminary analysis of user requirements.

## Security

Security standards are necessary not only for their own sake but also because of their role in open systems interconnection and consequently the multi-vendor policy.

The implementation of security measures affects almost all components of the architecture – both hardware and software. In the absence of international standards, the IT industry follows the solutions developed by leading customers, in most cases national defence administrations. Even if these technical solutions are allowed in products for private or public market, this remains a sensitive issue for both customers and suppliers in an open world market.

## References

[1] William de Backer, *Guidelines for an Informatics Architecture*, 3rd Edition, Commission of the European Communities, ISBN 92-825-7945-X, 1988.

# Creation of an Open Systems Market: The role of X/Open in the practical establishment of open system standards

*John Totman*

Director of European Programmes,
X/Open Company Limited.

## 1. Introduction

Information is a strategic asset to business. It is vital to the business as the human resource and raw material. The capability to organise manage and utilise information effectively within a business has become a major factor in the success of the business.

Historically the key to that capability has been the human and the computer.

The personalisation of computing in the early 80's brought about an explosion in the use of computing. People found that computing in essence was simple, it was not the domain of the programmer and the technologist, it was possible for you to collect information pertinent to your own job, analyse it, process it in your own way and extend your own knowledge and capability. The personal computer became a tool of business and of life. It improved effectiveness of business and the quality of life. It showed the way.

And like all major breakthroughs in knowledge management the opportunity it brought created some upheavals. For example:

- The traditional corporate computer staff felt that they were losing control.
- The user became frustrated that his personal knowledge system would not easily connect with other computers.
- Many traditional large mainframe suppliers were following proprietary strategies that failed to meet the real user needs.

The reality was that the person, the user of computers, was taking charge. The user was signaling to the industry that what was needed was a simple, reliable, easy to use computer, with a wide range of software available from shops. The user demand led to a new billion dollar market.

The net effect was that the computer industry got the message

> *''Study what the people want and innovate to deliver those needs,*
> *stop pushing technology down the throat of the user.*
> *Package the technology to suit the buyer.''*

This was a major change for the computer industry, an industry that has been historically technology led rather than market led.

From this point on the computer industry had to start focusing very sharply on what the user needed – it of course meant marketing.

## 2. The Open Systems Market

To signal this change the computer industry coined the expression ''OPEN SYSTEMS'' to describe this new global market, and the events of the last year mark a major turning point for the whole of the computer industry towards open systems based on industry standards.

The computer industry's ''Open Systems'' message is really another way of saying ''we care about you''.

Because ''Open Systems'' is all about bringing every user a choice.

Now to maximise the benefits of the personal computer on a wider scale you have to be able to do two things:

- Loan your own personally developed system to someone else.

- Share your personally developed knowledge system with other people.

These benefits are known in computer jargon as:

- Portability – the ability to transfer your personal application easily to another computer of your choice, inevitably a different manufacture to the one you currently use.
- Connectivity – the ability to connect your personally useful knowledge system to another probably running on a different manufactures system.

## 3. X/Open's Role

### 3.1. The Benefits of Industry Standards

The role of the X/Open Company is to bring these benefits to you today, by pragmatically overcoming the obstacles that inhibit you from easily loaning or borrowing applications or just plugging into other peoples systems.

X/Open's role is to provide you with an industry standard that brings:

- vendor independence
- freedom of choice
- protection of investment
- continuity of supply.

But first, let's look at the business and government changes that are the driving need.

We at X/Open believe that we are leading a strategic response to meet a true market need. For example, some months ago, the Financial Times, in conjunction with Price Waterhouse, fielded its 1988/89 IT Review, which was a study of attitudes to IT among chief executives in UK companies as, I quote, *''the emphasis on computers in business shifts from their traditional use in accounting and payroll to employing them as strategic weapons in, for example, management information and marketing''*.

There were two principal findings that are of interest to us.

First, there has been a dramatic increase in the number of companies attempting to measure the effectiveness of their IT operations.

Second, a majority of chief executives believe they now have the necessary expertise at board level to guide their companies in the strategic use of information management.

### 3.2. The Changing Use of Information Technology

The survey spotlighted a changing balance between the administrative use of computers and their use for strategic purposes; five years ago this was about 20% strategic. Today, the balance has moved to 50-60% strategic and the expectation is that in five years time it will have moved to 80% strategic.

The results of this survey confirm clearly that in the trend that information technology is now being used to make money for organisations rather than just to count it as it did in the past.

Now open systems are the key to unlocking an increased return on investment. But, are companies looking far enough when they tackle this issue.

### 3.3. The X/Open IT Investment Model

Let's take a look at the X/Open perspective in the form of an investment model.

What this says is that for every dollar spent on hardware a buyer spends ten dollars on software – no surprises.

But the same business spends $100 for every hardware dollar on training both the technologist and the user and $1000 for every hardware dollar in integrating the information produced into something meaningful for the business.

Now the costs of training and information integration are often not held in the MIS or DP budget, they are therefore hidden. These costs reduce your return on investment dramatically.

X/Open helps you attack these costs and improve your return on investment by reducing integration costs through portability and interconnectivity, while not constraining you to a single source of supply.

You will be able to multi source and competitive tendering will increase your return on investment.

You will be able to realise the asset of collective information, the bringing together of data held on a variety of systems on every aspect of an organisation's performance, status and position.

That information is power, and in Government and Commerce, those who can capitalise on the information asset effectively hold the key to true competitive advantage.

What's holding things up you may ask?

## 3.4. The Need for Standards Integration

This is how we might depict the current state of computing from a management perspective. In most organisations, the history of information technology purchase is one of disparate buying polices so that many different types of computer systems, working with a variety of different operating systems and software environment, meet the company's computing requirements.

This is so even though many of them do similar tasks and process similar data.

This means that the complete integration of non-compatible systems across large organisations can be extremely difficult to achieve, if not impossible.

So what we have are ''islands'' of computing function, isolated one from another by islands of technology. Each island is adequately fulfilling the information needs for that particular function, but there is no possibility of collectively gathering, exchanging and analysing data.

We need standards to integrate those islands of computing, and X/Open's work in the marketplace is to bring a common ground to permit the integration of standards and hence the portability of applications and interconnection in an open systems framework.

The demand from users to achieve this exists without doubt. The entire X/Open initiative is based upon the growing market demand for open systems.

At the outset, the rationale behind X/Open was the desire to give users more value from their computing investment through being able to take a practical step in open systems.

## 3.5. X/Open Common Applications Environment

The X/Open solution in providing a practical route to open systems has been the development and specification of the ''Common Applications Environment''. It involves the introduction of a platform of interfaces based on generally used standards and independent of any one vendor. To understand this is to appreciate a fundamental of X/Open's approach to open systems: we haven't started with a blank sheet of paper and defined standards from scratch.

X/Open adopts and adapts existing international and de-facto standards, choosing in that order of priority. It does not create standards. Where existing standards are not coherent, or gaps exist among them, X/Open's role is to co-operate technically with relevant bodies to resolve these issues in the best way possible, with end user needs in mind.

The Common Applications Environment is an integrated whole that encompasses all the aspects involved in providing a cohesive and consistently implemented environment supporting applications portability at source code level and delivering the full benefits of open systems.

It is a stable but evolving environment for software. And it is a public, open environment for all to use: information on the CAE is contained in a book available to all called the Portability Guide. This contains the technical data defining the X/Open CAE so that software and hardware can be developed to meet the criteria for X/Open conformance.

It provides the user with a short cut, a design specification for portable applications and the buyer with a means of procuring open systems.

You have heard why X/Open was founded. But how is it organised and who is investing time and money to make it work?

## 4. X/Open Sponsors and Supporters

### 4.1. X/Open Organisation

X/Open is an independent company whose current sponsors are fifteen of the world's most important data processing companies. Each of these corporations has made a public commitment to bringing open systems to the marketplace by joining X/Open. Their combined strength on a world scale is impressive.

Each of the companies has committed financial and human resources to X/Open on an equal basis. The strategy and progress of X/Open is currently governed by a Board of Directors, one from each of the sponsoring companies.

The Board currently comprising of corporate members is supported by marketing and technical groups which work with X/Open in each of these areas, tapping the wealth of expertise resident in some of the most advanced data processing organisations in the world.

Ensuring that X/Open moves forward in its aims has meant setting a number of objectives.

First and foremost, X/Open is intent on educating and accelerating market awareness of the value of open systems. This means establishing a common understand amongst users about the opportunities and benefits offered by open systems and the X/Open approach to implementing them.

### 4.2. User and Software Industry Councils

To help achieve this X/Open has established a user council, whose membership comprises of leading edge Government departments and commercial corporations who are committed to strategic benefits of open systems. This group revies our strategies and X/Open revises its plans to meet their needs. Our mutual goal is to use the X/Open specification as a standard for specifying open systems.

In the software world the X/Open market impact is to generate a large single market of many types of hardware to which software companies can market applications and achieve a far better return on investment.

X/Open has therefore forged close links with the software industry. we have established a software industry council comprising of leading edge software companies, who are committed to the delivery of applications that conform to the X/Open open systems standards. X/Open reviews the forward strategies and revises them to reflect the needs of the software industry as channeled through this group.

### 4.3. Software Partners

The software industry is so critical to the success of open systems that X/Open has additionally created an international programme called the Software Partnership Programme. This Programme meets as a seminar forum, in national groups. In 1988 we held meetings in USA – Boston, Washington and San Jose; Europe – Hanover, Milan, Brussels, Utrecht and London. There are already over 200 software partners in the programme, all committed to building products that conforms to the X/Open open systems standard.

Here in Germany, out of your top ten software companies, seven of them are X/Open Software Partners. The growing portfolio of conformant application products is being published in a software catalogue for you to pick and choose from when building or specifying your open systems needs.

## 5. Current Status and Future Directions

### 5.1. Verification and Branding

X/Open recognised that you need hardware that conforms to the X/Open standard to support the large portfolio of software. The Corporate members have all announced their delivery programme for conformant hardware. So you can specify X/Open as a requirement now and anticipate a multi-vendor response.

To ensure that the user knows that the software and the hardware conforms to the X/Open industry standard for open systems, to give portability and connectivity, X/Open has a simple solution; Look for the X/Open brand is the answer.

X/Open licenses software and hardware vendors to use the X/Open brand on their product provided the product has passed strict conformance tests. The test programme is currently available to suppliers for their own use and through an increasing public network of test centres. The first established with the software house, Unisoft, with test centre locations in London, San Francisco and Tokyo. More test centres with expertise to run the programmes and advise the supplier will be established during 1989.

The brand mark is therefore a good housekeeping seal – where you see this mark, you will obtain the immediate and practical benefits of open systems through the X/Open industry standard.

## 5.2. Portability Guide Editing 3 (XPG3)

To create the Common Applications Environment, it is necessary to define a series of interfaces between the application and the components that make up the world it depends upon, its environment.

The Portability Guide, mentioned earlier, contains technical information on each of these components. The five volumes of issue 2, the currently available guide, define the majority of the interfaces needed by applications developers. In addition to the basic operating system services, definitions are provided for the main programming languages, data management and networking.

I'll pick out some of the major points of issue 3 which has just been published.

Firstly, the structure of issue 3 is organised into sections concerned with

Portability
Connectivity
Environmental functions

If wee look at the operating system interfaces to applications, initially these were based on AT&T's System V Interface Definition, now the X/Open definitions are fully aligned with the POSIX standard. We were conscious early on of the possibility that X/Open, POSIX and AT&T would diverge on standards, but in fact the three definitions have converged.

AT&T is a full member of X/Open, accepting the commitment to conform to the X/Open Common Applications Environment and X/Open gave its early support to the IEEE POSIX initiative, committing to converge the Portability Guide and the POSIX definition.

Other sources of technology, like OSF, are committed to conform with the X/Open specification. This illustrates the dual role of X/Open in further converging solutions on one procurement specification, the X/Open Common Applications Environment and offering the user multiple choice.

Feedback at recent user and ISV council meetings put the user interface as one of their top priorities and we have shifted our own technical work effort to reflect this. A low-level programming interface definition to a networked windowing systems based on X-Windows is included in Portability Guide 3.

X/Open has two main objectives in the area of networking. The first is to provide consistent portable applications interfaces to networking services; the second is to facilitate the transition from existing network architecture towards open systems standards.

This is an area where there has been little coherence over the last few years, with competition between open systems and existing proprietary protocols. X/Open is concentrating on the definition of protocol-independent application interfaces to services and "trial use" definitions have been defined.

In networking and communications, Portability Guide 3 contains the first communications interface for X/Open, a transport interface for systems programmers. High-level applications communication interfaces are being defined as part of the overall transaction processing program.

Security is a concern of many organisations. Certain users, primarily in government, have expressed a demand for secure open systems, and we are investigating which security aspects inherent in UNIX-based systems that we can bring into CAE. We are aiming to bring the security level of the CAE to the C2 level specified by the American Department of Defense, the highest level that can be reached without a major rewrite of the operating system.

However, the security needs of most commercial users can be satisfied with features which are already available. X/Open is now publishing a handbook explaining how to exploit these features.

## 5.3. Global Standards Development

Looking forwards during 1989, X/Open will strengthen its cooperative work with other leading standards bodies in the US and Europe.

Already our successful work with IEEE and National Institute of Science & Technology (NIST, formerly NBS), has seen the convergence of POSIX 1003.1 and X/Open. Further work is already in hand to bring other areas of POSIX and X/Open together in the coming months.

Here in Europe, X/Open is working with the consortium led by the National Computing Centre (NCC) to establish harmonised POSIX testing centres throughout Europe within the CEC CTS programme.

In addition, CEN is now reviewing the X/Open Portability Guide as a proposed draft European Standard (ENV).

## 6. Conclusion

Today, X/Open provides you the user with:

- the means to increase your return on investment in information management
- the means to take charge and direct the computer industry to meet your needs more closely

To achieve this you can:

- direct your own MIS staff to build portable connectable applications in accordance with the X/Open specification in the portability guide.
- include the X/Open specification in your next procurement
- remember to specify X/Open conformant or branded products

X/Open provides you the software vendor with increased opportunity and sales, through:

- a wide scale international economy based on the X/Open brand

To achieve this you need to join the growing population of software vendors producing X/Open conformant products.

X/Open provides you the hardware vendor with the opportunity to respond to the demand for open systems, a demand sized at $40 billion by 1991.

- an opportunity to respond to the open systems vendors specifying the X/Open Common Application Environment
- to achieve this you need to join the growing population of hardware vendors producing X/Open conformant products.

X/Open is here to help you increase your return on investment, let's do it together now.

# The Open Software Foundation's Open Process and the End User

*Henning Oldenburg*

Open Software Foundation

*ABSTRACT*

The Open Software Foundation has passed a major test of credibility with its recent selection of user interface technologies, announced to OSF members December 30, 1988, and presented to industry, press and consultants January 11, 1989 in the US and January 13 in Frankfurt, West Germany.

The importance of the announcement of a User Environment Component, OSF/Motif, goes beyond the adoption of technologies. Of equal significance is the way in which OSF selected the technology, and the implications of OSF's "open process" for the industry and its end users.

## 1. The Open Process – How it Works

OSF's open process is an innovative development model which solicits technology requirements, direction and comment from OSF's members, invites the industry to submit relevant technology, and then utilizes member comment, industry consultants and Foundation staff to reach technology decisions.

In a typical commercial development, none of the above steps is taken. Instead, a corporation's research and marketing teams decide which technologies fit best for their markets, develop these, and wait for user acceptance. To the degree that the research group listens to the company's, user groups have input; but the product development remains, by and large, company-driven, not user requirement-driven. If the development cycle is viewed as a classic process control flow loop, it can be seen that changes in technology direction first lag demand, then exceed requirements, before balance is achieved between what the user wants and what the systems vendor delivers.

OSF's open process can be viewed as a mechanism to balance the push and pull effects of the development cycle. By having user requirements drive the technology selection and development cycle, the open process assures broad acceptance of technology, maximum usability and cost-effectiveness. The end user knows much earlier what technology will be available to him; the independent software vendor can develop applications even as the product is being developed; and the time to market for the final product is shortened. Involvement with OSF as a member and participation in the open process, provides these advantages, and more.

The open process does not take place in a vacuum. OSF works closely with standards bodies to insure that Foundation software offering companies complies with industry standards. Additionally, OSF is working to create standards. User interface technology was chosen for the development effort because it is a controversial area in which standardisation has been elusive. By choosing Presentation Manager-compatible technology, OSF can help the industry move toward standardisation of user interface appearance and behaviour. The technologies selected to form the core of the user environment component are based on X Windows, ensuring compatibility with non-UNIX system environments.

An additional benefit of the open process is the speed at which technology is specified, selected and developed. OSF was founded in May, 1988. Within two months, the Foundation had selected the area of user environment component (UEC) technology for development, based on input from its membership. OSF issued "Request For Technology" (RFT) to the industry in July. By the September deadline, thirty-one of the world's outstanding interface technologies had been submitted to the RFT process. Twenty-three met all submission criteria, and were presented to OSF's members. The members then reviewed the submissions and reconvened a month later to make their recommendations to OSF staff. With the additional help of a team of five influential industry consultants, OSF staff worked to match the members requirements with the technologies available, at the same time working to ensure the most advantageous licensing and business terms. The result, OSF/Motif, combines member-specified technology, innovative licensing, and a user environment which will bring the ease of use common to the PC world into the UNIX software environment.

## 2. OSF/Motif – What is it?

The technology selection process behind OSF/Motif has importance for end worldwide because of the decision process. Of equal importance to end users is the technology specified and now under development.

OSF/Motif, the core User Environment Component, is the first offering of the open process. OSF/Motif is an optimal composite of the technology submitted to OSF by Digital Equipment Corporation and the joint submission of Hewlett-Packard and Microsoft. The most important aspect of the software, for end users, is its Presentation Manager-compatible 3d look.

At its most basic level, OSF/Motif will provide users of UNIX systems, from desktop workstations to mainframes, with a single, homogeneous, PC-style look and feel. The proliferation of PC's in the workplace validates this approach to a user environment. The selection of Presentation Manager-compatible behaviour and 3d look will provide users with a virtually seamless transition among PC's, workstations and other classes of machines.

The underlying technology of OSF/Motif is based on Digital's applications programming interface (API), presentation description and toolkit. The OSF/Motif API will be enriched with features from Hewlett-Packard's API and an extended version of HP's Window Manager. The core offering will include a full documentation set with a style guide based on Presentation Manager behaviour.

In order to facilitate the selection of UEC technology, OSF developed a user environment technology framework on which to model the components (and their roles) of the UEC offering. The framework is based on a reference model developed by the National Institute of Standards and Technology (NIST), which was originally derived from a framework developed by X/Open.

OSF/Motif's core offering is based on technologies that are highly portable. The core consists of a style guide, a window manager, a toolkit, a presentation description language, and associated documentation. Each technology offered in the core is examined in the following pages.

## 3. Style Guide

A style guide serves many functions for the user. It is a specification of the performance characteristics of the window manager and toolkit behaviour. The style guide specifies the appearance and behaviour of these technologies from the user's perspective. The style guide also serves as a guide to usage of the toolkit's objects, or widgets.

OSF's style guide will be a merged document, reflecting the combination of technologies from Hewlett-Packard (the window manager), additional features from Digital's window manager, and a toolkit derived from merged HP-DEC technologies. OSF/Motif's style guide will describe the PM-compatible behaviour inherent in the OSF window manager and toolkit, as well as the extensions to its behaviour and appearance.

## 4. Window Manager

OSF selected HP's window manager and extended it with features from DEC's window manager, in particular DEC's Icon Box.

The window manager supports Presentation Manager behaviour and layout, and the provides the 3d, or bevelled, appearance, which has been chosen to be OSF's reference appearance. The appearance is supported on both monochrome and colour workstations. The window manager is ICCCM-compliant. It can be customised in both appearance and behaviour. In addition, the colours and styles associated with the 3d appearance can be varied widely to adapt to different screen sizes, resolutions and colour capabilities.

The Icon Box holds icons for each of the windows managed by the window manager, collecting them in a single location and enabling them to be moved as a group.

## 5. Toolkit

OSF/Motif's toolkit is based on a combination of the HP/Microsoft and DEC RFT submissions. Both toolkits are based on the X11 Release 3 version of the Xt Intrinsics, which is an object oriented foundation for building graphical toolkits available as part of the X Window System. Therefore, although OSF/Motif is not written in an object-oriented language, it is an object-oriented software offering, supporting the notion of inheritance found in object-oriented languages.

The OSF/Motif toolkit supports more that 30 widgets, from the most basic scrollbars to specialised dialogue widgets. The application program interface (API) provided by the toolkit is based on the one supplied by the Xt Intrinsics, further extended by the DEC toolkit. OSF/Motif's toolkit will be extended to support HP functionality.

The behaviour of the toolkit conforms to Microsoft's Presentation Manager, with additional extensions to take advantage of the power of networked, X-based workstations. This compatability means that users should have an easy learning transition between PC and workstation environments, and will additionally simplify the documentation needed to utilize both types of systems.

## 6. Presentation Description Language

OSF/Motif's presentation description language will be based on technology, known as UIL, submitted by DEC. A presentation description language describes the presentation aspects of a user interface. The presentation description language can be used by interface designers who need not have extensive programming experience.

Interfaces described in OSF/Motif's presentation description language are compiled to a binary format. This binary format is not linked to an application program, but is read by the program at runtime. For the application developer, this means there is a fast turnaround in tuning an interface design, since changing the presentation aspects of an interface does not involve recompiling or relinking an application program.

## 7. UEC Documentation

For many users, the weakest link in any technology or product is supporting documentation. OSF/Motif documentation will emphasize easy access to accurate and useful information by providing well-organised material, effective graphics, useful examples and high-quality indexes.

The documentation set for OSF/Motif will include a Style Guide, a UEC Application Environment Specification, reference manuals describing all programming interfaces, and a task oriented programmer's guide.

## 8. Native Language Support

The original UEC RFT specified that native language support (NLS) was mandatory for all user interface technologies submitted to OSF for review. NLS was raised repeatedly as an issue at OSF's November 1988 members' meeting, where member feedback on the submitted UEC technologies was gathered.

OSF/Motif's native language support will conform to the NLS solution proposed in X/Open XPG3 (the X/Open Portability Guide); will support Compound Strings technology from Digital Equipment Corporation (if accepted by the MIT X Consortium); and will further support the efforts of standards groups and the X Consortium in this area.

OSF/Motif will support mirror-image appearance and behaviour, necessary for right-to-left reading Semitic languages. Future offerings may include Asian-language text widgets. OSF intends to support the efforts of the X Consortium in this area and the areas of furnishing multiple-byte character sets, top-to-bottom rendering and input conversion methods.

## 9. Catalog Technology and Research Programs

OSF/Motif is comprised of technologies which OSF staff have identified as "foundation" or "core" technologies. It became apparent, with the UEC RFT, that many submissions were not yet appropriate for inclusion in the core technology for a number of reasons. Some were in technology areas where the market has not yet settled on a single model or set of techniques. Some were based on technology not yet widely available to some OSF members, or required base technologies that were not standardiise those technologies which fall into the second, or catalog program, area. As time progresses and acceptance of these technologies matures, OSF may move one or more of the technologies into the core offering. The final ring of the model, the OSF Research Program, includes areas of interest in which OSF is encouraging investigation. These technologies are not necessarily compatible with core offerings, and may not be portable to a wide variety of platforms; nevertheless, they represent areas which may have great potential value for the industry.

## 10. Early and Equal Access

A valuable aspect of OSF's open process is one of the Foundation's guiding principles – early and equal access. This means that all members of OSF have equal access to technologies that OSF is developing. OSF makes available licensed "snapshots" of code in development, which provide end users and applications designers with information on which to base business and technology decisions.

Early and equal access to technologies under development benefits system vendor, application provider and end user alike. The open process of OSF, as evidenced at member meetings, means that end users can specify the technologies they would like to have, in the knowledge that the providers of these technologies – the system vendors and independent software vendors (ISVs) – are in the same meeting, listening. The opportunity to exchange information, insights and perspectives is unequalled in the industry.

Clearly, the open process works. OSF's members are vocal, providing the OSF staff with technology direction and evaluation. The industry responds to RFTs. And OSF staff can, and do, make independent, vendor-neutral decisions, which the help of independent industry consultants and member feedback. The first technology offering of the open process, OSF/Motif, is available for license today. Snapshots are available today, and the entire offering will be tested and ready to ship by the summer of 1989.

The importance of the open process cannot be ignored. In less than six months, technology decisions were proposed and made that will bring a new, friendly face ot UNIX operating systems, a user environment that will open the gateway between the PC world, the proprietary operating system world, and the UNIX system world. End users will be able to move between PCs and powerful, networked workstations, all the way to mainframes, and see the same screens, the same tools, the same applications. The savings in training, application portability and portability across a wide range of platforms from PCs to mainframe, is enormous.

The commercialisation of open systems has begun, driven by the system vendors, applications providers and end user who are members of the Open Software Foundation. In the final analysis, the entire industry will benefit from the shared vision made possible through the open process.

## 11. Trademarks

OSF/Motif is a trademark of the Open Software Foundation.

# NFS refreshes the filesystem(s) A/UX cannot reach

*Matthew Kempthorne-Ley-Edwards*

*William Roberts*

Department of Computer Science
Queen Mary College
190 Mile End Road
London E1 4NS
UK
*liam@cs.qmc.ac.uk*

*ABSTRACT*

A/UX, the Apple Macintosh version of UNIX, coexists alongside the original MacOS operating system: separate partitions on the same disk with little real connection between the two. The potential for running MacOS applications under A/UX lead us to develop A/UX software which provides access to the MacOS files using standard UNIX file handling. This paper describes the A/UX environment and the techniques we used: a first version based on a library to simulate UNIX file system calls and the later version using the NFS protocol to mount the MacOS partition as a UNIX filestore. We offer some reflections on the problems and successes of our work and suggest a number of things which may help those providing NFS servers for other "exotic" filestores.

## 1. Introduction

This paper describes the work carried out a QMC during the summer of 1988, resulting in the production of an NFS server providing direct UNIX-style access to a standard Macintosh filesystem. Section 2 is a summary of A/UX which provides sufficient background information for Section 3 which states the problem which we are addressing and the approach we have taken. Section 4 describes the detailed internal structure of the Macintosh HFS filesystem, followed by Sections 5 and 6 which describe the two stages of our work. Section 7 highlights some of the issue and problems that we addressed, and finally Section 8 gives our conclusions.

## 2. The A/UX Environment

A/UX is based on System V Release 2.2 and has an extensive range of Berkeley facilities: it includes both streams and sockets, and goes into such details as the subtle differences between signal handling in the two systems (there is a system call controlling the degree of BSD compatibility required).

### 2.1. Booting A/UX

The most unusual aspect of A/UX is the way it is booted. A Macintosh II with A/UX *must* retain at least a minimal Macintosh native operating system (henceforth called MacOS), either on a floppy disk or more normally as a partition of the hard disk. The Mac II will therefore boot as a standard Macintosh with the familiar desktop and so on.

In order to boot A/UX itself, a MacOS application called "sash" (for Stand-Alone SHell) is used: this deals with reading the A/UX root partition and loading the desired UNIX kernel in much the same way as an ordinary disk or tape bootstrap program. In addition to this bootstrapping function, Sash acts a real shell and has a range of standard utilities such as fsck and ed. Sash can also invoke the interesting *eschatology* mechanism for recovering workable copies of vital files such as init, passwd and so on, but that is outside the scope of this paper.

Currently, the closest to a true automatic boot for A/UX is to set Sash as a Macintosh "startup application" which is invoked automatically when MacOS is booted. Sash also conflicts with some other Macintosh applications: Sash will not run if MultiFinder is active, nor if too many extras (e.g. lots of silly noises!) have been installed.

## 2.2. A/UX Disk Environment

Once running, A/UX proceeds with a conventional boot sequence controlled via /etc/inittab. The disk partitions are defined and named in a partition table understood by both MacOS and A/UX (though MacOS can only use one partition). Partitions are associated with UNIX physical devices by the *pname* command which allocates a *slice number* to a named partition on a particular disk: partitions used by A/UX must be initialised with mkfs in the normal way. The disk is formatted and partitioned using a graphical utility under MacOS: whole disk partitions can be picked up with the mouse and physically moved about on the disk! A/UX includes the System V File System Switch mechanism and the Network File System, so it can handle long names on remote filestores, but its local disk filestores use the normal System V filesystem with the 14 character maximum filenames.

| Index | Name | Type | Start, End | Sectors |
|---|---|---|---|---|
| 0 | MacOS | Apple_HFS | 96, 42194 | 42099 |
| 1 | Apple | Apple_partition_map | 1, 63 | 63 |
| 2 | Macintosh | Apple_Driver | 64, 95 | 32 |
| 3 | A/UX Root | Apple_UNIX_SVR2 | 42195, 62674 | 20480 |
| 4 | Swap | Apple_UNIX_SVR2 | 62675, 72914 | 10240 |
| 5 | Random A/UX fs | Apple_UNIX_SVR2 | 72915, 84293 | 11379 |

**Figure 1**: *The A/UX Disk Layout used at QMC*

Figure 1 shows the disk layout used on the Mac IIs at QMC: the 40 Megabyte disk is divided into 20 Megabytes of MacOS (equivalent to a Mac SE), 10 Megabytes of A/UX root partition, 5 Megabytes of swap partition and 5 Megabytes of /tmp. Subdirectories of /tmp can be used as a personal filestore if the user of a particular Mac II wants to do that. The /tmp partition is also potentially a second swap partition or even space that could be given back to the MacOS partition if desired. The full A/UX release is held on a VAX 11/750 fileserver and mounted via NFS. Most personal files are mounted from a Sequent Balance 21000, again via NFS.

## 2.3. Graphics under A/UX

The Mac II comes with a 640 x 480 pixel graphics screen, so one would expect A/UX to provide a window system comparable to that of say a Sun, an Apollo or even a Macintosh. Apple have provided a utility called the *toolbox daemon* which emulates the standard MacOS facilities: this emulation is sufficiently accurate to provide binary compatibility with graphics programs written in C for MacOS, allowing them to be run unchanged.

Unfortunately, the current release of A/UX (1.0 at time of writing) has only limited support for facilities such as the Macintosh printing and network managers, and has a limitation that only one such MacOS program can be run at a time. A/UX includes a multi-window terminal emulation program comparable to SunView with command tool windows, but unlike SunView the terminal emulator must be closed down before any graphics programs can be run. This does allow you to run one "well-behaved" Macintosh application such as MacDraw II, which through the toolbox daemon accesses files in the local System V filestore† as though they were Macintosh files. There is no graphical desktop (nor would we want one).

Not surprisingly, Apple promise to continue to extend the capabilities of the toolbox daemon, but they have also announced that A/UX release 1.1 will include the X window system. The present facilities do provide an incentive for developers to write "well-behaved" applications, but a large number of famous-name applications are not yet well-behaved.

† The toolbox daemon reads directories as files and interprets the contents itself. Naturally enough, a BSD filestore mounted via NFS has a different directory structure so the toolbox daemon can't make sense of it. BE WARNED: to write portable software, you MUST use the opendir/ readdir group of routines.

## 2.4. A/UX and MacOS Interworking

The major development effort at interworking between the A/UX world and the existing MacOS world seems to have been in the development of the toolbox daemon. A/UX provides no utilities which can read the MacOS partition, even though there must be one for boot purposes, and no support at all for any of the Apple networking protocols or the Apple LocalTalk networking hardware. The only mechanism for transferring a well-behaved MacOS application into A/UX is to put it onto a single-sided MacOS floppy, in which case an A/UX utility can read it.

## 3. The Problem and the Proposed Solution

At QMC, we intend our users to have both the MacOS and A/UX environments available to them: the rest of this paper describes our work aimed at making the MacOS disk partition accessible to A/UX. There are a number of reasons why we need to do this, many of them to do with licencing of Macintosh software, but originally we simply needed a sensible way to transfer files from the MacOS environment into A/UX.

The key to the problem of using an alien filestore under an operating system such as UNIX depends on how that filestore can be accessed. In this case, the only way was to read the MacOS disk partition as the raw disk device. This means that the techniques used could be adapted easily for other machines with similar problems. Everything was designed from scratch, including directory reading and location of file blocks; we could not use the toolbox daemon (because that would prevent us from running the programs once we had access to them) and in any case, the toolbox daemon provides no access to the MacOS partition. As no MacOS routines are available under A/UX, the necessary routines had to be written starting from the lowest level, which in this case is direct disk access.

Our initial solution was to write an emulation of the UNIX file handling routines so that source programs could be recompiled to use the MacOS filestore without needing major modification. For complete compatibility, all of the UNIX file handling routines must be emulated. A simple set of defines for all these routines (and any associated types), will then convert many existing programs to work on the Mac filestore, rather than the A/UX filestore, but this can never provide complete source compatibility for programs which involve fork (for example).

Given a working library, it can be used to recompile an NFS server program, so that the resulting server would provide NFS access to the MacOS filestore. An NFS server makes only simple demands on its access to the filestore, so a basic library should be sufficient. A/UX includes NFS, so such a server makes it possible to mount the MacOS filestore as part of the A/UX filestore. Once the MacOS partition has been NFS mounted as part of the A/UX filesystem, any UNIX program could make direct use of Macintosh data and program files without any modification at all. This approach also avoids needing kernel source code, because all of the NFS protocol can be handled by ordinary user processes.

Ultimately, the A/UX kernel itself could support the MacOS filesystem directly through the System V File System Switch mechanism. We didn't get this far and currently have no plans to obtain the necessary A/UX source licences.

## 4. MacOS Filestore Structure – HFS

To understand the structure of the library routines and the problems involved in putting a UNIX interface onto the MacOS filestore, it is first necessary to explain the way MacOS uses its disk to build the *Hierarchical File System* (HFS).

HFS uses standard SCSI disks with 512 byte sectors, and the disk device driver presents the disk as a linear sequence of sectors. Rather than referring to sectors directly, the disk is considered as a collection of *Allocation Blocks* each of which is a number of consecutive sectors. Allocation blocks have a fixed size on any particular disk, but may vary between disks; a very large disk would need to use large allocation blocks to avoid running out of block numbers (16 bits). The size of the allocation block (in sectors) is written into the first sector of the disk; this information must be obtained before any access to other data, to enable the correct sector(s) to be calculated from their allocation block number. On the disks we used, the allocation block size was always 1 sector.

HFS organises the storage of data and directories in terms of *Extents*, which are variable length sequences of consecutive allocation blocks. Each extent is recorded as a starting allocation block number and a length in allocation blocks. Files are then described by headers which contain a list of extents (typically 3 or less). This structure also helps to avoid fragmentation of the disk into isolated allocation blocks. Figure 2 illustrates this structure.

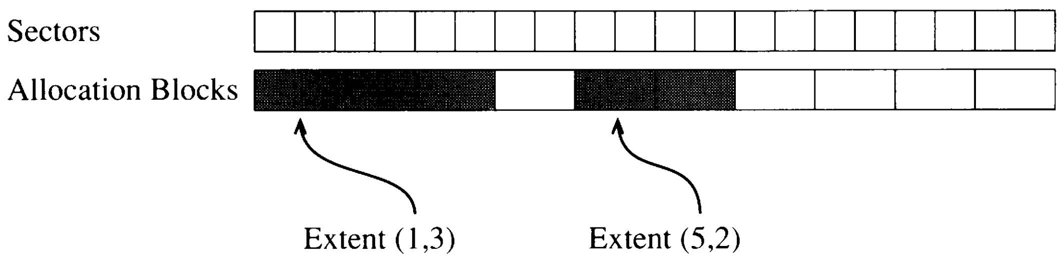


**Figure 2**: *Disk Surface Organisation in HFS*

As mentioned above, the first sector on the disk is used to hold information about the whole disk, such as the allocation block size and the sector offset for the start of the first allocation block: this first sector is known in MacOS terminology as the *Disk Volume Information Block* and is equivalent to the UNIX super-block. Besides historical information such as when the disc was formatted, this block contains the first three extents for the root of the directory system. When the library first opens an HFS disk, it reads the Disk Volume Information Block and saves this information for use in subsequent calls to that disk.

## 4.1. Locating Files on the Disk

In order to locate the contents of a file, a list of extents is required to tell the program which allocation blocks are used. Each file is described by a directory entry which holds the first 3 extents for the file contents. If a file is fragmented into more than three extents, then it is necessary either to copy the file (in the hope of reducing the fragmentation) or to identify the additional extents from a disk structure known as the extents tree. The library only deals with files that have three or less extents (i.e. it does not handle the extents tree), in which case the extent information is stored in the replacement FILE structure created when the Mac file is opened by the library. This extent information, together with the volume information block, makes it simple to calculate the sector offset of any sector in the file.

## 4.2. HFS Directory Structure

All HFS files are listed in a structure called the *B* Tree* which occupies the allocation blocks listed in the three extents given in the Volume Information block: this corresponds to the table of inodes on a UNIX filestore. In place of the inode number, every file has a unique ID number which is used to locate the file record in the B Tree.

Each allocation block in this tree is either a link node, a leaf node, or not used: each node is split into a number of records. All the nodes are marked with a level number and optional left/right links, where the top node is level 1, all the nodes directly below that are level 2, all below that level 3 etc. All leaf nodes are on the same level on this type of tree, and all unused blocks are level 0. The root node can be located by looking for a node of level 1 with no links left or right.

| | |
|---|---|
| 4 bytes | Link to right node in tree (or 0) |
| 4 bytes | Link to left node in tree (or 0) |
| 1 byte | Node type (0xFF = leaf, 0 = link) |
| 1 byte | Tree level (0 = unused node) |
| 2 bytes | Number of entries in this node |
| . | |
| . | data area for entries |
| . | |
| 2 bytes | offset of Nth entry in this node |
| . | |
| 2 bytes | offset of 1st entry in this node |

**Figure 3**: *Structure of a B-Star Node*

The individual entries contained within a node may be of variable length, so the node contains a table of offsets pointing to the start of each entry; this table grows downwards from the end of the block.

Link nodes hold KEY entries which contain the filename, the ID of the directory that contains it, and the number of the node which has that file as its first entry. The B Tree holds these key entries in sorted order so that finding a particular (filename, Directory ID) pair is a matter of scanning the top node to find the last entry less than or equal to the desired key, then repeating the process on the node referred to by that entry. This continues recursively until a leaf node is reached, in which case the node contains the actual file details.

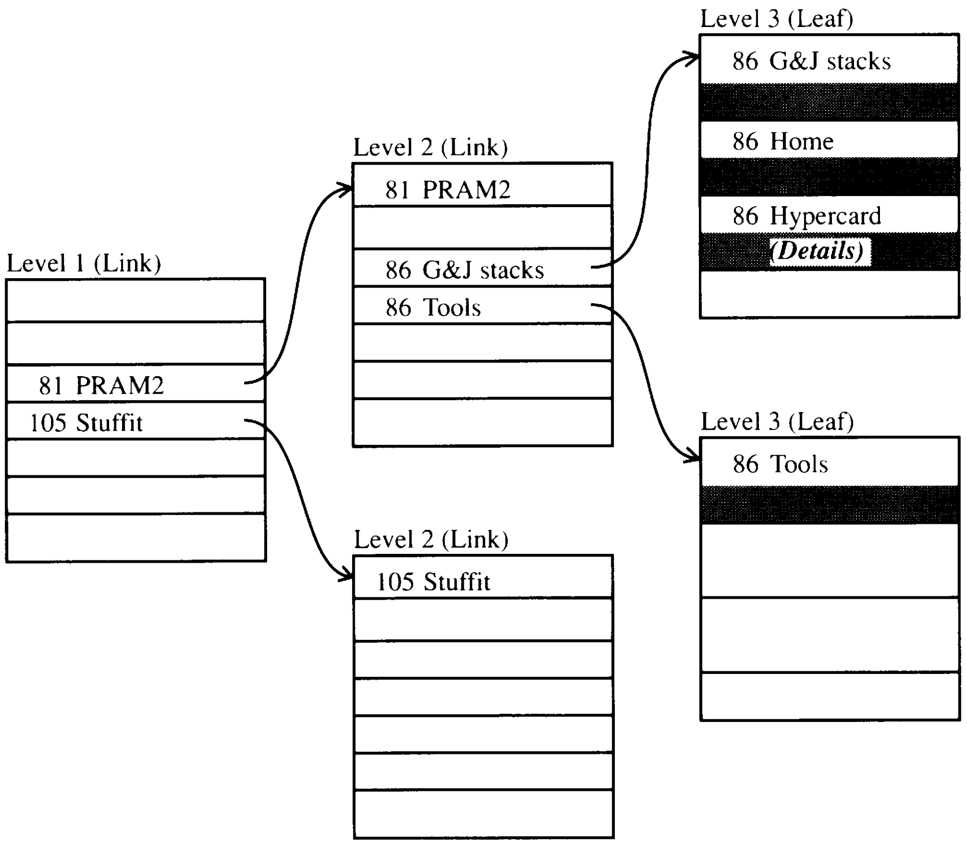


**Figure 4**: *An HFS B*-Tree*

The use of a B Tree allows quick access to any entry in the directory system, for example 4 block accesses to find an entry in a several thousand files, even with the comparatively small block size used by HFS. New entries can be added to the tree without it becoming lop-sided, and the sideways links between the leaf nodes make it easy to scan a consecutive range of keys. For more information on the general B Tree algorithms, see [Bay72a, Cla88a].

## 4.3. Leaf nodes

Leaf nodes contain the actual file entries, where each entry is either a FILE, a DIRECTORY, or a parent directory pointer. They have the same KEY part as for the link nodes, but a type code indicates the interpretation and length of the rest of the entry.

A *DIRECTORY* entry contains just the new directory ID number and three timestamps; creation time, last modification time and last archive time. The contents of a directory can only be determined by scanning the B Tree for entries which have the appropriate ID number in the key, but the ID number is the most siginficant part of the KEY and so the entries will be consecutive and can be scanned quickly using the sideways links between leaf nodes.

A *LINK* entry is similar to '..' (dotdot) in UNIX and contains just the ID of the parent directory.

The most complicated is the *FILE* entry which has the structure shown in figure 5.

| | |
|---|---|
| 1 byte | MacOS Flags |
| 1 byte | File type (?) |
| 16 bytes | Finder information (first 16 bytes) |
| 4 bytes | Unique File ID |
| 2 bytes | not used |
| 4 bytes | Length of Data Fork |
| 6 bytes | not used |
| 4 bytes | Length of Resource Fork |
| 4 bytes | not used |
| 4 bytes | Creation time |
| 4 bytes | Last Modification time |
| 4 bytes | Archive time |
| 16 bytes | Finder information (last 16 bytes) |
| 2 bytes | not used |
| 2 bytes | Start allocation block number (this and the next 2 bytes make one extent) |
| 2 bytes | Number of allocation blocks in extent |
| 8 bytes | The next two extents of the data fork |
| 12 bytes | The first 3 extents of the resource fork |

**Figure 5**: *Structure of a FILE entry*

The Macintosh documentation available [Inc87a] was not explicit about the contents of all of these fields, though it gave lots of details about the results of various ToolBox routines for low level disk access. Even strenuous efforts on the part of SRL Data (our A/UX support, and the only people we spoke to who seemed to understand the significance of our approach) failed to obtain this basic information. Without knowing what to put in the unknown parts of these entries, it is not possible to attempt to create new files or write to existing ones: reluctantly we therefore restricted ourselves to READ-ONLY access to the HFS partition. This is also the reason why the library does not use the extents tree: we don't even know where to find it, let alone what it contains.

## 5. First Solution: The Library

The first objective was to produce a library that gave access to the HFS filestore through UNIX compatible routines for handling the directories and files. An include file containing some simple define instructions should then be able to swap all UNIX file handling routines for the equivalent HFS ones, enabling existing programs such as "ls" to be run as a test of the routines.

Some of the UNIX routines have associated data types that must also be compatible, in particular the type FILE. This has to be done carefully, so that it can hold all the information required for the HFS routines but still be used for handling normal UNIX devices and pipes: stdin, stdout and stderr are all pointers to type FILE. To deal with this, a FILE structure was designed that contained the existing UNIX FILE structure as well as the additional information required by the library.

The library uses a set of routines that perform basic operations and can locate information on the disk, e.g. to find out where a directory is, and where file blocks are on the disk. These routines are called by the main emulation routines, which will be described in groups.

### 5.0.1. The Directory Handling Routines

These routines emulate opendir, readdir, seekdir, telldir and closedir, together with the (opaque) type DIR which is used as a "placeholder" for scanning a given directory. The default directory is assumed to be the root directory of the HFS disk, so all paths start from there (recall that the HFS B-Tree has keys consisting of a filename AND the ID of its parent directory). These routines are complicated somewhat by the lack of a single file containing the directory information: instead, the contents of a directory are merely implied by the keys in the B-Tree, though they are stored in a convenient order.

The routine Mac_opendir was written to locate the given directory, and to return a value of type Mac_DIR. Declaring a new structure allows the library to store any information useful to the other directory operations; fortunately DIR is not intended to be interpreted by programs using opendir. The define statements used with Mac_opendir were:

```
define opendir Mac_opendir
define DIR Mac_DIR
```

All of the other library routines use a similar scheme.

Mac_readdir emulates readdir and so must find the next entry in the directory associated with the given Mac_DIR structure, returning a struct stat identical to that used by A/UX. The Mac_DIR structure includes a counter so that Mac_telldir can return a number referring to the current entry.

All MacOS files have an internal structure consisting of up to three parts: an optional *data fork*, an optional *resource fork* and some *Finder information* such as comments, file type, last backup etc. The A/UX toolbox daemon expects to find files in *AppleDouble format* which stores the data fork in one file and the resource fork (plus finder information) in another file. This requirement complicates Mac_readdir somewhat, as it has to produce more answers than exist in the natural HFS directory. Not all files have both forks, and it was decided to provide an entry for the data fork only in the case where there is actual data to be in it, i.e. no null-length data forks.

HFS directories do not have entries that correspond to the UNIX notion of '.' and '..', so Mac_readdir creates fake entries for them, numbering them zero and one respectively and returning them when the counter in the Mac_DIR structure has the appropriate number. For the duplicate entries (data and resource forks) however, the information describing the second entry is created at the same time as the first entry and a flag is set. The next call to Mac_readdir will see the flag and return the stored information for the second fork, clearing the flag as it does so. This enables the directory to be enumerated without too much difficulty.

Mac_seekdir needs to move the DIR pointers to a specific entry in the directory, but is complicated by the fact that some files may generate two entries. The simplest way to deal with this is to scan through the directory again, incrementing the counter for each entry that would be generated and stopping when the desired number is reached. To avoid some unnecessary work, the algorithm used took account of the last read position as follows:

```
If SEEKing forwards ( SEEK position > last READ position )
    read the entries repeatedly until the correct number is reached.
ElseIf SEEKing backward ( SEEK position < last READ position )
    Start from Beginning of the Directory
    Read entries repeatedly until the correct number is reached.
```

This is not the quickest or best method, but as Mac_seekdir is hardly ever used it is satisfactory for the purpose.

### 5.0.2. The Stat Routines

The Mac_stat and Mac_lstat routines get information about a particular file, returning it as a stat structure. To do this, the file entry in the relevant directory on the Mac filestore is looked up and the required information obtained from the Finder information part of the file. The HFS filestore has no notion of symbolic links, so Mac_lstat and Mac_stat are the same routine.

MacOS does not include the notion of "user", so the Apple filestore does not have such things as owner and group ID. Files do have a "creator" but this refers to an application rather than a human being. There is some information about file permissions, but only at the level of whole directories. This does not map easily into the UNIX permissions system, so the library invents suitable permissions; no write permission is granted because the library only provides Read access.

HFS files have types given as four character names in the Finder information, and type "APPL" (meaning Application) was taken to imply that the file should have execute permissions. Other stat information which had to be forged included setting the number of links to be 1 and calculating the block size for the appropriate fork. Resource forks have an AppleDouble format header inserted at the beginning by the FRead routines, so Mac_stat has to return a suitably increased length for the fork. The stat structure is NOT redefined, as all application programs make direct use of its internal structure.

### 5.0.3. The File Handling Routines

The routines emulated are:

fopen, fclose, fread, fseek, ftell, fwrite, fscanf, fprintf, fgetc
open, close, read, lseek, tell, write

A new type FILE is declared for the library, along with suitable redefinitions for stdin, stdout and stderr. The new Mac_Stdin, Mac_Stdout and Mac_Stderr contain the corresponding stdio definition as a field in the Mac_FILE structure. Special checks in the emulations of fread, fwrite, fprintf etc. identify use of these values and substitute the original stdio definition in the original stdio call; this retains the ability to use command line redirection such as > and < with programs recompiled under the library.

The system call emulations (open, read, write etc) are implemented as calls to their stdio equivalents (fopen, fread, fwrite) in order to have access to the extra information in the Mac_FILE structure; the library maintains an array of Mac_FILE structures in much the same way as UNIX maintains the per-process open file table.

The Mac_fopen routine checks the access requested and returns EROFS (Read-only File System) to everything except straight read access. The file is located on the disk (if it exists) and a free Mac_FILE structure is initialised with the relevant details, including the length of the file and the three extent records which describe the location of its data on the disk. There is also a buffer big enough to hold a single allocation block, with fields describing the current position in the file as the offset into the buffer, the extent containing this allocation block and the index of the block within the extent, e.g. byte 4 in allocation block 2 of extent 1.

When reading from the file, data is copied from the buffer until it is exhausted, and the buffer is refilled as necessary working through the allocation blocks of the extents. If the file name refers to the resource fork (indicated by a leading percent sign) then the first 256 bytes of the file are the appropriate AppleDouble format header, with the actual resource data following afterwards.

## 5.1. Examples of Using the Library

The routines were organised as a library and tested on two applications. The range of test applications is small because our ultimate goal was to build an NFS server using the library, and the facilities required for that are fairly straightforward.

### 5.1.1. The 'maccopy' program

This short program (Figure 6) copies files from the HFS filestore to the A/UX filestore so that they can then be run by the toolbox daemon. This wastes disk space by holding two copies of a single program, but is a better than nothing.

The "undodefs.h" file reverses all of the masking defines, requiring the programmer to make explicit which routines and data structures are part of our library and which ones use stdio.

### 5.1.2. Converting ls

The conversion of the standard 4.2BSD source for 'ls' involved no more than adding the instruction to include "Macfile.h".

Figure 7 shows the output of macls when applied to our Macintosh Hypercard folder; note that the Hypercard application is entirely a resource fork and so only appears as %Hypercard.

## 6. Second Solution: NFS server using the Library

Once the library had been written, the next goal was to produce an NFS server [Sana] which could provide access to the HFS filestore for all UNIX clients. Rather than writing an NFS server from scratch, the code for UNFSD (comp.sources.unix) was modified to use our library†. The main purpose of such a server would be to mount the local HFS partition onto A/UX, rather than supply files for other A/UX machines, but nothing in the UNFSD code makes it impossible to access the filesystem remotely.

† UNFSD was written by Mark Shand at the Joint Microelectronics Research Centre, School of Electrical Engineering, University of New South Wales, AUSTRALIA (shand@cad.jmrc.eecs.unsw.oz) and posted to comp.sources.unix. It comes complete with documentation and UNFSMNTD, a replacement for the NFS mount daemon.

```
/*
 * A/UX HFS library and utilities.  Version 1.0
 *
 * Copyright: Matthew Kempthorne-Ley-Edwards, August 1988
 *
 * for Department of Computer Science
 * Queen Mary College
 * London E1 4NS
 * UK
 *
 * This software is provisional and carries no guarantee of fitness
 * for any purpose.  It may be freely copied provided this copyright
 * message included in any such distribution.  It may not be sold or
 * incorporated into any product without prior permission of both
 * the author and Queen Mary College.
 */

 include <malloc.h>
 include "macfile.h" /* sets up all  defines */
 include "undodefs.h" /* undo defs for open, close to mac */
 define BUFSIZE 50000

main(argc,argv)
int argc;
char *argv[];
{
        Unix_FILE *outname;
        Mac_FILE *inname;
        char *Buffer;
        int bytes;
        if (argc!=3) {
                printf("Usage :%s <macfilename> <unixfilename>\n",argv[0]);
                exit(-1);
        }
        if ((inname=Mac_fopen(argv[1],"r"))== NULL ) {
                printf("%s not found in filestore\n",argv[1]);
                exit(-1);
        }
        if ((outname=Unix_fopen(argv[2],"w"))==NULL) {
                printf("%s not writable on unix filestore\n",argv[2]);
                Mac_fclose(inname);
                exit(-1);
        }
        Buffer=malloc(BUFSIZE);
        if (Buffer==NULL) {
                printf("%s out of memory.\n",argv[0]);
                exit(-1);
        }
        while ((bytes=Mac_fread(Buffer,sizeof(*Buffer),BUFSIZE,inname))>0)
        {
                Unix_fwrite(Buffer,sizeof(*Buffer),bytes,outname);
        }
        Mac_fclose(inname);
        Unix_fclose(outname);
        free(Buffer);
}
```

**Figure 6**: *The maccopy program*

A few modifications were required in order to use the library with the nfs server, mainly concerned with the use of UNIX open to handle network communications; the use of explicit library names (rather than the defines) dealt with most of these. The UNFSD server only implements read-only NFS mounts, which corresponds well with the read-only HFS file handling implemented by the library.

The main design decision was choosing the contents of the opaque *file handle* to be quoted by clients of the server when accessing files: the NFS protocol provides 16 bytes for this purpose, so we used the HFS file ID number and a number to distinguish the particular HFS partition involved.

```
aux0% macls -ial /Hypercard
total 523
   93 -r-xr-xr-x  1 root       377386 Jan 22  1988 %HyperCard
   97 -r--r--r--  1 root        36660 May  1  1987 %ImageWriter
   98 -r--r--r--  1 root        25335 May  1  1987 %Laser Prep
   99 -r--r--r--  1 root        50410 Oct 28  1904 %LaserWriter
   86 dr-xr-xr-x  2 root          512 Nov 21 08:50 .
    2 dr-xr-xr-x  2 root          512 Nov 21 08:50 ..
   90 dr-xr-xr-x  2 root          512 Jun  6  1988 Diary Stacks
   91 dr-xr-xr-x  2 root          512 Jun  6  1988 G&J stacks
   92 -r--r--r--  1 root        32768 Oct 28 03:47 Home
   94 dr-xr-xr-x  2 root          512 Jun  6  1988 HyperCard Dev. Toolkit
   95 dr-xr-xr-x  2 root          512 Jun  6  1988 HyperCard Stacks
   96 dr-xr-xr-x  2 root          512 Jun  6  1988 Ideas Stacks
  308 dr-xr-xr-x  2 root          512 Jun  6  1988 Mel's Stacks
  100 dr-xr-xr-x  2 root          512 Jun  6  1988 More Stacks
  101 dr-xr-xr-x  2 root          512 Jun  6  1988 Other Stacks
  102 dr-xr-xr-x  2 root          512 Jun  6  1988 Tools
```

**Figure 7**: *Output of "macls -ial /Hypercard"*

```
lhcl  /etc/mount -o soft,bg,ro aux0:mac: /mnt
lhcl  cd /mnt/Utilities
lhcl  ls
%BWIIC                %MakePaint v1.1       %cachectrl
%CIcon Edit           %New Colorize         %interlace INIT
%Compare              %PRAM2                %roff
%Dialog Creator       %Paint Cutter?        Apple utilities
%DivJoin 1.0d9        %Password             CIcon doc
%Fedit Plus           %Password Docs        Copy II Mac Folder
%Fontastic? Plus      %RMOVER               Password Docs
%ICON Designer        %RMaker               Postscript utilities
%Interferon           %ResComp 1.0          QMCHeader
%JoyPaint             %ResEdit 1.1d4        bwIIc.doc
%Localizer            %Screen Maker         interferon doc
%MacID                %ScreenMaker2         network tools
lhcl  cd /mnt
lhcl  ls
 Applications     A/UX Startup      Matts Stuff       foobar
%Crystal Quest    C Folder          System            lecturers advert
%Desktop          Comms             Utilities         pd's rubbish
%temp.rsrc        Hypercard         beeb to 21k       scratch
1510 to 21k       Martins Stuff     chapter3.txt      temp
lhcl  cd Comms
lhcl  ls
%BinHex4          %MacTerminal 2.2  %sequent.2        sequent.2
%BinHex5          %PackIt III       StuffIt
%FreeTerm 3.0B2   %echo             Telnet
lhcl  du
111     ./StuffIt
153     ./Telnet/old telnets
352     ./Telnet
683     .
lhcl
```

**Figure 8**: *Using the NFS server*

Figure 8 shows the HFS filestore being mounted on a Sun and examined with standard UNIX utilities. Note that strange characters in filenames (e.g. Fontastic™ Plus) are translated into ? by the library.

## 7. Observations and Anecdotes

There were a few problems encountered during development of this system, some trivial and some more serious.

## 7.1. Name of '..' (dotdot)

Library routines such as getpwd have to work out the name for . (dot) by scanning the parent directory .. (dotdot) for a name with the same inode number. The problem comes with deciding when to stop, i.e. identifying the root directory. Our initial solution was to make the (forged) entry for .. in the HFS root directory have the same inode number as the root itself. This appeared to work fine on Suns, but under A/UX the C-Shell would dump core when executing the command pwd. This didn't happen with csh on other machines, but the A/UX csh program was not derived from the Berkeley original for licencing reasons; Apple wrote it themselves.

Examining the diagnostic output from UNFSD, the problem was finally narrowed down to the inode number associated with the root inode (as you will have guessed!). Under A/UX the inode number for the root directory HAS TO BE 2. The directory IDs 0, 1 and 2 are reserved in the HFS file system, where 0 is the blank file system, 1 is the desk top, and 2 is the root of the filestore in question: the UNIX analogy would be that all real disk filestores are "mounted" on mount points in a logical directory called "DeskTop", itself a subdirectory of a logical root directory. The parent ID of the root directory of a filestore is therefore stored as 1, and the library has to check for this and supply 2 instead as the parent directory ID of the root directory.

## 7.2. MacOS Filenames containing '/' (slash)

Filenames in MacOS can contain almost any character from a 256 character set, including the UNIX path separator character '/' (slash). To get around this, characters which can not be used in UNIX filenames are converted to a questionmark by the name reader in the directory reader routines. A special case was made for the slash character as the library has to handle directory separators; the library looks for the filename in the current directory which matches the longest slash-terminated prefix of the remaining path, including slash characters. This allows access to directories with otherwise problematical names, e.g. "A/UX Startup".

## 7.3. Providing Several NFS servers on one machine

The NFS mount daemon does not return the port number to be using in contacting the NFS server for a particular filesystem. This leaves the option of specifying an explicit port number as an option to the mount program, or providing all of the NFS services within a single server. We used the "single server" approach because it was easy enough to do; the A/UX and HFS filestores are distinguished by having the HFS filestores prefixed by "mac:" in the mount request (as per Figure 8), but this means that an A/UX machine running our server cannot provide NFS write access to its own System V filestores. This is not a problem for local access, or at QMC where all of the Mac II disks are small, but may be a problem for other sites wanting to use this software. It would be better if Sun extended the mount protocol to supply the port number of the NFS server as well as its blessing on the mount request.

## 7.4. One Disaster

During an enthusiastic demonstration of the modified NFS server, we mounted the HFS partition on the A/UX partition and executed the Macintosh BinHex4 program from it, using the toolbox daemon. Selecting the "Open" option from the file menu produced the usual Macintosh dialogue listing the likely looking files from the A/UX root directory. We perversely selected the HFS directory, whereupon the system went mad and eventually destroyed the /tmp partition. It is still not clear which part of the system to blame, but it definitely wasn't our server. The toolbox daemon attempted to open the HFS directory as a file and presumably ignored the error message, but that should not have damaged an unrelated filestore, especially by making read requests!?

## 8. Conclusions

The NFS protocol is sufficiently flexible to accommodate even very non-UNIX filesystems, though the suggested extension to the mount protocol would be very helpful. Our NFS server provides a convenient way to overcome a basic deficiency in A/UX and will be increasingly useful as the A/UX system matures. Who knows, perhaps we will even penetrate the wall of secrecy and discover how to *write* HFS files.

## References

Bay72a. R. Bayer and E.M. McCreight, ``Organization and maintenance of large ordered indices,'' *Acta Informatica*, vol. 1, no. 3, pp. 173-189, 1972.

Cla88a. K. Clarke, ``A Pascal Implementation of B-Trees,'' Computer Science Departmental Report, Queen Mary College, March 1988.

Inc87a. Apple Computer Inc., *Inside Macintosh*, IV, Addison-Wesley, April 1987.

Sana. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, *Design and Implementation of the Sun Network Filesystem*, Sun Microsystems.

## Acknowledgements

The authors would like to thank the Department of Computer Science at Queen Mary College for funding this work and providing the Macintosh II on which it was carried out. Thanks are also due to Mark Shand for providing the UNFSD software which arrived over the USENET at such a convenient time, and to SRL Data for their assistance in getting detailed information from Apple.

# A Contiguous High Performance File System

*Shigetoshi Yokoyama, Shin-ichi Yamada,*
*Takehiko Nishiyama, and Kazuo Ito*

Development Headquarters
NTT DATA COMMUNICATIONS SYSTEMS CORPORATION
2-11-6 Kita Saiwai, Nishi-ku, Yokohama-shi, Kanagawa 220, Japan
*yoko@nttdpe.ntt.jp*

*ABSTRACT*

Recently the UNIX operating system is ported to various machines, and in proportion to its expanse, UNIX's application area grows. One of its typical new applications is an operating system for multi-media workstations. In this application, UNIX must efficiently handle huge amounts of data, such as for images and video. According to the media properties, some media must be accessed in real-time.

In this paper, first, the problems and limitations of UNIX System V manipulating these huge data files are discussed. It is concluded that the best way to solve these problems is to support a contiguous file system in UNIX. Second, implementation of a contiguous high-performance file system needing no modification of file access semantics is described. Third, the following two key technologies to increase the performance of HPF are proposed.

- High speed directory search
- High speed free space search

And last, the contiguous free space search performance of this file system from the result of a simulation is shown.

## 1. Introduction

The UNIX operating system is originally designed for the software development environment, and it mainly handles text data files, the sizes of which are less than one hundred Kbytes. In addition to this, the file access need not be finished in a predetermined time (i.e., in real-time), but, as the UNIX operating system is ported to various machines, its application area grows. One of its typical new applications is an operating system of multi-media workstations. In this application UNIX must efficiently handle huge amounts of data. That is, graphic, image, audio, animation, and video data. According to the media properties, some of these media must be accessed in real-time. In particular video data can't be handled unless the file I/O bandwidth is increased to several hundreds of Kbytes per second continuously for certain periods of time. That means some hundreds of Mbytes in total. (For example, if Digital Video Interactive algorithm is used to compress video data, the lower limit is 150K bytes/sec.)

In this paper, the problems of UNIX system V manipulating these huge data files, the implementation of a contiguous high-performance file system which resolves these problems, and the efficiency of this file system by simulating the file allocation algorithms are discussed.

The unique advantages of this new file system are as follows.

(1) It can achieve a file input/output speed almost as fast as the hardware transmission speed, and fast enough to manipulate even video data.

(2) It can be applied to a huge file system such as an optical disk file system which will be more popular for the storage of multi-media data.

(3) It can support UNIX file semantics without any changes to the application programs.

## 2. Problems (Performance limitation factors)

There are two major problems limiting UNIX file access performance in large files.

(1) Each ordinary and directory file is randomly located block-wise in the disk.

(The size of block is relatively small.)

(2) The large file can't be accessed without a multi-level index search.

So, disk seeking time lengthens when creat/open/read/write system calls are invoked. Because the optical disk's physical seek time is much longer than that of magnetic disks in current technology, the above problems become more serious.

## 3. How to solve the problems

These problems can be solved in four ways.

**Solution 1**:

Increase the size of blocks, and decrease the randomness of block locations.

(e.g. Berkeley's fast file system: cylinder group implementation)

**Solution 2**:

Utilize the raw file access method directly from application programs.

**Solution 3**:

Utilize the one-level storage. (ex. Memory mapped file facility)

**Solution 4**:

Support the contiguous files in UNIX kernel

In solution 1, performance is insufficient for video data manipulation.

In solution 2, the hardware dependency and the lack of the UNIX file system semantics are big disadvantages.

In solution 3, the performance is increased only if the file access has locality, so it is not efficient for the sequentially accessed file, such as for audio or video data.

In solution 4, file read/write access performance is as fast as hardware transmission speed for the sequentially accessed files. Because the blocks of the file are located physically contiguous in the disk, only one disk seek operation is needed in order to access that file.

The above explanations are based on the implementation of contiguous file system in the UNIX kernel. From now on, the file system will be referred to briefly as the HPF (High Performance File system).

## 4. Issues in the implementation

In this chapter, HPF implementation methods, including application interfaces, are discussed as well as HPF estimation.

### 4.1. System call interfaces

File access system call interfaces aren't changed, but it may be better for the application programs to specify the file sizes because of the following reasons. HPF has a file extension facility, so the created file grows automatically when it needs more file space than allocated. This extension, however, occurs during the write operation, so in order to increase write performance, it is better to pre-allocate necessary blocks in the creation time. For this reason, in HPF, the *fcntl* system call can control the initial file size and the unit size of growth.

Since the other system call interfaces, *read/write/lseek/close*, are the same as the originals, every current UNIX tools can be used on HPF. If necessary for performance, slight modifications can be made. These system calls are discussed below.

**(1) Create a file** *(creat/open)*

File creation has two phases.

**First phase**: i-node allocation

The system allocates a free i-node entry in the memory and loads a disk i-node. The parent directory is modified at this phase, using the path-name look up routine.

**Second phase**: File space allocation

The system allocates a free disk space of user specified size from the table of free blocks. The size and the disk address of the file are stored in the i-node allocated in the first phase.

**Delete a file** *(unlink)*

File deletion has two phases.

**First phase**: File space release

The system modifies the table of free blocks to remember that the specified file is released.

**Second phase**: I-node release

The system releases the i-node and the parent directory is modified at this phase also using the path-name look up routine.

**(3) Read from an ordinary file** *(read)*

From the read pointer in the file structure, the system can find a corresponding disk address using the disk address information in the i-node assigned in the creation sequence. Then the system calls the disk driver routine to read the data.

**(4) Write to an ordinary file** *(write)*

The methods to convert the write pointer to the disk address and to call the disk driver routine are the same as that of the read operation, but in the write operation, the file extension may be needed if the next write pointer is greater than the allocated file size.

The directory file manipulation is discussed later in section 4.4, because it is the first key in our file system.

## 4.2. Increasing the performance of the system call

As mentioned in the previous chapter, the performance of the *read* and *write* system calls is quite satisfactory if the file allocation size is suitably large. So, concern is mainly about file creation and deletion performance. In particular, creation performance is important to enable file access in the predetermined time. This performance depends on the performance of the path-name look up routine and the contiguous free space allocation.

For the above reason, the following speed up strategies are the points in our implementation.

**(1) Speed up the directory search**

In the original UNIX operating system, several directory searches occur to access a file. In general, the directory files needed to find an accessed file are randomly located in the file system. For this reason, the deeper the directory path, the longer the seek time. So, a way to decrease the time seeking the directory files is important.

**(2) Speed up the contiguous free space search**

When a file is created in HPF, a contiguous file space in the disk must be found. In sequential search algorithm, it is difficult to get large contiguous free space in a short time, so a fast algorithm to find a contiguous free space is also important.

## 4.3. HPF management mechanism

HPF is constructed with the following elements.

**(a) The global information** *(Super block area)*

- *Common super block*
  The common information as original UNIX super block.
- *HPF specified super block*
  The disk address and size of the following areas.

**(b) Free block information** *(Free block bitmap area)*

The management information for the free space in *data area*. That is, the table of free blocks.

**(c) Directory information** *(Directory area)*

The directory tree structure's information is stored in a special form.

**(d) I-node information** *(I-node area [i.e., i-list])*

Each file's information, such as owner, created time and so on, is stored in the i-node.

**(e) User data** *(Data area)*
User data are stored in this area. This area is constructed of many fixed length blocks and the system allocates a contiguous region per file.

The *superblock area* must be located at a predetermined place. This is at the top of the HPF disk partition.

For the performance improvement of the seeking directory and file information, a *directory area* and the *i-node area* are in the physically contiguous location. For the compatibility of file access semantics, the link facility must be supported. So, the *directory area* and *i-node area* can't be merged. The correspondence between them expresses the linking information. The physical disk map of HPF can be seen in Figure 1.

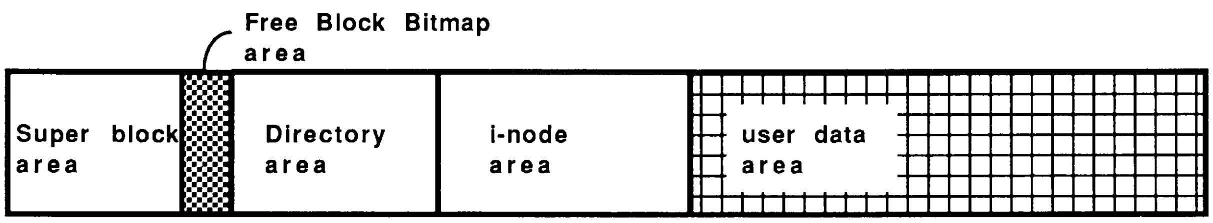


**Figure 1**: *The elements of HPF*

## 4.4. The method to speed up directory search

As mentioned above, the directory files are located randomly in the disk and the directory search speed is slow if the directory depth becomes great.

In HPF, there is no directory file. That is, the directory information isn't stored as a file. It is constructed by a double-linked list expressing both the correspondency between path-name and i-node number, and the information on the directory tree structure (the double pointers between parent and child directories, between children ones) (see Figure 2).

### 4.4.1. The data structure of the directory area

The following structure is called a *Path-name entry*, which is stored in the directory area.

```
#define DIRSIZ 14

typedef ushort dir_t;           /* directory entry slot number */

struct hpf_direct {
        ino_t d_ino;            /* i_number for this entry */
        char  d_name[DIRSIZ];   /* path-name for this entry */
        dir_t d_parent;         /* parent dirent slot number */
        dir_t d_child;          /* child dirent slot number */
        dir_t d_elder;          /* elder dirent slot number */
        dir_t d_younger;        /* younger dirent slot number */
};
```

Even though the directory "." and ".." don't exist in the *directory area*, the link count in i-node tables are modified as if there were "." and ".." entries in the directory files.

### 4.4.2. The management of free path-name entries

Free path-name entries are managed by a chain structure. In the directory area, there is an array of the above structures, and the first element of this array points the first free entry. This chain is managed by the LIFO (Last In First Out) algorithm.

In the free entry, "d_ino"= 0 and "d_parent" points to the previous entry, and "d_child" points to the next entry.

### 4.4.3. Caching of path-name entries

Like the other information such as i-node, path-name entries must be synchronously written to the disk when it is modified, but for the performance of management, it is cached and hashed by its slot number.

### 4.4.4. The read/write mechanism of directory information

The read operation to a directory is implemented as if there is a directory file. In this implementation, the path-name look up routine need not be modified. The write operation to the directory must be implemented for HPF.

(1) In the read operation
In the read operation, if the file mode is directory, a buffer is allocated to load all directory entries which include "." and ".." entries. After this loading, the read operation invokes the data copy from this buffer to the user buffer. If the modification to the directory occurs, this buffer will be flushed and reconstructed.

(2) In the write operation
The side-effect of the path-name look up routine to the user structure must be implemented because of the performance in the modifications. In HPF, the offset entry of user structure has the slot number of the path-name entry.

### 4.4.5. The management of i-nodes

(1) The data structure of i-node
In the following structure the data structure of HPF disk i-node can be seen.

```
#define NDADDR_HP 16

struct dinode {
        /* Here are original i-node entries */

        ushort  di_grow;                /* growth size */
        daddr_t db_addr[NDADDR_HP];     /* disk block address */
        ushort  db_size[NDADDR_HP];     /* disk block size */
};
```

(2) The i-node management algorithm
The i-node management algorithm of HPF is the same as that of normal system V.

### 4.4.6. The link mechanism of HPF

Normally, path-name entry and i-node have one-to-one correspondence with the "hpf_direct" entry "d_ino", but when the file linking operation occurs, two or more path-name entries have the same i-node number in its "d_ino", and the link count in the i-node is incremented.

## 4.5. The method to speed up contiguous free space search

A fast algorithm to find the contiguous space in the disk is desired for HPF. A physical disk is divided into many fixed size blocks and these blocks are managed by the table of free blocks.

### 4.5.1. The table of free blocks

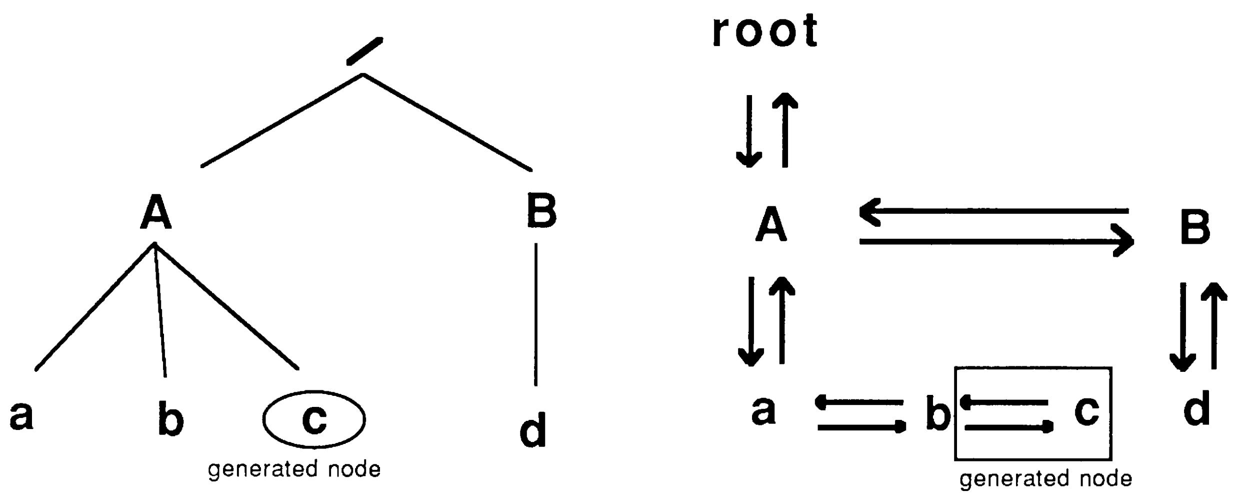
The table of free blocks is a bitmap table for the following reasons.

(1) Security against crash.

(2) Table size depends only on the total file system size.

(3) No necessity for garbage collection.

In the table of free block, there is one-to-one correspondence between a free block in the disk and a bit on the table. So, to find the contiguous free space in the disk is to pick up contiguous bits in that table.



| slot number | d_ino | d_name | d_parent | d_child | d_elder | d_younger |
|---|---|---|---|---|---|---|
| 0 | 0 | *free-list-root* | 0 | 4 | - | - |
| 1 | 2 | *root* | 0 | 2 | 0 | 0 |
| 2 | 15 | A | 1 | 3 | 0 | 5 |
| 3 | 57 | a | 2 | 0 | 0 | 6 |
| 4 | 0 | *free* | 0 | 7 | - | - |
| 5 | 32 | B | 0 | 8 | 2 | 0 |
| 6 | 777 | b | 0 | 0 | 3 | 0 |
| 7 | 0 | *free* | 4 | 9 | - | - |
| 8 | 31 | d | 5 | 0 | 0 | 0 |
| 9 | 0 | *free* | 7 | 18 | - | - |
| : | : | : | : | : | : | : |

| slot number | d_ino | d_name | d_parent | d_child | d_elder | d_younger |
|---|---|---|---|---|---|---|
| 0 | 0 | *free-list-root* | 0 | 7 | - | - |
| 1 | 2 | *root* | 0 | 2 | 0 | 0 |
| 2 | 15 | A | 1 | 3 | 0 | 5 |
| 3 | 57 | a | 2 | 0 | 0 | 6 |
| 4 | 89 | c | 0 | 0 | 6 | 0 |
| 5 | 32 | B | 0 | 8 | 2 | 0 |
| 6 | 777 | b | 0 | 0 | 3 | 0 |
| 7 | 0 | *free* | 0 | 9 | - | - |
| 8 | 31 | d | 5 | 0 | 0 | 0 |
| 9 | 0 | *free* | 7 | 18 | - | - |
| : | : | : | : | : | : | : |

**Figure 2**: *Path-name entries*

### 4.5.2. The free space management algorithm

The construction of the table is hierarchical. To allocate a larger file, a bigger bitmap is used to find a suitable free space. With the flat bitmap, if the requested size is small, the possibility to find the free space quickly is high, but when the requested size is large, it becomes difficult to find the suitable free space in a short time. This hierarchical bitmap works well in both cases (see figure 3).

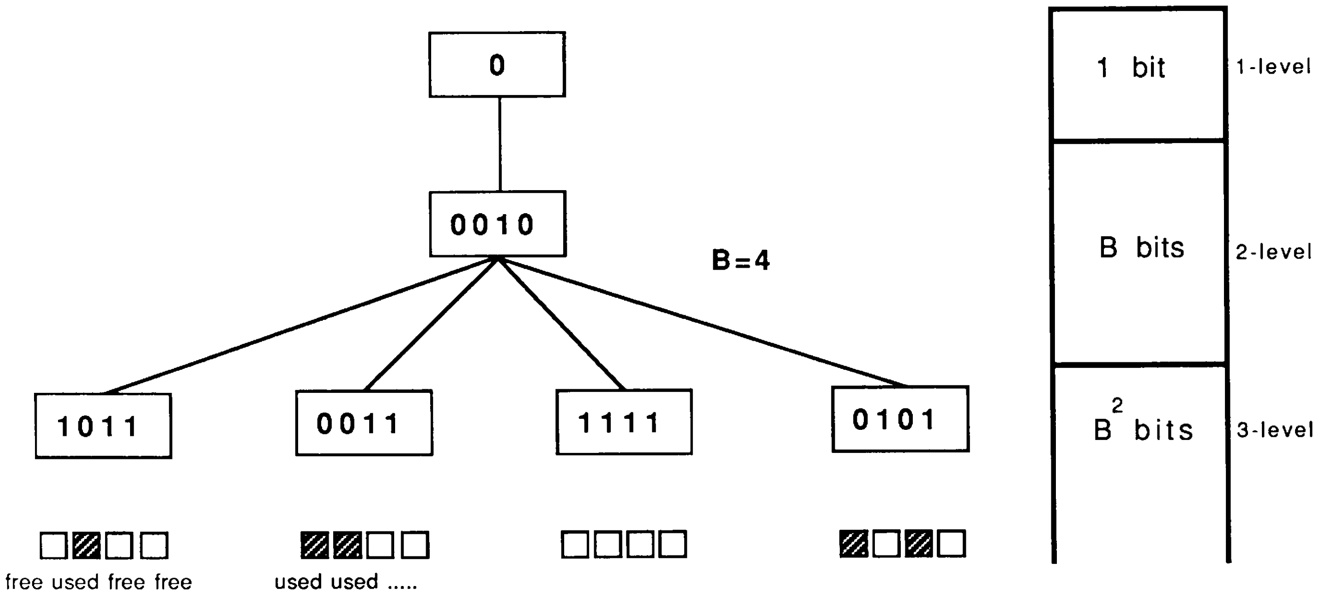


**Figure 3**: *Hierarchical Bitmap*

In Figure 3, the hierarchical bitmap having three levels and a B=4 bulk size can be seen. The lowest level bit has one-to-one correspondence with a disk block, and a higher level OFF bit expresses that the bulk of the next level has one or more OFF bits.

(An OFF bit expresses that the corresponding disk block, or bulk of disk blocks, isn't free.)

**The algorithm**

Let's define the requested size SIZE and requested allocation address ADDR. For simplicity, it is assumed that the bulk size B=2 and the total size of the file system is N-th of 2 blocks. In this case, the free contiguous space will be allocated by the following sequence.

(1) To find the minimum value "n" which satisfies the next formula.

$$2^n \geq SIZE$$

(2) To calculate the base address of the n-th level bitmap table from the next formula.

$$(2^{(N+1-n)} - 1 = 1+2+2^2+ \cdots +2^{(N-n)})$$

(3) To search for the ON-bit in the bitmap of this level from the

$$(2^{(N+1-n)} - 1) + \frac{ADDR}{2^n}$$

position. If there is no ON-bit in this level, the next lower level of the bitmap and must then be searched. This sequence is continued recursively until a contiguous space is found. We need not to go down more if there is no ON-bit in (n + 2)-level, because, in this case, there is no possibility of finding requested space in a level lower than this level.

(4) To calculate the disk address from a found bit location.

(5) To make all the bits OFF, corresponding to the just allocated blocks.

### 4.5.3. The simulation

To obtain an optimum algorithm, the following four algorithms are simulated and evaluated from the performance viewpoint. Because outer disk fragmentation must be avoided, these algorithms are also evaluated from the disk usage viewpoint.

**Algorithm 1:** Flat bitmap management

**Algorithm 2:** Flat bitmap management (with the current search position information)

**Algorithm 3:** Hierarchical bitmap management

**Algorithm 4:** Hierarchical bitmap management (with the current search position information)

Here, the current search position means the position next to the just allocated space.

**The simulation program**

The simulation program is written in C, and runs on a Sun4/280 workstation. This program is constructed with following parts.

(1) Generation of the file allocation and deletion events
This part generates the file allocation and deletion events at random, but we can control the ratio of deletion events to allocation events. In addition to this, the frequency of deletion events increases automatically when the total allocation file size becomes nearly as large as the total file system size. There are more two parameters in this part. The minimum size and maximum size of the allocated file and the total file system size can be controlled.

(2) Allocation and deletion of the file
The implementation of each algorithm of file allocation and deletion using common routines is here (see 3).

(3) The common routines of bitmap manipulation

* The bits allocation routine of specified offset and size
* The bits deletion routine of specified offset and size
* The bits inquiring routine of specified offset and size

**The result from the simulation**

The result of the simulation in a typical case can be summarized as follows. In this case, the total size is 500 Mbytes, the minimum allocation is 100 Kbytes, and the maximum size is 1 Mbytes. (Here, the block size of 1Kbytes is assumed.) The result of the simulation in this case can be seen in **Figure 4**.

This result indicates the third algorithm is best from performance and disk usage (i.e., frequency of file allocation error.) That is, it is recognized that the hierarchical bitmap management (**without** current search position information) is the best for applications from both view points of performance and disk usage. Only the result of this simulation in this special case is reported, but it is concluded that the larger the total size and each file size, the more effective this algorithm becomes.

## 5. Conclusion

In the application area of UNIX, the size of file and file system are increasing rapidly. According to these circumstances, we must increase file I/O performance without any change in the application interfaces. In this paper, we suggest that the HPF file access method will do well in this environment.

HPF is a file system adapted to the UNIX in order to support physically contiguous files for the high speed I/O bandwidth. We mainly discuss on the two following points in order to increase performance.

- The directory search
- The contiguous free space search

On the first point, we conclude that it is possible to implement the directory file-less UNIX file system without any change to the UNIX file access semantics. In HPF, it takes one or less to seek disk in the directory search because directory information is concentrated in the disk and almost all necessary information cached in memory. So, the directory path search time can be greatly decreased.

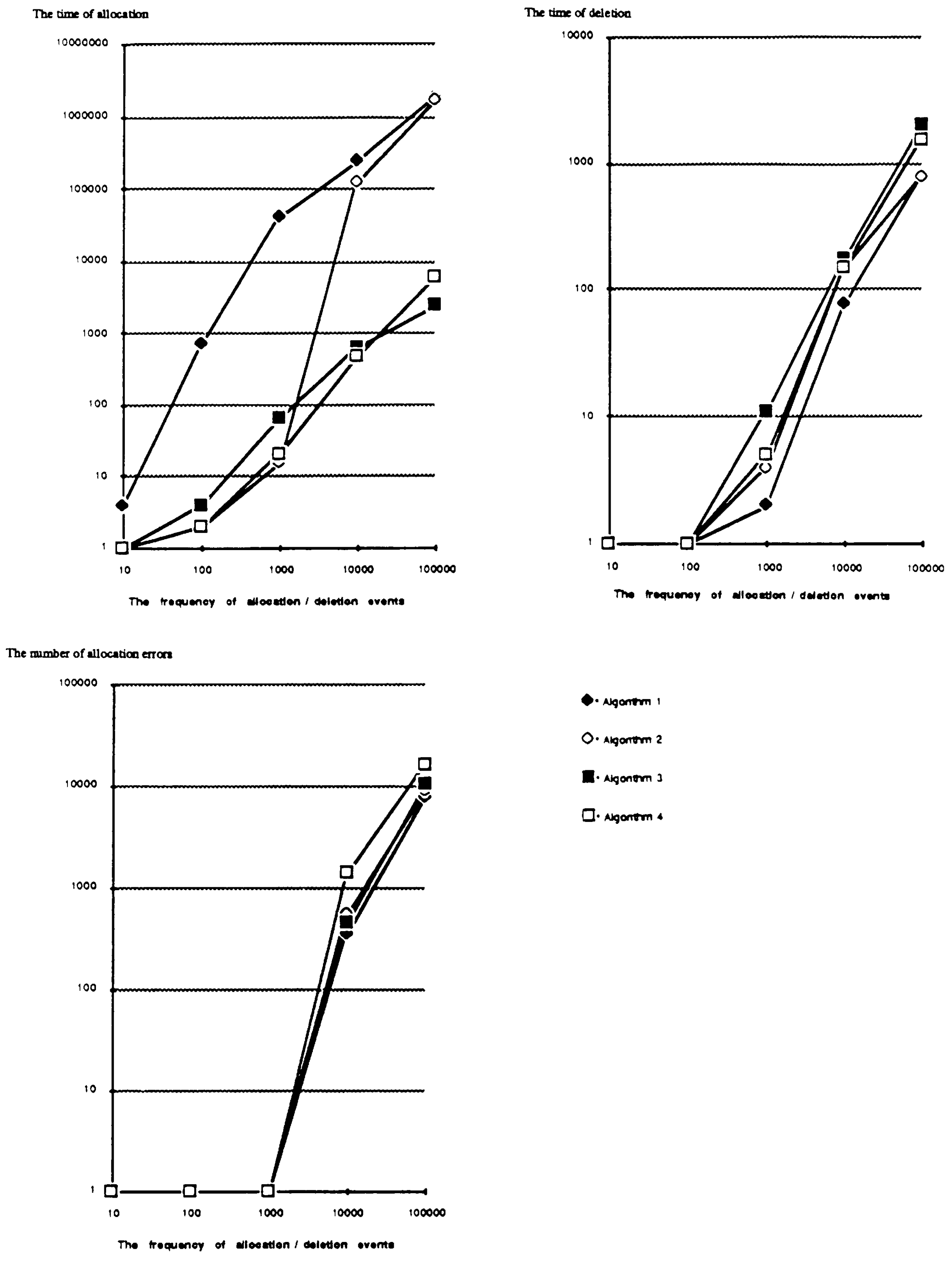


**Figure 4**: *The results of the simulation*

About the second point, we conclude that with hierarchical bitmap management, it is possible to allocate contiguous free space more than 1000 times faster than flat bitmap free space management.

From this discussion, we are ready to actually implement HPF in UNIX.

## Acknowledgements

We wish to express our gratitude for the guidance and encouragement received from Mr. Yoshiaki Tanaka, Mr. Norio Tokuda and Mr. Takashi Katsukawa.

## References

[1] Bach, M.J., *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[2] M. J. Folk and B. Zoellick, *File Structures: A Conceptual Toolkit* Addison-Wesley, 1987.

[3] J. Gait, "The Optical File Cabinet" in *Computer*, June 1988, pp. 11-22.

[4] J. S Quarterman, A. Silberschatz, and J. L. Peterson, "4.2BSD and 4.3BSD as Examples of the UNIX System" in *Computing Surveys*, Dec. 1985, pp. 379-418.

[5] B. Zoellick, "CD-ROM software development." in *Byte*, May 1986, pp. 173-188.

# Dp: a System for Inter-Program Communication

*Robin Faichney*

Computing Laboratory
The University
Canterbury, Kent CT2 7NF
England
*rjf@ukc.ac.uk*

*ABSTRACT*

**Dp** is a system designed to facilitate communication between application programs. Originally intended to prototype a partial solution to the problem of the interconnection of highly interactive software modules, dp constitutes a high level interface to the standard UNIX Inter-Process Communications (IPC) facilities. The means by which one application may address another using dp are much simpler and more flexible than standard IPC. Integers, character strings and user-defined data may be communicated, in addition to byte streams. Dp was designed for the graphical workstation environment, and is as network transparent as the windowing systems alongside which it is used. Experience in building applications with dp has lead to additions and modifications to its facilities. The design and implementation of applications in which communications play a substantial part, require considerably less effort using dp than would otherwise be the case.

## Introduction

This paper describes the first fruits of an attempt to tackle a problem which was identified during work on developing software tools for what was the UK Science and Engineering Research Council's Common Base and is now EASE (Engineering Application Support Environments), for which Kent is the Software Tools Centre.

The problem was concerned with the interconnection of highly interactive software modules, and in particular the possibility of carrying over the benefits of UNIX command line interconnection (input/output redirection) into the WIMPS (Windows Icons Mouse Pointer (or Pull-down menus) System) environment. An analysis was made of the potential purposes and methods of interconnection and the result of this is described in the first section below. It was decided that interconnection could be divided into a number of more-or-less independent categories, and that these could and should be tackled separately. The next section of the paper describes the work on the first of these categories, which culminated in the implementation of a "development package", and the following one describes some experience in using the package. Related work is then mentioned, followed by a short summary and some acknowledgements.

## 1. An Analysis of Interconnection

As user-interfaces in general become more sophisticated (the graphical user-interface being just one aspect of this), a widespread consensus of opinion has arisen that the user-interface should be separated from the rest of the application (the "functionality"). Whether such separation is viewed as entirely desirable or even possible, the design of any system intended for use by a variety of applications must take such trends into account. This has an important implication for interconnection: a decision has to be taken as to whether an interconnection channel should connect to the user-interface or the functionality. Logically, there are four permutations of interconnection:

(1) between the user-interfaces of two or more applications;

(2) between the user-interface and the functionality of one application;

(3) between the user-interface and the functionality of different applications;

(4) between the functionalities of two or more applications.

Different interconnection purposes would seem to require different interconnection channel routing. (For simplicity, and assuming that the principles will remain the same, we consider here only one-to-one connections.)

## 1.1. Between User-Interfaces

Type (1) is required where cooperation between the user-interfaces of different applications is desirable. Typically, this will be wherever the sharing of user-interface resources is required. The state of the art at the moment and in the foreseeable future is that in the context in which we are interested (the graphics workstation), this is managed by the windowing system. For instance, in the case of the X Window System [13] and some others, [1, 12] all applications using one display communicate with a server program which mediates between their user-interface resource demands.

## 1.2. User-Interface and Functionality, Same Application

Type (2) is simply the usual user-interface/functionality traffic within any application – though that does not mean that for the purposes of this project we are not interested in it. Progress with this might facilitate, for instance, the sharing of application functionality modules between different applications, thus reducing functional duplication within a system.

## 1.3. User-Interface and Functionality, Different Applications

Type (3) seems most suited to implementing the simulation of the user of one application, by another. The means of reaching this conclusion was primarily an abstract analysis of user-interface and application functionality.

As user-interface and functionality modules are themselves simply abstractions from the general concept of an application program, it might seem at first to make no sense to connect the user-interface of one program to the functionality of another. However, if one of these programs is designed specifically to simulate either part of another program or the user, then this arrangement could be used, for example, for testing purposes.

The purpose of the user-interface is to provide a communications channel between user and application functionality. Looked at in this way, user and functionality are each both a source and a sink of information which is conveyed by the user-interface. (See Figure 1.)

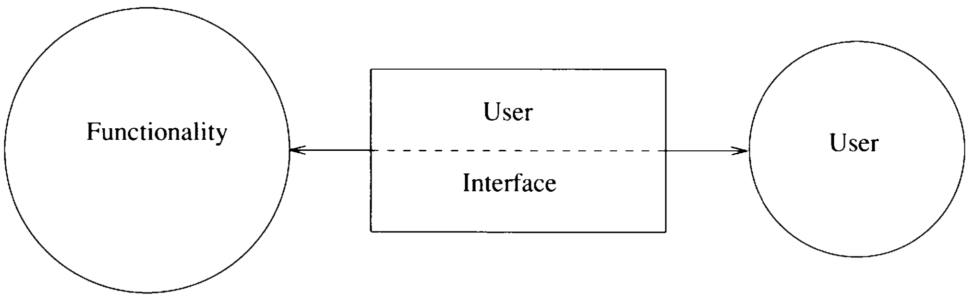


**Figure 1:** *Information flow, standard case*

This arrangement – sink/source-channel-sink/source – is the simplest possible and cannot be varied except to be made more complex, involving more modules. For present purposes, such possibilities are ignored.

Given the simplicity of the arrangement, a sink/source cannot simulate a channel nor vice versa. So the alternatives are to have the user-interface simulate another user-interface, or the functionality simulate either another functionality or the user. Interchangability of connections between interface and functionality would seem to require a type (2) solution, as it depends on some standardisation of communications between functionalities and "their own" interfaces. Further consideration of such usages must therefore be postponed until such a solution is available. Of these alternatives, only simulation of the user does not require such interchangability. (See Figure 2.) Though in principle this involves more modules and is more complex than the sink/source-channel-sink/source arrangement, in practice the user and user-interface of the simulator are not strictly interconnection-related; in fact, the whole of the

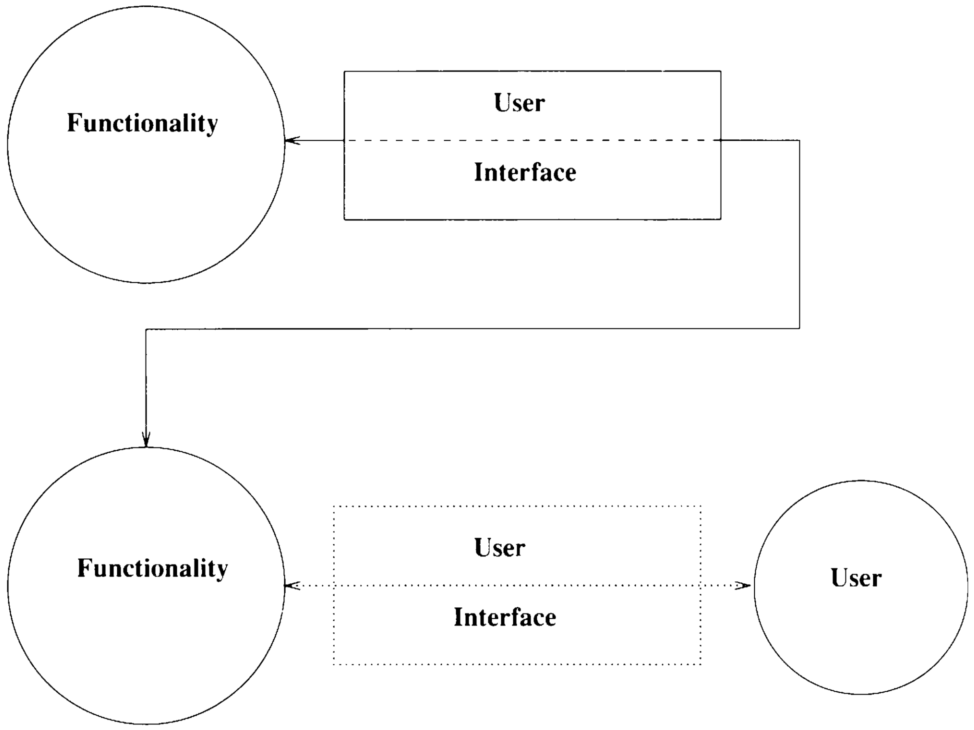


**Figure 2:** *Information flow, user simulation*

simulator may be considered a simple sink/source.

Simulation of the user is not a new concept. It occurs, for instance, whenever a transcript of user-actions is replayed. What might be developed, however, is a standardised way of doing this, which could be easily utilised by any application. It may well be found that different methods are applicable in different contexts. It may also be the case that using another program to simulate the user is not the best method in any context. Research into these and related issues is continuing.

### 1.4. Between Functionalities

Type (4) is relatively simple, though very broad in scope. It covers all cases where an exchange of data is required between programs. This might be a cut-and-paste operation (though there is some controversy about this, as it may be viewed as simulation of the user), text being distributed by a conferencing system (see "Applications", below), graphics data being sent from an interactive generator to some kind of specialist display program, perhaps on a special terminal, etc..

This type of interconnection, being straightforward compared with the others, was chosen as the first to be tackled in depth. It was expected that some of the experience gained here, in particular on the lower levels of interconnection which all four areas have in common, would be useful later, and this has already proved to be the case.

## 2. The Data Pack

The main aim of **dp** is to facilitate simple data interchange between application programs.

The first stage was to work out just what facilities were required. This was done partly by looking at existing IPC facilities, partly by talking to potential users, and partly by speculation. For instance, from UNIX BSD 4.2 IPC were taken the concepts of continuous connections and discrete messaging (stream and datagram sockets), both of which are provided. A colleague suggested a feature whereby a transmitter need not specify the receiver, but could request that something be sent to any application which has made a special request. This is so that an end-user, instead of having to provide an application name, could merely click on a light button at each correspondent. It also seemed desirable to make dp as network-transparent as possible, in order that it be as flexible as the windowing systems alongside which it would be used. The other facilities were initially determined in similar ways.

Having decided on the main features, the next stage was the detailed design of the programmer's interface to dp. A document in the form of a programmer's manual was produced, and this was the main design document throughout.

## 2.1. The Programmer's Interface

From the programmer's point of view, there are two main data structures: a type which identifies an application using three character strings for application name, host name and user name respectively; and a variable which holds dp state information. An application first must call a function which initialises dp, and this function returns a file descriptor. Dp is event driven, and a particular dp function interprets any dp input as an event whose description is put in the state variable. It is the responsibility of the programmer to ensure that if any activity on the file descriptor occurs, this function is called. One member of the state structure indicates which particular dp event has occurred, another contains the data sent, if any, another identifies the sender, and so on. Most event types signify either the arrival of data (character strings, integers and user-defined as well as byte-stream), or one of a variety of types of acknowledgement.

The distinction between discrete and connected communications, mentioned above, is important in dp. A discrete communication may be sent to any other dp application using just one function call. Addressing is by specification of the name, host and user of the correspondent. The alternative is to use a connection, which requires at least two function calls: one to request the connection, and one to actually send the communication. One justification of the extra activity for the connection is that, once established, communications on a connection are significantly more efficient than discrete ones. So connections are preferable where a substantial volume of traffic is expected. Also, the addressing requirements are fewer, and any communication on a connection is guaranteed to go only to the appropriate correspondent. On the other hand, the potential ambiguities in discrete addressing are sometimes desirable (and can in fact be increased – see below). Failures on connections always result in negative acknowledgements, whereas this is not guaranteed in the case of discrete messaging.

Discrete communications may be addressed using any combination of correspondent name, host and user, including none. Where more than one currently executing dp program would match the pattern given, each will receive the communication. However, non-specification is not always a "wild card": in most cases, unspecified program and user names are interpreted as other instances of this program and this user, respectively. "Broadcasting" to all hosts is possible with most functions, and to all users, with a particular enquiry function. Network transparency is thus maximised, while broadcasting is otherwise limited in order to minimise the load both on dp and on individual programs, which should not often have to deal with dp events not specifically intended for them. (Such events may simply be ignored in many cases, but where it is clear that certain event types will not be appropriate for a particular program, it may register them as "uninteresting", and will not thereafter be notified of them.)

## 2.2. Structure

The underlying structure of dp is a client-server model. Each application is a client of the dp server, which acts like a telephone exchange: there are "hard" connections between client and server, using which the server provides "soft" connections between clients when requested to do so, for as long as required (for discrete messaging, a rough analogy is with packet-switching systems). One client, however, may have many concurrent connections. There is just one server per host: where communication between clients on different hosts is required, it is relayed through both servers.

Although the functionality of dp includes both discrete messaging and connections, for both local and remote communication, the implementation uses connections between clients and servers, and discrete messages between servers, so local discrete messaging and remote connections are simulated. This arrangement evolved over a period of time, and there are a number of reasons for it. The first arrangement considered was the same locally, but utilised a single network server through which all inter-host traffic would be routed. Each local server would open a connection to the network server when it came up, as do clients to local servers. So the local servers would be the clients of the network server. The rationale for this arrangement was simply that for using a client/server model generally: reduction of duplication of functionality by having a single, specialist module provide services to a number of clients.

From a practical point of view, however, there were a number of problems with this, of which the two main ones probably were the number of IPC jumps for each remote operation (client-local server-network server-local server-client) and the burden of work which the network server might have to undertake on a net where dp was being widely used. One of the major differences between IPC connections (streams) and discrete messaging (datagrams) is that the former are normally client/server while the latter are usually between peers. (Unlike dp, in which both types are between peers.) As discrete messaging can simulate connections, local servers can fulfil all requirements by communicating with each other direct.† The provision of IPC discrete messaging is significantly simpler than for connections, so there is less of the duplication of functionality which the network server was intended to obviate.

## 3. Applications

In order to test dp – at design as well as implementation levels – a number of applications were built using it.

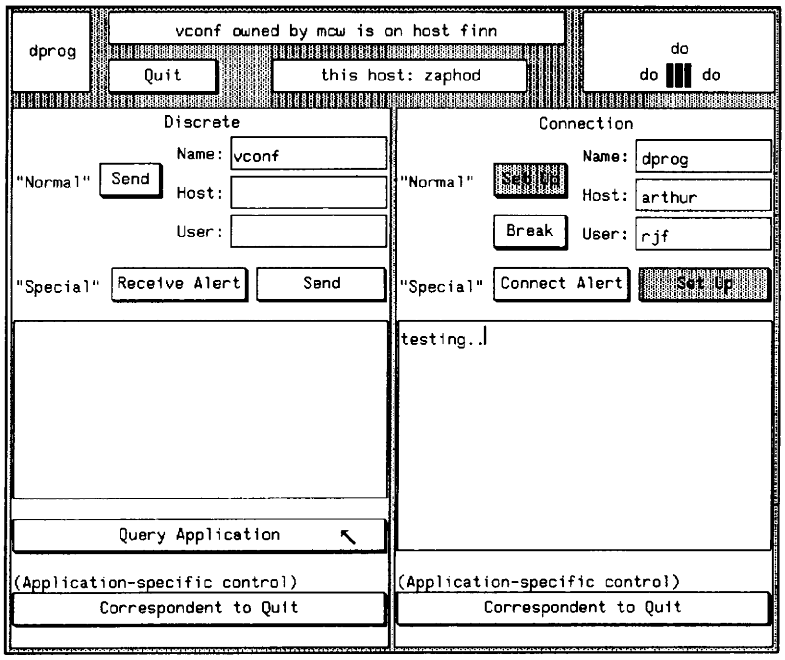
**Dprog** was the first, designed as a demonstration of dp facilities, and so consists of a "thin-layer" graphical user-interface to dp itself. It has proven to be a good tool for educating potential users of dp.

**Vconf** is a visual conferencing program.

**Dpvf** is the Kent Software Tools visual file browser **vf** [4] with a dp interface built on. **Dpi** is a standard output/dp interface. Used together, dpvf and dpi allow the monitoring of multiple output streams across a network.

### 3.1. Dprog

This is the dp demonstration program; all of dp's main features are represented, excepting user-defined and byte-stream data.



vconf owned by mcw is on host finn

this host: zaphod

Discrete: Name: vconf, Host: , User:

Connection: Name: dprog, Host: arthur, User: rjf

testing..

(Application-specific control)

(Application-specific control)

**Figure 3:** *Dprog*

Dprog has a graphical user-interface, and puts up a window which consists of two main areas, for discrete messaging and connections. (See Figure 3.) Using *discrete* messaging, dprog can

> send text to any other dp program, specified as described above (see "The Programmer's Interface");
>
> send text to an unspecified dp program, possibly on another machine, owned by the same user, which has requested a "discrete alert" status;
>
> itself request such a status;
>
> query the existence of any dp program, specified as above (except that this uses the enquiry function mentioned above ("The Programmer's Interface") so that an unspecified user means "any user" instead of "this user");

† Strictly speaking, the simulation of connections is not perfect, at present at least, and dp connections between different machines are less reliable than their IPC counterparts; though this has caused no problems in practice.

if the correspondent specified is another instance of dprog (or another program so designed), make it quit.

Regarding *connections*, dprog can

request that a connection be established with a correspondent specified as for discrete messaging (strictly, of course, this request is a discrete operation);

request a connection with any program owned by this user which has requested a "connection alert" status;

itself request such a status;

break a connection;

send/receive text on a connection (actually, any change in a certain editable text area will automatically occur also at the correspondent);

if the connected correspondent is another instance of dprog (or another program so designed), make it quit.

Dprog is perhaps particularly useful with a number of very simple, non-graphical example programs which have been designed for tutorial purposes: these programs print out appropriate messages when dprog enquires about them, sends messages, connects to them, etc..

## 3.2. Vconf

Here the aim was to provide a means of communication between workstation users on the local network, using graphical user-interaction, and generally making the user-interface as simple as possible. (See Figure 4.) By default all local users participate in one conference, but each may select a subset for sending to, and "private" messages – sent to one correspondent only – are labelled as such at the receiver.

The topology of vconf is a fully interconnected network, each instance being connected to every other instance.

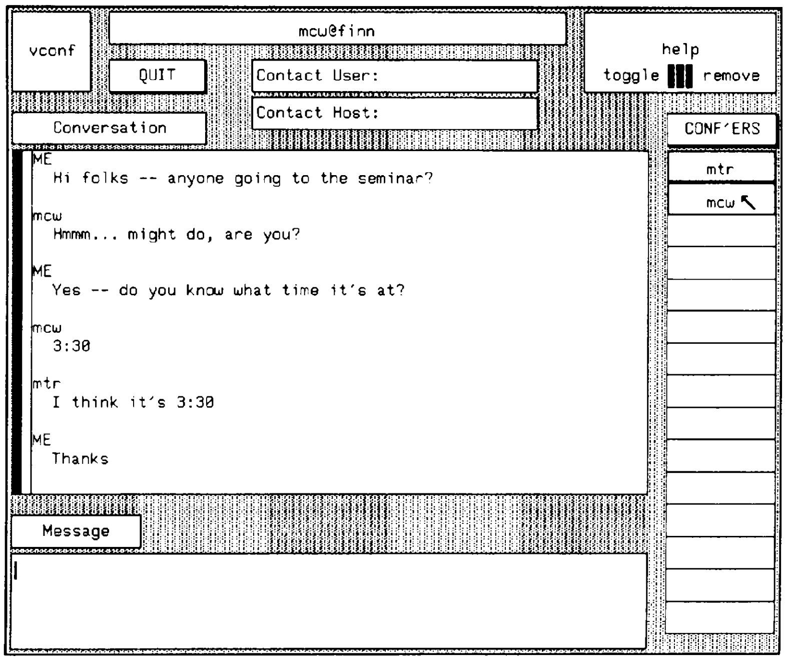
When an instance of vconf starts up, it calls the dp query function already mentioned to ascertain the existence of any other instances on the network. When one instance is informed that another is enquiring, the enquiree requests to be connected to the enquirer. So the enquiry function is being used here merely to notify existing instances about the new one. (It has an option, not used here, whereby the enquiree is not notified of the enquiry.) This function is an example of a feature inspired by actual experience, as it was designed and implemented in the first instance specifically for vconf.

Following the connection request from the enquiree, both will receive notification from dp that a connection has been established. Each then enters the details of the other in it's own list of correspondents. From this point, any message sent by either instance will be received and displayed by the other. Also, if an instance already has other correspondents, it will send a list of them to the new one, which then attempts to contact each of them (unless already connected to it, of course). This feature is intended to strengthen the bias towards there being just one conference. Vconf distinguishes between these lists, ordinary messages to be displayed and "private" messages by making them different types of user-defined data.

Thanks to dp, the communication facilities of vconf are very simple: one function call to make contact with other instances and one to request a connection, then another to send each message to each correspondent. "Broadcasting" (leaving one or more correspondent identification fields blank – see "The Programmer's Interface") discrete messages would require only one function call for a message to be sent to all other instances. (Vconf uses broadcasting with the enquiry function.) As it relies simply on ambiguity of addressing, however, it is rather a blunt weapon, and is too inefficient for other than occasional use. The ability to define groups of correspondents, and send to a group using just one function call, is a feature which may be added to dp in a future version.

## 3.3. Dpvf and dpi

Vf is a tool for browsing many types of files. It allows smooth scrolling and context searching as appropriate on text, data, object, archive, directory and various graphics formats, initially guessing the format and displaying accordingly. It may be given one or more file names, in which case it will display the first and the user may choose to view any other via a pop-up menu. Alternatively, it will read and display it's standard input, using a temporary file. Vf may be called from **fs**, [7] a file system editor, and has an interface whereby it can receive a list of files from fs. On it's own, vf is very often used to display UNIX manual pages, via standard input.



```
ME
  Hi folks -- anyone going to the seminar?

mcw
  Hmmm... might do, are you?

ME
  Yes -- do you know what time it's at?

mcw
  3:30

mtr
  I think it's 3:30

ME
  Thanks
```

**Figure 4:** *Vconf*

A dp-vf interface was built — dpvf — which allows the monitoring of multiple "live" streams. (See Figure 5.) This is quite simple, and was put together in a relatively short period. When a dp connection is made to dpvf, it opens a temporary file, adds the file name to the current list, and passes the list to vf using the fs-vf interface, first executing vf if necessary. Thereafter any data received on a connection will be appended to the appropriate file, and vf told to display it, if not already doing so.

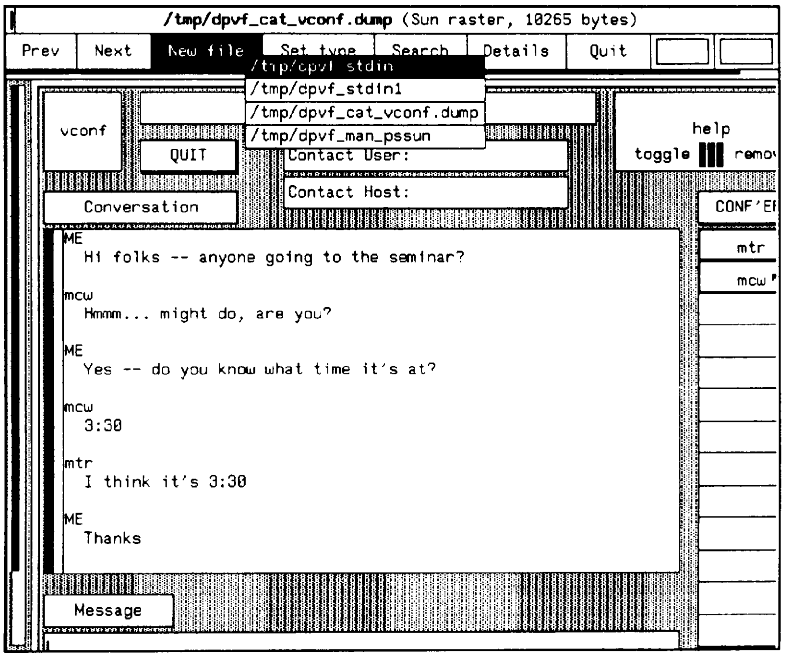
In principle any dp program might connect to dpvf and send data for display, but so far it has only been used with dpi, a standard input/output-dp interface. Dpi is either given a command of which the standard output is to be displayed, or will read it's standard input. In either case it attempts to connect to an instance of dpvf and if successful, will send on any data it receives. (Command line options inform it that dpvf is either on "this host", a named host, or "any host", and dpvf's and dpi's users must be the same.) If a command is given to dpi, it will pass the name to dpvf so that the temporary file can be named appropriately (Figure 5). The dpvf/dpi combination is superior to vf wherever more than one standard output stream is to be displayed, and/or a stream on another host needs to be viewed where either a network-transparent windowing system is not being used or the remote machine does not support it. Examples are the display of two or more manual pages, or debugging output from a number of separate processes.

This combination of dpvf and dpi is sub-optimal in a number of ways, which need not be gone into here. The main problem, however, is the fs-vf interface. Ideally, vf should have dp capabilities built-in. These could be used not only to provide what is presently the dpvf functionality, but also to supercede the fs-vf interface. One instance of vf could then fulfil many purposes, reducing screen clutter and functional redundancy. Vf may be rewritten to thus use dp at some future time.

## 4. Related Work

A number of other systems which provide interconnection facilities have been described.

Switchboard [15] and Sassafras [10] concentrate on concurrency, communication and synchronisation within a User Interface Management System (UIMS). Upconn, [6] Charlotte, [3] the hierarchical process-composition model [11] and the Poker Parallel Programming Environment [14] are mainly concerned with distributed applications, and in none of them can connections be reconfigured.



```
ME
  Hi folks -- anyone going to the seminar?
mcw
  Hmmm... might do, are you?
ME
  Yes -- do you know what time it's at?
mcw
  3:30
mtr
  I think it's 3:30
ME
  Thanks
```

**Figure 5:** *Dpvf*

ConMan [9] is much closer to the concerns of this project, being concerned with the modularisation of graphics-interfaced applications and the dynamic reconfiguration of their interconnections under the control of the end-user. However, it is specifically oriented towards graphics applications, and cannot provide inter-host connections.

Dp provides a similar functionality, in general terms, to that of the switchboard in DEMOS, [5] but differs in it's implementation, being constrained by the decisions to aim specifically at the graphics workstation environment, and not to modify the operating system kernel. Also, though full details were not available at the time of writing, the programmer's interface of dp seems to be higher level than that of DEMOS, and dp consequently should be easier to use, if not quite as powerful. Similarly, the functionality of dp overlaps considerably with that of ANSA, [2] but dp is much narrower in scope, and generally less powerful, while being much easier to use. One possibility for future consideration would be a version of dp built upon ANSA, thereby combining some of the benefits of each.

These systems will be further studied (as will the work of Cockton [8] — though not explicitly concerned with interconnection nevertheless highly relevant) in the context of the next stage of the project: interconnection between the user-interface of one application and the functionality of another. (See "An Analysis of Interconnection".)

## 5. Summary

As part of a research project concerning application interconnection, a development package ("dp") has been designed and implemented which provides a high level interface to system Inter-Process Communications (IPC) mechanisms, and incorporates additional facilities. One dp client may address any other(s) across a network by any combination of application name, user name and host name. As well as byte-stream data, integers, character strings and user-defined data may be sent. A number of applications have been implemented using dp, resulting in design modifications as well as facilitating debugging. A package comprising dp and the applications is currently available as a commercial product.

## 6. Acknowledgements

The work described here is supported by SERC Research Grant GR/D/80612. P. J. Brown must take much of the credit for the original concept, and he and other members of the Software Tools Group, Computing Laboratory, University of Kent at Canterbury, in particular David Barnes, have subsequently made substantial contributions.

## References

1. *NeWS Manual,* Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California, 19 December 1986.
2. *ANSA Reference Manual Release 01.00,* Architecture Projects Management Limited, Cambridge, England, March 1989.
3. Y. Artsy, H. Chang, and R. Finkel, "Interprocess Communication in Charlotte," *IEEE Software*, pp. 22-28, January 1987.
4. David Barnes, Mark Russell, and Mark Wheadon, "Developing and Adapting UNIX Tools for Workstations," in *Proceedings of the EUUG Autumn 1988 Conference* , pp. 321-333, October 1988.
5. Forest Baskett, John H. Howard, and John T. Montague, "Task Communication in DEMOS," in *Proceedings of the Sixth Symposium on Operating Systems Principles*, pp. 23-31, ACM, New York, 1977.
6. Mitali Bhattacharyya, David Cohrs, and Barton Miller, "A Visual Process Connector for Unix," *IEEE Software*, pp. 43-50, July 1988.
7. J.D. Bovey, M.T. Russell, and O. Folkestadt, "Direct Manipulation Tools for UNIX Workstations," in *Proceedings of the EUUG Autumn 1988 Conference* , pp. 311-319, October 1988.
8. G. Cockton, Interaction Ergonomics, Control and Separation: Open Problems in User Interface Management Systems, Scottish HCI Centre, Heriot-Watt University, Chambers Street, Edinburgh, EH1 1HX, February 1988.
9. Paul E. Haeberli, "ConMan: A Visual Programming Language for Interactive Graphics," *Computer Graphics*, vol. 22, no. 4 (ACM SIGGRAPH '88 Conference Proceedings), pp. 103-111, August 1988.
10. Ralph D. Hill, "Supporting Concurrency, Communication, and Synchronisation in Human–Computer Interaction — The Sassafras UIMS," *ACM Transactions on Graphics*, vol. 5, no. 3, pp. 179-210, July 1986.
11. T. LeBlanc and S. Friedberg, "Hierarchical Process Composition in Distributed Operating Systems," in *Proceedings of the Fifth International Conference on Distributed Operating Systems*, pp. 26-34, CS Press, Los Alamitos, California, 1985.
12. J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, vol. 29, no. 3, pp. 184-201, March 1986.
13. Robert W. Scheifler and Jim Gettys, "The X Window System," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1986.
14. L. Snyder, "Parallel Programming and the Poker Programming Environment," *Computing*, pp. 27-36, July 1984.
15. Peter P. Tanner, Stephen A. MacKay, Darlene A. Stewart, and Marceli Wein, "A Multitasking Switchboard Approach to User Interface Management," *Computer Graphics*, vol. 20, no. 4 (ACM SIGGRAPH '86 Conference Proceedings), pp. 241-248, August 1986.

# Memory Management Hardware: Panacea or Pain?

*Aarron Gull*
*Sunil K. Das*

City University London
*aarron@cs.city.ac.uk*
*sunil@cs.city.ac.uk*

*ABSTRACT*

In recent years computer manufacturers have flooded the bottom end of the market with a diverse range of low budget hardware. Although the processing power of these machines has steadily increased, such workstations are typically devoid of the expensive memory management hardware often found on larger machines.

This paper investigates the difficulties of supporting UNIX on such machines. It draws examples from two separate re-implementations of the MINIX operating system both of which support efficient, if not secure, UNIX-like processes without the aid of memory management hardware.

## 1. UNIX Without Magic Hardware

It is often said that UNIX is the first truly portable operating system; part of this is due to the few demands it places on the hardware: split levels of execution, prioritized interrupts, I/O support and memory management hardware. Indeed UNIX is offered today by countless manufacturers on numerous hardware architectures.

UNIX's reliance on memory management hardware, however, greatly reduces its portability. Almost every computer manufacturer feels it necessary to provide machines with proprietary memory management hardware totally unlike any of those already on the market. Traditionally, UNIX systems have faced this problem by either restricting the virtual memory facilities provided or by emulating previous systems.

How can it be said that the UNIX operating system is truly portable or that it will gain universal appeal while it is still firmly shackled by its dependence on a hardware Memory Management Unit (MMU)?

This paper investigates the problems involved with running UNIX on machines such as the Atari ST or complex networks of transputers which do not have memory management hardware.

## 2. UNIX and Machines Without a MMU

The UNIX process model is classically simple; a process is created when its parent executes a *fork()* system call. The child process is originally an identical copy of the parent but the *exec()* system call may be used to overlay the program address space with different images. Process synchronisation is provided by the combination of the *exit()* and *wait()* primitives.

There are three basic difficulties with providing such a process control scheme on a machine without memory management hardware: static relocation, dynamic relocation and process protection.

### 2.1. Static Relocation

When an *exec()* system call is issued it cannot be guaranteed that the new program image will fit into the memory vacated by the previous one. Modern memory management hardware provides each program with the illusion of running on a virtual machine whose entire memory is free to be used. Such a scheme allows the machine to execute code whose virtual address need not be identical to the physical address it occupies. With virtual memory hardware, therefore, the *exec()* system call can be implemented without special compiler support.

To support UNIX processes without such hardware requires executable files of a form that enables them to be run in any available area of memory.

## 2.2. Dynamic Relocation

The mechanism of the *fork()* system call guarantees that a child process will initially be an identical copy of its parent. Virtual memory hardware maps virtual program addresses onto their corresponding physical addresses. This allows identical processes to co-exist on the same machine. Compilers can, therefore, generate addresses that are independent of any other process. Unfortunately, this is not possible on machines without memory management hardware. A different solution to dynamic relocation, therefore, has to be found.

## 2.3. Process Protection

Protecting the address space of processes is important for two reasons:

1) The address space of each UNIX process is normally protected from the accidental or malicious actions of other processes. Address space protection is predominately undertaken in hardware at the page level.

   Attempts by a process to access memory outside its address space result in a protection fault that is trapped by the hardware and vectored back to the guilty process.

2) Programs also need to be protected from themselves; coding bugs, for example, often cause a stack to overflow. Memory protection will trap such errors and allow core dumps to be produced for debugging purposes.

It should be noted that software solutions to memory protection, the only alternative when lacking hardware memory management, are often neither secure nor efficient.

## 3. Possible Software Alternatives to a MMU

Two integrated solutions are immediately obvious:

1) Cheat by discarding the UNIX process creation mechanism! Superficially, this drastic solution is trivial to implement. The *fork()* and *exec()* system calls can be simply replaced with a *forkexec()* call to create a new processes running a given filename.

   This simplicity, however, is mostly illusionary; by departing from the accepted UNIX system call interface many utilities would need to be rewritten. For complex utilities such as *make* [Feldman 1979] this is not a trivial course of action. Furthermore, a simple *forkexec()* call does not provide any form of process protection.

2) Use compilers that generate position independent code (PIC). A PIC compiler produces code in which all addressing is made relative to a base register. Such code can be made to execute in any region of memory by simply changing the contents of the base register.

   This solution, which has the advantage of being both simple and elegant, solves both the static and dynamic relocation problems. By modifying the compiler to validate memory references it is also possible to provided a semblance of process protection.

   Unfortunately the instruction set of the MC68000 makes this solution particularly expensive in terms of lost performance.

Clearly a single satisfactory solution to all three problems is not immediately obvious. A more complex solution, therefore, needs to be devised; the following solution uses three separate mechanisms to provide UNIX-like process control:

1) The relocation information that is generated and often discarded by compilers can be retained and used to implement static relocation. Relocation information consists of a several program addresses that need to be relocated before execution. Consequently relocating a binary program simply consists of adding a constant offset, the difference between the physical address of the code in memory and the virtual address it was compiled to run at, to each of these addresses.

   The format of such an executable file is shown in figure 1 below; there is no space between the header, text, data and relocation segments.

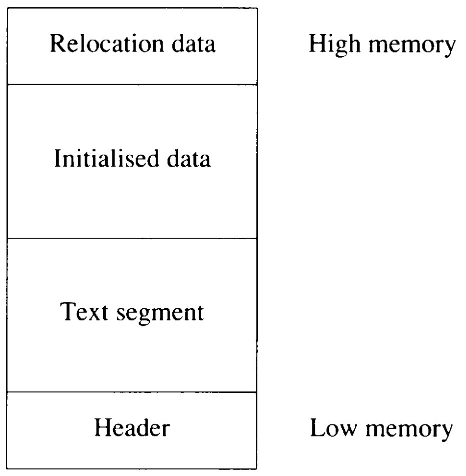


**Figure 1**: *Executable file format*

All relocation information must be validated to ensure that the address to be relocated really resides within the address space of the process. If this precautionary measure is not taken executable files that are corrupted could either accidentally or maliciously cause a system crash.

Using relocation information in this way has the advantage of not affecting the overall system performance, but does slightly increase both the size of executable files and the time taken to do the *exec()* system call.

2) Dynamic relocation can be implemented by allowing related processes to share the region of memory they were statically relocated to run at.

Consider a processes that consists of a read only text segment (TEXT) and read/write data and stack segments (collectively called DATA). When it executes a *fork()* system call a space large enough to hold a copy of its DATA, called a shadow region†, is allocated in memory. Assuming the *fork()* is successful, the child process is created and scheduled to run before the parent.

When the child begins execution, the parent's DATA is copied into the shadow region. The DATA of the parent process then becomes that of the child. This subterfuge substantially reduces the overhead of a typical *fork()* and *exec()* sequence.

Whenever either process is then scheduled, their DATA must be relocated, if necessary, from the shadow region into the memory required for execution. As a text segment is read only it may be shared between related processes.

A typical *fork()* sequence is illustrated in figure 2.

Shadowing is only possible if process sizes are fixed at creation; the *brk()* and *sbrk()* system calls must not change the size of a process, but simply validate there is room on the heap for the amount of memory requested.

The most typical action after issuing a *fork()* system call is for the child process to do an *exec()*. Once it is certain that the *exec()* can succeed, a check is made to determine whether any shadows exist for this process. If a shadow region is found the process' DATA region is swapped into this memory and then released. Otherwise, this process is known to be the last that is running the current TEXT image and therefore, both TEXT and DATA regions are released. Finally, memory is allocated for the new process image.

The typical case of a child process issuing an *exec()* is shown in figure 3.

† A term used by Tanenbaum [Tanenbaum 1988]

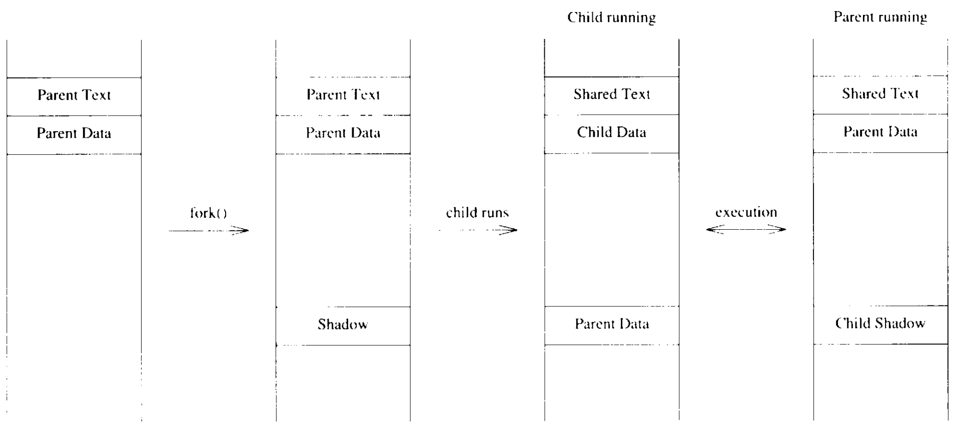


**Figure 2**: *A typical* fork() *sequence*

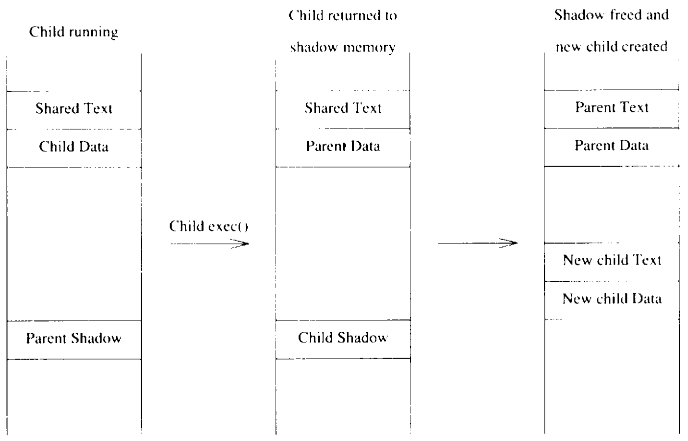


**Figure 3**: *Issuing an* exec()

3) It is difficult to provide process protection without incurring large performance overheads. It is a simple task to validate the position of each process stack pointer, for example, but to do this constantly would considerably reduce the throughput of the processor.

   It is possible to reach a compromise between constant monitoring and none at all by simply checking the stacks each time a context switch occurs. It is doubtful, however, if the extra overhead of validation is worth producing an imprecise core dump from the occasional process that acts in a strange manner.

## 4. Putting the Theory into Practice

The combination method described above was used in both MINIX-ST and STIX: two *independent* ports of the MINIX operating system [Tanenbaum 1987] to the Atari ST [Gerits 1988].

The remainder of this paper describes the contrasting implementation of MINIX-ST and STIX. The effect the differing algorithms have on the overall system performance is also compared.

### 4.1. The Implementation of MINIX-ST

MINIX-ST [Tanenbaum 1988], ported to the Atari by J Stevenson, has recently been released by Prentice-Hall. Externally, the port differs little from its IBM PC counterpart. This similarity, however, is only skin deep; major re-implementations of the kernel and memory manager have taken place. Only the file system remains largely unchanged.

The MINIX-ST process management has been implemented by prioritized scheduling queues. Four queues exist: two associated with kernel tasks and two with user processes.

Each kernel task is created at boot time and since they do not *fork()*, they are never shadowed. Consequently both kernel queues are devoted to holding the id's of tasks that are ready to run.

One user queue holds the id's of processes that are ready to run, while the other is devoted to holding id's of shadowed processes. Shadowed processes are never executed.

Every 500 milliseconds (5 clock ticks) the scheduler invokes a routine that causes the process at the top of the shadow queue to be exchanged with the one currently occupying its execution memory.

MINIX-ST provides limited stack overflow detection during each context switch.

### 4.2. The Implementation of STIX

The STIX project [Gull 88], the construction of a MC68000 cross development system and the subsequent port of MINIX to the Atari, was undertaken by the principal author at the City University, London. Although the port was conducted independently, the implementation of STIX differs only slightly from that of MINIX-ST.

The task of ensuring scheduled processes are not in shadow memory is left to the context switching code. Each time a process is to be restarted a check is made on the location of the process' DATA. If this is currently situated in shadow memory a swapping function exchanges the shadowed DATA with that currently held in the memory required for execution. By giving processes 500 millisecond time slices the amount of extra system overhead is kept to a minimum.

While swapping is taking place the processor is locked to ignore interrupts. This is necessary as the swapper is non re-entrant.

There is no attempt for STIX to provide any stack pointer validation at each context switch. This would be a waste of time since most processes which overflow their stacks, core dump through related side effects before they fully consume their time slices.

## 5. Conclusions

The MINIX-ST and STIX operating systems have shown that, contrary to general opinion, it is possible to support UNIX-like processes without memory management hardware.

It is not clear, however, of how much benefit further investigation will be; most commercial machines are used to run specific applications (which don't need memory management) as opposed to integrated development environments (which do). Unless commercial applications grow so large that virtual memory hardware will be required to support them, no matter how cheap memory management hardware becomes it will may remain a feature that is not required by the typical computer user.

Software techniques such as those described are then only of use to programmers who specifically require the type of environment UNIX provides, but who cannot afford the hardware that is really required to support it.

## 6. References

[Feldman 1979]

S. Feldman, *Make – A Program for Maintaining Computer Programs*, Software – Practice & Experience, April 1979.

[Gerits 1988]

K. Gerits L. Englisch, R. Bruckman, *Atari ST Internals – The authoritative insiders guide*, Abacus Software, 1988.

[Gull 1988]

A. Gull, S. K. Das, *STIX: A Port of the MINIX Operating System to the Atari ST*, UKUUG Winter Conference, 1988.

[Tanenbaum 1987]

A. Tanenbaum, *MINIX: A UNIX Clone with Source Code*, EUUG Newsletter, Vol 7, No 3, pp 3-11, 1987.

[Tanenbaum 1988]

A. Tanenbaum, J. Stevenson, and J. Müller, *MINIX Manual for the Atari ST*, Prentice-Hall, 1988.

# Mk'ing Hardware: A Tutorial

*Tom Killian*

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974, USA
+1 (201) 582-6215
*tom@research.att.com*

*ABSTRACT*

Suppose we wish to build a piece of hardware, using wire-wrap technology. At the end of the design process, we will have a set of "object" files, most likely a list of point-to-point wiring instructions for an automatic or semi-automatic wiring machine, and one or more files for configuring programmed array logic (PAL's). These "object" files will be generated from a plethora of 'source' files, graphic as well as textual; the details depend on what programs are available. `Mk` is the natural tool to tie these programs together, and ensure that the computer always has an accurate representation of the physical state of the board as it is debugged and modifications are made. In contrast to the primordial `make`, `mk` allows meta-rules to be defined with regular expressions, and transitive closure is part of its semantics. Using examples from the Unix Circuit Design System (UCDS), we show how these properties can be applied to construct a master `mk` "library" that allows rules for individual designs to be specified compactly. Such a library also facilitates tracking of changes in the design system.

*AT&T Bell Labs were unable to release the full text of this paper in time for publication in these proceedings. The full text will be available from Bell Labs, free on request, as a Computing Science Technical Report.*

# A Model-Based Diagnostic System for the UNIX Operating System

*Gail Anderson*

*Paul Chung*

*Robert Inder*

Artificial Intelligence Applications Institute
University of Edinburgh
*gail.anderson@ed.ac.uk*

*ABSTRACT*

The use of model-based reasoning techniques allows for the development of systems which describe the underlying structure of problems better than rule-based systems can. A model of the UNIX Operating System based on the filesystem structure is developed. Two applications of the model are described. The model is used as the basis of a program to help the user run filesystem checks on UNIX machines. It is also used to perform diagnostics, in particular to diagnose problems which occur during the boot procedure.

## 1. A Model-Based Diagnostic System for UNIX

This paper describes a program developed in partial fulfillment of the requirements of the Degree of Master of Science (Information Technology) at the University of Edinburgh – a more detailed description is to be found in [AND88]. The model used in the system is based on a user-oriented model of UNIX and the UNIX filesystem, and has two modules. The first is designed to offer advice to a system manager who has to run *fsck* – it can ascertain the importance of individual files, and suggest suitable actions to the system manager if a particular file is deleted. The second module is intended to diagnose problems which occur during a boot and cause it to fail; this module is only partially implemented. A process of incremental development was used: as the model was more clearly defined, the code was expanded or rewritten.

The system is aimed at naive system managers (for example, someone who has a Sun workstation sitting on her desk and has to act as her own system manager). It is not sophisticated enough to be of much help to an experienced system manager.

It is written in Inference ART, and was developed on a Sun 3/260 with 16 megabytes of memory. ART can run within the SunView windowing environment, and provides the programmer with graphics and windowing utilities. The user interface uses three ART windows. The main window is used to display menus, from which the user makes selections using the mouse. The other two windows are a control menu window (for resetting and quitting) and a user interface window (used to display selections made by the user or to read keyboard input).

The flavour of UNIX modelled is SunOS (version 3.2) and it is assumed that UNIX is running on a Sun 3 with two disks, configured standalone. The system could easily be adapted to represent other configurations and versions of the operating system.

The model includes representations of the filesystem structure, processes and programs, and the relationship between the hardware and UNIX devices. The system does not include any representation of the kernel (unlike [FER88]), but concentrates on the appearance of UNIX to the user. The model is tailored to apply directly to the boot process. Further assumptions are that the hardware is fully functional and that any UNIX file (or directory) can be in one of two states ("working" or "not working").

## 2. Model-Based Reasoning

Systems based on rules of the form "IF <condition> THEN <action>" have been built and used for many years ([SHO73], [SHO76], [MCD82]). However, problems can arise from the unprincipled use of rules. Often, systems which are entirely rule-based will be composed of lots of little rules each of which represents a very small part of the knowledge encapsulated in the rule set; a process of "fine tuning" will be needed to force the system into producing the correct results. Care is needed to ensure that the system can easily be maintained and extended – too often, it will be difficult to see patterns in the knowledge or to gain an overview of the information because facts and assumptions are tucked away inside rule definitions. Also, the knowledge represented will generally be superficial in nature; it will be based on observations of cause and effect within the domain of the expert system.

A better method for building expert systems involves the construction of a model of the domain [CHA84]. For example, the system under discussion is based on a model of UNIX; it is not just a set of rules about the behaviour of UNIX (for example "IF the line printer isn't working, THEN the daemon may have died."). The model is used to reason about the problem – for example, "We know that in a working UNIX system there is a line printer daemon. In our model of UNIX, if we kill the daemon, the printer doesn't work. Therefore, it is worth checking whether the daemon is running."

There are several advantages to model-based reasoning. For example, in a rule-based system, information is spread out over lots of rules. If we construct a model about which to reason, not only will the knowledge be collected together, it will be structured. This makes maintenance and extension of the system far easier. Also, if we use a set of rules on its own, there can be no guarantee that the knowledge is complete, or that the system will work for every contingency. If we base our system on a model, the results are more likely to be accurate and it will be easier to see whether the system is complete, according to a comparison between the model and the "real world". It is possible to build a larger model out of several partial models of different components. This allows individual component models to be re-used, as long as the representation of each component is based on its structure and not on its interaction with the outside world [DEK83].

Lastly, there is an argument for the use of models based on our impressions of the way in which people solve problems. We tend to build models of the world in our minds, and reason about them. The models will be more or less detailed, according to the depth of knowledge we possess about the subject and according to the difficulty of the problem. Qualitative reasoning techniques (which are not exploited in this system) were developed because, intuitively, simple problems should take less effort to solve than complex ones. In a system based on qualitative reasoning techniques, values will be tend to be qualitative rather than quantitative – much in the way that we think of something as "big" rather than, say, "4.5 metres high" when only a qualitative feel for its size is needed or when quantitative information is not available.

Further information about, and examples of the use of, model-based reasoning (including qualitative reasoning) can be found in [ART84].

## 3. Inference ART

Inference ART provides some very powerful facilities for the programmer; those of interest in this context are outlined here. A more detailed discussion is to be found in [IND88].

### 3.1. Schemata

Frames [MIN81] are provided as ART schemata. The use of schemata allows the programmer to collect together associated facts as attributes of an object; inheritance between objects and classes provides a powerful mechanism for organising information. A sample directory schema is shown in figure 1. In this example, the definition for *usr* states that it is an *instance-of* a directory, and that it is a *sub-directory-of root* and its real name is */usr*. The slot *mode* is inherited from its parent schema, *directory*.

### 3.2. Relations

ART schema relations are very powerful; in addition to relations defined explicitly, the programmer can instruct ART to define new relations as soon as a particular relation is defined (for example, in the model, any time a directory is created new relations are created to represent the new links).

```
;;; Name: usr
;;; Purpose: Represents /usr - the directory on which the usr filesystem
;;; is mounted.

(defschema usr "/usr"
    (instance-of directory)
    (real-name #L|/usr|)
    (sub-directory-of root)
    (mode ((r w x) (r - x) (r - x))))
```

**Figure 1**: *A Sample Directory Schema*

## 3.3. Rules

IF...THEN... rules are provided in ART (everything before the "=>" is the premise, and everything after it the conclusion – see figures 6 and 7 for examples).

## 3.4. Viewpoints

One of the more interesting features of ART is the Viewpoint mechanism, which is a modified implementation of ATMS ([DEK86a], [DEK86b], [DEK86c]). This mechanism allows the programmer to explore different, hypothetical situations, with the maximum of information sharing. It can be used to represent changes in the world state over time.

## 3.5. LISP Functions

All the power of Common LISP [STE84] (plus some ART defined LISP functions) is available anywhere in the system, although because of optimisation routines there are restrictions on what may be used in the premise parts of rules.

## 3.6. Object Oriented Programming

ART also provides the programmer with sophisticated facilities for object-oriented and access-oriented programming.

## 4. The Model of UNIX

The model of UNIX uses schemata, rules, viewpoints and LISP functions. It does not use the facilities for object-oriented programming, although there is scope for expansion (see section 6).

There are three schema hierarchies defined in the model. They represent physical objects (CPU board, disks etc.), logical objects (UNIX files, directories etc.) and processes (the representation of processes is minimal). Schema slots are used to represent file contents and other attributes, and schema relations are used to represent links within the filesystem and so on.

Rules are used to reason about the boot process and to generate new viewpoints – the UNIX system boot is viewed as a series of different boot stages, each represented by a different viewpoint. The machine first loads *boot*, then loads *vmunix*. It then starts *init*, runs *rc.boot*, and boots single-user (or runs */etc/rc* and boots multi-user).

## 4.1. Schemata

The top-level schema in the hierarchy for physical objects is *physical-object*. Any physical object will be either a board, a disk, a partition, a console (representing a large Sun workstation screen) or an ordinary terminal.

The top-level schema in the hierarchy for logical objects is *unix-object*; sub-types of *unix-object* are *unix-directory*, *plain-file*, *special-file*, *socket* and *symbolic-link*. Part of the logical object hierarchy is shown in figure 2. Objects can be related to each other in several different ways (see section 4.2).

The plain-file schema slot "contents" is filled by a value dependent on the type and purpose of the file – there are representations of the contents of those files which are of interest to the boot process. The system startup scripts (*/etc/rc**) are represented as a series of condition-action pairs, which are evaluated by LISP. A condition *t* states that the action should always be carried out. The Bourne shell syntax is mapped directly into LISP – shell built-ins and UNIX commands are represented by LISP functions (see section 4.3). There is also a representation of the contents of filesystem table files. The original and encoded versions of

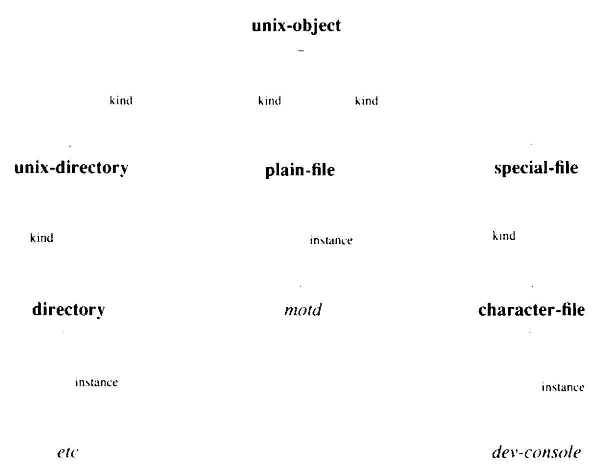


**Figure 2**: *Part of the logical-object hierarchy*

part of */etc/rc* and of */etc/fstab* are shown in figures 3 and 4.

```
if [ -f /etc/ypbind ]; then
        /etc/ypbind; (echo -n ' ypbind')       >/dev/console
fi
if [ -f /etc/in.routed ]; then
        /etc/in.routed & (echo -n ' router')   >/dev/console
fi
if [ -f /etc/biod ]; then
        /etc/biod 4; (echo -n ' biod')         >/dev/console
fi
(echo '.')                                     >/dev/console
/etc/mount -vat nfs                            >/dev/console
```

```
((file-ok 'ypbind) (and (u-ypbind) (u-echo "ypbind")))
((file-ok 'routed) (and (u-routed) (u-echo "router")))
((file-ok 'biod) (and (u-biod 4) (u-echo "biod")))
(t (u-echo "."))
(t (u-mount 'at 'nfs))
```

**Figure 3**: *Original and Encoded Versions of part of /etc/rc.local*

Schemata are also used to represent daemon processes (transient processes are not represented explicitly). An example process schema is shown in figure 5. The use of schemata to represent processes makes it possible to check for their existence, which would obviously be useful if diagnosing problems with a running UNIX system. It also allows for a deeper representation of processes – the facilities within ART for object-oriented programming could be used to represent the behaviour of running daemons.

## 4.2. Relations

Schema relations are used mainly to represent the construction of the filesystem. There is considerable use of the facility to specify inverse relations in ART (to allow for simpler and more meaningful pattern matching). Hard and symbolic links are represented by different relations. There are relations explicitly defined to state where an object is linked into the filesystem, and which disk partition it is physically stored in. Different relations state which controller a disk is attached to and which partitions are on that disk.

```
/dev/xy0a / 4.2 rw 1 1
/dev/xy1d /usr.MC68020 4.2 rw 1 2
/dev/xy1f /usr.MC68020/skye 4.2 rw 1 3
/dev/xy2g /usr.MC68020/skye/local 4.2 rw 1 3
```

```
((4.2 xy0a root)
 (4.2 xy1d usr)
 (4.2 xy0d HOSTNAME)
 (4.2 xy1f HOSTNAME-local))))
```

**Figure 4**: *Original and Encoded Versions of /etc/fstab*

```
(defschema update-process
    (instance-of process)
    (called-from-file update))
```

**Figure 5**: *Schema for update-process*

There are also relations to specify the mount-point of a filesystem and to state which physical component (if any) is represented by a special file.

A special relation, *depends-on*, is set up automatically when the model of the filesystem is loaded into ART. If a *unix-object*, a, *depends-on* an item, b, then it is necessary that b be in working order for us to access a. For example, *vmunix* depends on both / and */dev/xy0a*, and so on.

## 4.3. Rules, Viewpoints and LISP Functions

The actions of the machine during boot are represented by several rules in conjunction with ART Viewpoints. Simple stages of the boot, such the load of *vmunix*, are represented by a simple change in state; more complex steps, such as the execution of commands in the *rc* files, are represented by LISP functions called from the "action" parts of rules. Different viewpoints are used to represent different stages of the boot and different possible behaviours. An additional schema, *boot-parameters*, is used to specify certain details about the boot (for example, whether it is single- or multi-user).

Two example rules, *start-init* (which starts off init) and *run-rc.boot* (which runs sh on rc.boot) are shown in figures 6 and 7. The code for *run-rc.boot* is interesting because it contains a *for* loop to execute a set of condition-action pairs. The local variable *first* is bound to each pair in turn; then, the LISP code in the condition of first is executed. If it evaluates to *t* (holds true), the code in the action is executed. If the action fails, */etc/rc.boot* will exit abnormally, and the machine will never reach the *rc* state. If every condition-action pair is successfully evaluated, the boot will proceed to the *rc* stage.

```
;;; Name: start-init
;;; States: That init is the first process under UNIX.

(defrule start-init
    (boot-state boot-parameters vmunix)
    (loaded boot-parameters vmunix)
    (condition init 0)
    (not (and (depends-on init ?other) (condition ?other 1)))
    (condition dev 0)
    (not (and (depends-on dev ?another) (condition ?another 1)))
=>
    (sprout (modify (schema boot-parameters (boot-state init)))))
```

**Figure 6**: *A Sample Rule: start-init*

```
;;; Name: run-rc.boot
;;; States: That if we are at that state, and all is well, we
;;; should run rc.boot in a shell.

(defrule run-rc.boot
    (boot-state boot-parameters rc.boot)
    (contents rc.boot ?contents)
 ?exit <- (exit rc.boot 0)
=>
    (for first in$ ?contents do
       (if (eval (car (list*$ first))) then
          (if (null (eval (cadr (list*$ first))))
          then (and (assert (exit rc.boot 1))
             (retract ?exit)
             (bind ?result 1)))))
    (if (not (equal ?result 1)) then
       (sprout (modify (schema boot-parameters (boot-state rc))))))
```

**Figure 7**: *A Sample Rule: run-rc.boot*

LISP functions were used to represent the actions of UNIX programs. These are conventionally prefixed "u-". Some examples follow:

- *u-rm* – deletes links-to relations and ensures condition is "not working"
- *u-echo* – asserts a prints fact about its arguments
- *u-hostname* – sets hostname in boot-parameters
- *u-chown* – sets value of owner slot in a file
- *u-lpd* – creates an lpd process

## 5. Using the Model

The filesystem check help module is aimed at the naive system manager with little or no experience of running *fsck*. Other parts of the diagnostic system might recommend that the user run *fsck* with the help of the filesystem check help module; alternatively, the user can enter the module as soon as she has started running the diagnostic system.

The module explains how to run *fsck,* and guides the user through the process. It asks her to input the name of every file deleted during the check, and then ascertains the importance of each one by simulating a system boot with that file missing. It then recommends an appropriate course of action, according to the importance of the file. This helps the naive user to avoid pitfalls (an obvious example: *fsck* has deleted *vmunix;* the system then prints out "ROOT FIXED – REBOOT UNIX").

As mentioned in section 4, different viewpoints are used to represent different stages of the boot. The *fsck* help module ascertains the importance of a file by creating a new viewpoint in which that file is corrupt, and starting off a boot simulation from that viewpoint. The simulation then progresses, creating new viewpoints as necessary until the boot has either failed or is complete; see figure 8. The system then examines the state of the boot in the last viewpoint created, and determines the importance of the file according to the state. For example, if the boot state in the newest viewpoint is still *rc.boot*, it concludes that the file is important, as the naive user is likely to have trouble fixing a system which does not boot properly to single-user.

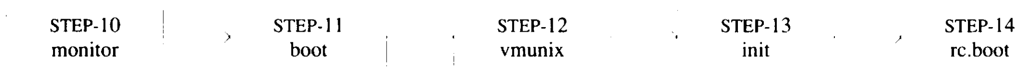


**Figure 8**: *Viewpoints Created During a Successful Boot*

Once the system has finished considering that file and made its recommendations to the user, it will prune the viewpoint tree back to the root and consider the next file in the list.

This is how a run of the *fsck* help system might look:

- welcome screen – the system prints a welcome message, and offers the user a choice between the boot failure diagnostic module and the *fsck* help module
- *fsck* welcome screen – this explains a bit about the *fsck* process
- *fsck* input screen – this gives the user a bit more information about *fsck*, and asks the user to input the names of files deleted during the check (the user can select files from a list, or enter filenames by hand if they don't appear on the list)

We will assume that the user has selected files */boot*, */usr/HOSTNAME/local* and */etc/rc.boot*.

- */boot* screen – informs the user that */boot* should be restored before a reboot, as the system will not boot at all without it
- */usr/HOSTNAME/local* screen – informs the user that */usr/HOSTNAME/local* is not essential to the boot process, and can therefore be restored after a reboot
- */etc/rc.boot* screen – suggests that the user restore rc.boot before attempting a reboot, unless he/she is competent with UNIX
- end screen – states that the system has finished its answer, and suggests that the user select *restart* or *quit* from the control window menu

The second, uncompleted, module uses different viewpoints to represent different possibilities as well as to represent different stages of the boot. It simulates a boot concurrently with questioning the user about the state of the faulty machine. In addition to asking the user about observable symptoms, it suggests tests for the user to carry out. According to the answers it is given, it eliminates certain possibilities (so pruning the search tree). The completed module would hopefully, in this way, arrive at the cause of the problem.

A run of this system might look like this:

- welcome screen
- state screen – asks the user whether the machine has been left in monitor, single-user or multi-user

We will assume that the user says that it is in single-user. If she selected multi-user, she would be told that the system does not diagnose problems with a running system. If she selected monitor, the system would help her reboot the machine (this section of the diagnostic module is complete, but is uninteresting as it uses none of the more sophisticated features of ART).

- output screen – asks the user to make an observation about the current state of the machine (for example, has the date appeared on the screen yet?)
- interactive screen – asks the user to try a command on the machine and report the result (for example, what is the output to the *mount* command?)
- diagnosis screen – makes a diagnosis (for example, *rc.boot* is corrupt) and suggests corrective action (for example, restore the file).

## 6. Room for Further Work

It is obvious that further work could include the completion of the boot failure diagnosis module; however, there is room for further extension and for improvement in the system design (which was kept simple because of project deadlines).

### 6.1. System Extension

#### 6.1.1. Automatic Generation of Filesystem Model

It should be possible to generate the schema hierarchies directly from the UNIX filesystem, by examining its structure. The program to do this could be written in C, or even in Prolog (making use of the pattern matching available), or LISP (as this is easily accessible from ART). This would make the system more generally useful, as it would eliminate the need to build the filesystem model by hand for individual configurations of UNIX.

### 6.1.2. Diagnosing Problems with a Running System

The filesystem model could be generalised to apply to a running system; there is no reason why the domain of diagnosis has to be limited to the boot process. One possible application area is the line printer software; it can be difficult to figure out why line printing has suddenly stopped working. As the diagnostic system runs on a UNIX workstation, there are particularly interesting possibilities for direct interaction between the diagnostic system and UNIX itself. For example, if there is a possibility that the line printer daemon might be dead, the diagnostic system could interrogate UNIX (by means of analysing the output from *ps*?) rather than asking the user for information. This interrogation could be carried out from a remote "diagnosis machine" by remote procedure calls.

## 6.2. Design Improvement

### 6.2.1. Use of the Facilities for Object-Oriented Programming in ART

At present, files are represented by schemata, and programs by LISP functions. There is a relationship between them, but it is buried in the code for the functions. It would be preferable to attach the LISP code to the file schemata as methods; this would not only make the relationship more explicit, it would make the model more analogous to UNIX. It would mean that programs would be executed in the model when messages were sent to the file schemata, much as they are executed in UNIX when the filenames are called. The use of methods also allows values to be returned, representing the exit status of the programs.

### 6.2.2. Better use of Viewpoints

Viewpoints are used in the filesystem check module for "time-stamping" – they are not used to consider alternatives concurrently. A better and more efficient method would be to run the simulations in parallel, sharing information, rather than running them sequentially.

## 7. Conclusion

The use of ART has certain advantages; it allows for fast prototyping, and the user interface provides the programmer with a good deal of help (there are commands to display schema hierarchies in graphical form, and so on). With careful programming, ART schemata can be used to represent knowledge in a clear and structured form. ART's disadvantages arise from its size: it is a complicated tool and as such can take some time to learn. In addition, ART applications (including the diagnostic system) require a good deal of computing power.

The model within the diagnostic system defines UNIX from a users-eye view, using some of the more complicated features of ART. Schema hierarchies are defined to represent the physical structure of the machine and the logical structure of the filesystem; there is sophisticated use of relations to represent the links within the filesystem. A minimal representation of processes is included (although this could be extended to model the behaviour of running daemons). Rules are used to control the simulation of the boot process, and LISP functions to represent the actions of UNIX commands. The model is easy to modify; although it is tailored to a particular flavour and configuration of UNIX, it could easily be adapted. It is also versatile; although only two applications are described in this paper, it lends itself to others. It is an essential part of the diagnostic system, and could form the basis of other useful advisory systems.

## References

[AND88] Gail Anderson. *A Model-Based Diagnostic System for Sun Workstations.* MSc. Dissertation, Department of Artificial Intelligence, University of Edinburgh, 1988.

[ART84] *Artificial Intelligence,* 24, 1984.

[CHA84] B. Chandrasekaran and Sanjay Mittal. Deep versus compiled knowledge approaches to diagnostic problem solving. In M. J. Coombs, editor, *Developments in Expert Systems,* 1984.

[DEK83] J. de Kleer and J. S. Brown. Assumptions and ambiguities in mechanistic mental models. In Gentner, editor, *Mental Models,* 1983

[DEK86a] Johan de Kleer. An Assumption-based TMS. *Artificial Intelligence,* 28, 1986.

[DEK86b] Johan de Kleer. Extending the ATMS. *Artificial Intelligence,* 28, 1986.

[DEK86c] Johan de Kleer. Problem Solving with the ATMS. *Artificial Intelligence,* 28, 1986.

[FER88] Conception Fernandez. Modelling UNIX with an Object Oriented Model. In *New Directions for UNIX,* EUUG Conference Proceedings, Autumn 1988.

[IND88] Robert Inder. The State of the ART. Technical Report AIAI-TR-41, A.I. Applications Institute, University of Edinburgh, 1988.

[MCD82] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence,* 19, 1982.

[MIN81] Marvin Minsky. A Framework for Representing Knowledge. In J. Haugeland, editor, *Mind Design,* 1981.

[SHO73] E. H. Shortcliffe, A. G. Axline, B. G. Buchanan, T. C. Merigan and S. N. Cohen. An artificial intelligence program to advise physicians regarding antimicrobial therapy. *Computers and Biomedical Research,* 6, 1973.

[SHO76] E. H. Shortcliffe. *Computer-based medical consultations: MYCIN.* 1976.

[STE84] Guy L. Steele Jr. *Common LISP: The Language.* 1984.

## 8. Trademarks

INFERENCE is a trademark of Inference Corporation

SunView is a registered trademark of Sun Microsystems, Inc.

# SCOOP: a Software environment for C++ Object-Oriented Programming

*Ph. Fontaine,*
*Ch. Masson,*
*L. Stefanelli*

Intecs International
Avenue Rogier, 385
B-1030 Brussels Belgium
+32 2 7355571

*ABSTRACT*

In this paper we describe a programming environment intended for those users of the C++ language who design programs according to a true object-oriented approach. First, we present the underlying principles of this approach which may be considered as a satisfying alternative to the conventional approaches. Next, we give a description of the main features of SCOOP, a tool developed by Intecs International. SCOOP allows the interactive design of the architecture and the coding of software built on top of the concepts of encapsulation and inheritance. It also provides automatic translation into Smalltalk and C++.

## 1. Why SCOOP

### 1.1. Introduction

The existence of a software crisis has been noticeable for several years, mostly due to the increasing contrast with the hardware evolution. The underlying cause to these problems lies in the intrinsic complexity of the software. This complexity is due to the fact that at every level of an informatics system, one encounters the operator-operand dichotomy, which invariably imposes dependencies (e.g. between an operator and the set of the operands that it is able to handle correctly) that are never formally expressed. In the best case these dependencies will be described, in natural language, by means of comments. Thus, the ubiquity of rules not formally specified sooner or later leads to disregard of these rules, all the more likely to occur when a large number of people are involved in the same development activity.

In the seventies, principles like modularity, localization, data abstraction and information hiding emerged to reduce this complexity. These principles have given birth to structured programming, and effectively supply the means to structure both operators and operands, but they do not allow their inter-dependency to be explicitly taken into account. From the early eighties onward, the object-oriented approach has been developing, whose major characteristics are: to implement the above mentioned principles, to take explicitly into account the dependencies between operators and operands (by installing the object structure on top of this dichotomy) and to supply a simple mechanism ensuring reusability.

### 1.2. Object-oriented Programming

One can distinguish three progressive stages leading from the conventional programming to the object-oriented programming; they are successively: encapsulation, inheritance and uniformity.

#### 1.2.1. Encapsulation

Encapsulation means a way of programming in which the dependencies between operators and operands are formally defined. Encapsulation achieves reliability. It is based on the notion of object.

- objects constitute the first structuring unit of a system, but this is a unit of architecture and not one of algorithm. Objects are referred to by names.
- objects export operations. The operations exported by an object represent the total of the actions likely to be required from it by other objects, or by itself.
- each object has a state, characterized by values of variables strictly local to the object. We have to distinguish two kinds of objects: primitive objects for which the values of the variables have a self-evident significance (e.g. a character), and others, for which these values are objects (e.g. a rectangle can be made of two points).
- the operations exported by an object are implemented by methods, which mainly are sequences of messages, solely authorized to access the state of the object.
- when a method of an object A necessitates the execution of an operation OP of an object B, object A sends a message to object B, most often textually presented as: B OP. Message sending, which is the only possibility of action for an object, and of interaction between objects, is indifferently related to the call of subprograms or to the information exchange between concurrent processes, although most current implementations essentially use the first mechanism. An operation sometimes needs parameters: B OP p1 p2 ... pn; these parameters have to be objects. Also a message may return a value, which should also take the form of an object.
- in order to structure the set of objects, a relation of equivalence is introduced among them that gives birth to the concept of class. A class represents a collection of objects that exhibit the same behaviour. More precisely, objects gathered in a given class export the same operations and have states constituted of the same number of variables.
- most languages implementing the object-oriented approach allow polymorphism, that is, variables within the state of objects may have values that are objects of any class. Polymorphism together with operation overloading (that fact that two operations of different classes may have the same name) doesn't allow any more static binding between pieces of code: binding occurs at run-time, this is called dynamic binding.
- when the language also allows dynamic creating of objects, this is called instantiation of the class, and the created objects, instances of that class.

The operator-operand dichotomy is in fact found again in the object under the method-state terminology, but what differentiates encapsulation from the conventional programming is that the whole of the operators (methods) likely to be applied to an operand (state) is encapsulated at the same time and/or in the same place as this operand: in the object. Encapsulation ensures each object is able to manage itself completely.

### 1.2.2. Inheritance

Inheritance means a way of programming in which it is the rule to reuse previously designed software when conceiving a system. Inheritance achieves reusability. It is based on the notion of derivation.

- in order to structure the set of classes, a relation of order is introduced among them that gives birth to the concept of derivation. If a class A derives from a class B then class A inherits all the operations and variables of class B, plus other operations and/or variables make the derived class more specialized.
- simple inheritance means that the derivation relation leads to a tree of classes, whereas multiple inheritance, to a dag (directed acyclic graph) of classes. In the case of simple inheritance, there is a unique class at the top of the hierarchy; it is usually called class Object and represents the most general behaviours.

Inheritance introduces the notion of differential programming: the software components to be designed are considered as specializations of components already realized.

### 1.2.3. Uniformity

Uniformity means a way of programming in which all the manipulated entities are managed according to a minimum of principles. The principles here are that each class is an object and that each object is an instance of one single class. Uniformity achieves economy of concepts. It is based on the notion of meta-class.

- if each class is an object and each object is an instance of a class, then Meta-class is to be defined as the class whose instances are classes. Hence, Meta-class is one of its own instances.

## 1.3. Practicality

From more than one point of view Smalltalk constitutes the best synthesis of concepts relative to object-oriented programming. The use of this language leads to an advantageous solution for most of the problems previously mentioned. However, Smalltalk has not been designed for real-time applications nor for distributed computing: this, together with a certain inefficiency compared to compiled languages, today turns out to be a serious obstacle to its general use; on the other hand, it is regularly used for prototyping.

At the moment, it seems that any compromise should at least fulfill the requirement of encapsulation; languages like Ada and Modula-2 have been designed with that purpose in mind. Hybrid languages like C++ or Objective-C, that preserve the portability and the efficiency of C while proposing decent solutions to ensure encapsulation and inheritance, are also worth considering. Of course, none of these languages adheres to the uniformity principle, but this deficiency will not be felt seriously except in the applications relative to the modelling of cognitive processes (artificial intelligence), for knowledge is essentially reflexive (it applies to itself) and will be handled with much greater difficulty using formalisms that do not simply allow self-reference.

At present, there is a need for programming environments suited to object-oriented programming and producing code in efficient languages. SCOOP aims at being a step towards answering that need.

## 2. What is SCOOP?

Through a carefully designed user-interface, the tool SCOOP assists the software designer in the phases of architectural design and detailed design according to a formalism implementing a strict object-oriented approach. SCOOP also provides fully automatic translation from that formalism into Smalltalk for easy debugging, and into C++ as an efficient target language.

In the rest of this article we present the main features of the total SCOOP.

## 2.1. Object oriented approach within SCOOP

Within SCOOP the object-oriented approach can be considered at every level:

- the tool itself is being designed exclusively according to the object-oriented approach, simulating its own existence for its elaboration. It is being written in C++ on a Sun-3/50 using the MIT X Window system.
- when used by a software designer, the tool offers only a set of objects (with well defined operations). Among these, there are objects of the edited software, like protocols (addOperation, ...), and also other ones, like windows (move, ...).
- the tool allows software to be built in a formalism that implements only object-oriented concepts.

In the next few sections, "environment" means the set of all objects available to the user, and "system", the part of the environment made of objects related to the edited software. According to this terminology, for example, windows are part of the environment but not of the system.

## 2.2. Formalism of the software built via SCOOP

The reason why C++ was not chosen as a direct formalism for the user to write its own programs is that this language offers too many degrees of freedom, many of which are quite opposite to the object-oriented principles. Let's not forget that C++ is, above all, a super-set of C.

From the semantic and syntactic points of view, the formalism proposed to users of SCOOP only provides constructions ensuring encapsulation and simple inheritance. The formalism can be seen as a large subset of Smalltalk. The main restriction is relative to uniformity. For example, classes are not objects like all other ones: a class is not able to answer a message other than "new".

The reasons why C++ has been chosen as target language are that it is portable and efficient, and also that it permits a rather direct mapping of object-oriented constructions.

## 2.3. Visualisation of the system

Edited software has to be visually represented. The objects composing it (methods, variables, ...) are gathered in two sets, relations and entities, with the following meaning and implications:

- relations are, among others, methods (sequencesOf statements), protocols (setsOf operations), or the whole system (treeOf classes). These objects are never visually represented by textual means, using for example separators like "a.b.c" to designate a sequence, or nested constructions like "(a (b c))" to designate a tree. Instead, relations are always visually represented, according to the user's preference, either by specific spatial localization of the objects they relate, such as indentation for trees, or by explicit drawings, such as lines or arrows between related objects.
- entities are, among others, classes, statements or variables. These are represented either textually (for example, by an identifier) or graphically (by an icon), according to the user's preference.

Notice that both entities and relations are objects.

## 2.4. Navigation

As it is rarely possible to display a complete view of the system, navigation is introduced so that the user is permanently able to choose the most pertinent visual representation of the part of the system he is interested in.

As the system consists of entities and relations, there is at any time one current object (entity or relation) within one current relation. At a given moment, one can imagine the current relation to be inheritance, and the current object, the entity: class Object.

The current relation determines what subsystem the user is interested in. The current object determines what part of that subsystem has to be the center of the representation. Finally, the size of the window determines what context around the current object is actually represented.

Navigation represents all the possibilities for the user to choose other currencies.

Three primitives of navigation are implemented:

- navigation according to a given relation. For example, this allows the passage from an object to its successors or predecessors in a sequential relation, or from an object to its father or son in a hierarchical relation.
- navigation according to a given entity. This allows, for example, the passage from the definition to the use of a variable.
- direct navigation. This allows the passage from any currency to the definition of any object, specified, for example, by its name.

Shortcuts are provided to permit convenient combinations of these three primitives; they correspond to cursor moves, scrolling mechanisms and the "find" feature in normal editors.

## 2.5. Editing

Editing represents the set of possibilities for the user to modify the system. As the system consists only of objects (entities and relations), modifying the system means creating new objects (e.g. local variables within a method), deleting objects (e.g. an operation within a protocol) and replacing objects (e.g. the name of a class).

Two primitives are implemented for editing:

- "delete" removes the given object from the system.
- "create" generates a new object related to the given one in a way that depends on the given relation (e.g. a successor, in case of a sequential given relation).

Shortcuts are provided to permit convenient combinations of these primitives; they correspond to the "cut", "duplicate", ... features of normal editors. To realize these combinations we introduce the notion of selection. The selection consists of whatever part of the system; by default, the selection is the current object.

## 2.6. Consistency

The ensurance of consistency aims at eliminating any error that can be diagnosed before run-time. The checks of consistency occur at each editing operation. We distinguish the syntactic consistency from the semantic consistency:

- syntactic consistency: the permanent use of templates ensures that it will always be respected. These templates are provided by a syntax-driven editor. For example, it is not possible to introduce a message that contains no selector.
- semantic consistency: we confine ourselves to guarantee the existence of all statically referenced objects. For example, it is not possible to use a variable that is not defined anywhere. But, because of the dynamic binding, it is usually not possible to predict which method a given message will actually trigger at run-time.

## 2.7. User interface

The user interface aims at providing the most convenient way of commanding all actions. The user interface of SCOOP supposes a keyboard, a mouse with one button and a graphic display. The keyboard is mainly used for typing identifiers names. The mouse permits management of the windows, the selection of options in menus and the selection of objects of the system within the active window.

Some facilities are offered, related to the use of the keyboard, like macros and identifier completion.

The relation between moves of the mouse and moves of the corresponding pointer on the screen is configurable and is not necessarily linear.

Windows are the only places of the display where visualisation of parts of the system occurs.

Visualisation can be textual or graphic according to the user's preference. All windows are equivalent as to the possibilities they offer. There is no definite upper bound to their number but there is at least one window. As usual, if there are several windows, only one is active at a time. Windows freely overlap and may be partly located outside the display. The active window always appears on top of the other windows. Navigation and editing commands are issued only in the active window.

When editing, the default situation is that only the contents of the active window is updated according to the changed objects. But when issuing an editing command the user can optionally ask for a refresh of all the inactive windows if the system affects them.

When navigating in the system, the default situation is that the contents of the active window is updated according to the visited objects. But, when issuing a navigation command, the user can optionally ask for a new window to be created. In this case the active window becomes inactive and its contents remains unchanged, whereas the new window becomes active and its contents is determined by the destination of the navigation command. Furthermore, this is the only way to create a window.

This leads to a hierarchy of windows, such that window1 is the father of window2 if window2 has been created as a consequence of a navigation command issued in window1. This hierarchy has no topological consequences on the localisation of windows – only the titles of the windows keep track of it – but it allows some convenient managing facilities: when the user executes a windowing operation (move, ...), the default situation is that the operation applies only to the active window, but the user can optionally ask for the operation also to apply to all subwindows of the active one.

Four menus are supplied, which contain all the SCOOP commands. The user can choose between "pop-up" or "pull-down" appearance.

- environmentMenu: with general-purpose options, like "save", "translate", ... that apply to the whole environment.
- windowMenu: with options like "move", "close", ... that apply to the active window or also to its subwindows. It provides the visualisation facilities.
- currencyMenu: with options like "toTop", "toNext", ... that apply to the currencies. It provides the navigation facilities.
- selectionMenu: with options like "create", "copy", ... that apply to the selection. It provides the editing facilities.

Each option has a keyboard equivalence. Options are never added to nor removed from a menu; options may sometimes appear as not selectable.

## 2.8. Translation

Translation aims at producing executable code. Two translators are provided, one into Smalltalk and the other into C++:

- the semantics of the formalism being a strict subset of the one of Smalltalk, translation into that language is straightforward and establishes a one to one correspondence between the entities of the source and the ones of the target. Translation is thus fast and because of the close correspondence between the source and the target, messages issued from the debugging facilities of Smalltalk are directly transportable to the source. All this ensures easy debugging at source level.
- once the previous phase is complete, the translator into C++ is invoked. The following simple example illustrates the correspondence between the object-oriented notions and some of the features of C++:

classing and subclassing:
(inheritance)

```
class Rectangle: public Object {
```

variables of instances:
(encapsulation, polymorphism)

```
   Object*   upperLeftCorner;
   Object*   extent;
public:
```

operations:
(dynamic binding)

```
   virtual Object* display();
   virtual Object* extentGet();
   virtual Object* extentSet(Object*);
   ...
}
```

methods:
(messages)

```
Object* Rectangle::display() {
    leftParenthesis ->display();
    upperLeftCorner ->display();
    comma           ->display();
    upperLeftCorner ->plus(extent)
                    ->display();
    rightParenthesis->display();
}
  ...
```

Of course, this whole mechanism works only if there exist primitive classes, elaborated from the native types of C++ (int, ...) and from the native operations on these types (+,...).

## 3. Conclusion

Far from a revolution implying that one is to make tabula rasa of all acquirements, object-oriented programming may be considered as a significant evolution, leading to the topping of the conventional functional decomposition with a coherent super-structure: the objects. Many organisations involved in the development of software have, by the way, taken to the object-oriented approach: NASA, ESA, NATO.

At present, there are very few development environments suited to a strict object-oriented approach. SCOOP aims to remedy this absence. Among the elements constituting the originality of the tool, we recall:

- pure object-oriented approach: encapsulation for reliability, inheritance for reusability.
- fully automatic translation into Smalltalk for debugging, and into C++ for final executable.
- power of navigation along all the dimensions of the system
- convenience of the user interface
- verification of the syntactic and semantic consistency

Concluding this article, we hope to have clarified the principles of the object-oriented programming for those readers who were not familiar with the subject and also to have aroused an interest in the elaboration of the tool SCOOP.

## 4. Bibliography

G. Booch, *Software Engineering with Ada*, Benjamin and Cummings, 1987.

B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

B. J. Cox, *Object Oriented Programming*, Addison-Wesley, 1987.

A. M. Gueguen, *Smalltalk-80*, Eyrolles, 1987.

G. Booch, "Object Oriented Development", *IEEE transactions on Software Engineering*, February, 1986.

B. J. MacLennan, "Values and Objects in Programming Languages", *ACM SIGPLAN Notices* Volume 17, Number 12, December 1982.

C. Livercy, *Theorie des Programmes*, DUNOD informatique, January 1984.

B. J. Cox, "Object-Oriented Computing Concepts and Implementations", *IEEE Computer Society Press*, 1987.

S. Gjessin and K. Nygaard, *ECOOP '88: Proceedings*, Springer-Verlag, 1988.

B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

# Step-by-step Transition from Dusty-deck FORTRAN to Object-Oriented Programming

*Michel Bardiaux*

*Philippe Delhaise*

Plant Genetic Systems
U.C.M.B Laboratories
U.L.B. CP160
Avenue Paul Heger
1050 Bruxelles
BELGIUM
*mbardiaux@rubens.uucp*

*ABSTRACT*

BRUGEL is a moderately large software for computer-assisted design of biological macromolecules. Its development began 10 years ago using standard FORTRAN-77 and continues to be based on that language. To overcome two major flaws of FORTRAN, viz. lack of expressive power and waste of the address space, we have developed a preprocessor and a dynamic memory management system featuring automatic memory reclamation, structured objects and error handling. The implementation of BRUGEL has evolved towards object-oriented programming by selecting from various programming languages which features were promising enough to warrant implementation as well as the recoding needed for their usage. Each such feature had to compete not only with other features but with the necessary priority accorded to implementation of new scientific methods in the software. This demonstrates, in a sense, the *inevitability* of object-oriented approaches.

## 1. Introducing BRUGEL

BRUGEL (for Biochemistry Research Utilities, Graphics, Energy and Language) is an integrated software for computer-assisted design of biological macromolecules. Like any such package, it contains four major application-level components:

- Model building tools: the ability to construct (models of) new molecules from previously known ones.
- Simulation tools: computing physico-chemical properties of modelled molecules.
- Analysis tools: to exploit the enormous amount of data generated by simulations, compute geometric and other properties, etc...
- 3-D interactive graphics

It is important to note that, while electrical and mechanical CAD softwares were built by capitalizing on centuries of experience with drawing boards, chemical CAD is a very recent discipline, where no consensus exists regarding which questions are to be asked, to say nothing of how to compute them. Hence, the application contents of BRUGEL must evolve at least as fast as its implementation techniques.

In the context of this paper, the most interesting aspect of BRUGEL as an application is that the user interface is based on an object-oriented command language (without methods). Most commands perform one simple action, storing results in named objects that will be used as input for subsequent commands. However, objects at the command level do not currently enjoy uniform semantics, and moreover are not truly related to the object-management system that is used for implementing BRUGEL. Of course, this will be fixed in some future release.

## 2. Why FORTRAN ?

The development of BRUGEL began 10 years ago using standard FORTRAN 77, and will be continued in FORTRAN, not only because of the cost of conversion, but also because FORTRAN remains the dominant language both in biology-oriented public-domain software and on the supercomputers that are absolutely required for the *very* compute-intensive simulations.

Two major problems encountered while developing any large application in FORTRAN are:

- Lack of expressive power: card format, 6-characters identifiers, very few compile-time checks, no encapsulation mechanism at all, no complex object, no pointer. Although most compilers provide language extensions, their use compromises portability - a serious concern if one wants to benefit from a competitive market.
- Waste of the address space: all arrays must be given the largest size that one expects to need someday. Although most state-of-the-art computer systems feature both virtual memory and large physical memories, disks and boards are by no means cheap, and the needs for memory always seem to grow faster than prices decrease.

After being exposed for the first time to the UNIX **cpp**, we decided to write our own pre-processor, with similar specifications but tuned to FORTRAN, written in strictly standard FORTRAN and able to produce strictly standard FORTRAN. The macro facilities of **bpp** were then used to implement a dynamic memory management system. An unexpected result was that the system quickly started (and continues) to evolve towards a true object-oriented programming language.

## 3. The bpp pre-processor

**bpp** supports the **cpp** -like directives **#define** (useful replacement for the **parameter** statement), **#if** (useful for system-dependent unavoidable code, such as system calls and **open** options), **include** (vital for modular development). **bpp** macros have the same syntax and semantics as with **cpp**; their primary use is to hide the details of heap management behind a (more or less) standard-looking syntax.

**bpp** also provides some non-universally available FORTRAN extensions that improve the quality of code, such as **implicit none**, unlimited line length, in-statement comments, unlabelled **do/end do** loops, **do while**(), and unlimited identifier length (by maintaining a dictionary of re-mappings to 6-characters identifiers, if necessary).

Drawbacks: pre-processing takes times, can confuse symbolic debuggers, knows very little of FORTRAN syntax and nothing at all of semantics.

## 4. Memory Manager

### 4.1. Physical Level

Pointers in BRUGEL are similar to PASCAL or ADA pointers, i.e. they designate an object in a global area called the **heap**. Moreover, they are implemented in a portable way: not as addresses but as FORTRAN **integer**'s which are indexes in several large **equivalence**'d **common** FORTRAN arrays (one for each primitive FORTRAN type).

Memory blocks in the heap are obtained by calling a pointer-valued function, which uses the most classical algorithm: non-roving first-fit with boundary tags and an **avail** list (see (3)). When needed, and possible under the operating system, the heap can be extended with extra virtual space (e.g. with **malloc(3)**).

### 4.2. Objects Level

Very early in the design of the memory manager, we decided that to be able to manipulate large and complex data structures, we could not rely on manual memory reclamation but had to use some automatic system. But any garbage collection algorithm requires run-time knowledge of the contents of reclaimed blocks, to be able to identify and follow pointers *inside* heap objects.

For this purpose, all heap objects are endowed with a header containing:

- Physical size (total space used in heap seen as array of integers).
- Type (pointer to a heap object known as **type descriptor**, which contains such things as name of the type, size of a scalar of the given primitive type reported to size of an integer).

- Some debugging information.
- Logical size (useable space in heap seen as array of the corresponding primitive type, exclusive of header).
- Reference count (see below).

Since we needed structured objects as type descriptors, that important facility was made generally available. Record types, however, are not described at compile-time but *at run-time* , by calling subprograms. This means that the first thing an application using **bpp** has to do is to call **elaboration** routines (ADA terminology) to initialize the heap manager and then create its own types, initialize static pointers to the special value **NULL**, etc.

## 4.3. User Level

Fresh objects are created by the function **new(type,size)** where **type** is a pointer to a type descriptor (these are manifested as **common** variables in **#include** files) and **size** is either the desired length for heap arrays, or a variant tag for records. Pointer variables are assigned a value, not with an assignment statement, but by using the **_setq** macro, which handles the details of reference counting (see below). Non-pointer elements of heap objects are accessed using *normal FORTRAN semantics* , except that macros are used instead of variables: predefined macros, or new ones added for improved readability. Example:

```
_setq(vec, new(typ_r8,10))
#define _vec(i) _r8(vec,i)
_vec(3)=1.2
```

It is worth mentioning that the last statement will be pre-processed into

```
r8((vec+4)/2+3)=1.2
```

More complex **bpp** statements will result in very long FORTRAN statements, taxing the capabilities of some compilers.

## 5. Automatic Memory Reclamation

When we had to choose a garbage collection algorithm, it was soon clear that mark-and-sweep was out of the question since it requires all references to be reachable from a small number of root pointers, while in a FORTRAN implementation we would have many pointers in common, local and argument blocks. These could have been moved to special global areas, but it would have required another level of indirection and either non-dynamic resources or another memory manager.

Hence, we decided to re-use a reference-counting algorithm that had been developed by one of us (M. Bardiaux) and B. Marchal, while implementing a PROLOG interpreter (based, of all things, on parts of a LISP interpreter, itself written in ADA - influences still very much apparent in BRUGEL).

The basic rules are:

- Each heap object has a non-negative integer-valued property called **reference count**
- When a pointer variable receives the 'index' of a heap object (i.e. **references** it), increment the reference count of the object.
- When a **pointer** variable stops designating an object (i.e. **dereferences** it), decrement reference count.
- When the reference count goes below zero, return the object to the free memory pool.

Drawback: as is well known, reference-counting is unable to reclaim structures containing circular references. We are now adding debugging tools to detect such cases.

## 5.1. Functions and Anonymous Pointers

A tricky situation occurs when a pointer returned by a function is used as argument of another subprogram, since the reference is in the call packet and thus anonymous. This has been solved by adding the following rules:

- **reference** pointer arguments at entry and **dereference** them at exit.
- All pointer-valued functions return their result with a reference count of zero. Hence, an extra check is needed at each **dereference** against the possibility that the object is actually a function result to be returned.

Drawback: **out** and **inout** (in ADA terminology) pointers *must* be avoided (anyway it is must better to write functions returning a pointer to a list of pointers).

Drawback: user subprograms must be festooned with housekeeping code at entry and exit. We are currently extending the **bpp** preprocessor to generate that code automatically.

## 6. Error Handling

The conceptual model of errors in **bpp** is based on the ADA **exception**, i.e. the program can be in one of three states: no exception; an exception was raised; an exception was raised but is being handled. Exceptions are implemented as pointers to strings (the name of the exception), manifested as **common** variables in **#include** files.

However, FORTRAN does not offer a portable mechanism to transfer control from the point where an exception was raised to the point where it will be handled. Even non-portable mechanisms (such as **setjmp/longjmp** under UNIX) would not allow us to **dereference** heap objects while pointers are popped off the stack. That goal is achieved by adopting a fixed style of control flow: all subprograms should exit immediately after a called subprogram returns with exception raised; all subprograms should exit immediately after entry if an exception was raised. In this context, **entry** means after **reference-ing** all pointer arguments, and **exit-ing** includes **dereference-ing** pointer arguments and local pointer variables **reference** and **dereference** act even when an exception was raised).

Hence, although several subprograms might be called before the exception is handled, they should have no effect. Pointer-valued functions allow us to reinforce this security by designating the resulting object through a *local* pointer and committing it to the returned pointer at the last time before exiting.

### 6.1. Interruptions

Many operating systems allow users to **interrupt** a running program. In BRUGEL, when such an event arises, we would of course like to abort the current command rather than terminate the whole program. However, the FORTRAN runtime library is usually *not* interruptible. This problem was relatively easily solved by limiting the system-dependent part of event handling to the raising of an exception.

## 7. Debugging Facilities

A first set of tools contains subprograms to check or dump heap objects, using information from their type descriptor. These are callable from FORTRAN, from the BRUGEL parser, and, under UNIX, from the symbolic debugger.

A second set consists of redefinitions for the interface macros between user code and the heap, allowing tracing and type and index range checking at each access to heap objects in a source file.

Of course, the heap manager also maintains and prints statistics about current and peak memory usage.

Drawback: as usual when using a preprocessor, symbolic debuggers have no knowledge of macros.

## 8. Tools Library

### 8.1. Conversion

Subprograms are provided to convert between heap arrays and 'classical' FORTRAN arrays (i.e. declared via **dimension**), such as:

> *_pointer function copy_i4(vector, nelem)*
> *subroutine export_i4(ptr, ito, maxelem)*
> *_pointer function others_bit(logicval,number)*

## 8.2. Arrays and Records

While conversion subprograms must exist for each basic type, the following ones can be programmed to be universal since all pointer arguments carry with them the full description of the designated heap objects:

> *_pointer function conc(p_to_ar_1, p_to_ar_2)*
> *subroutine copy_slice(into,li,ri,from,lf,rf)*
> *_pointer function slice(p_to_ar,l,r)*
> *logical function equal(pl,pr) ! non-recursive*

A need is currently felt for the possibility of handling *totally* arbitrary data structures, so we will add functions such as equivalent (recursive comparison), duplicate (recursive copy) and decircularize.

## 8.3. Coded Output

The style of coded output mimics the ADA **text_io** package, because we felt it provided maximum security by avoiding mismatches between the lengths of formats and I/O lists. A typical output statement would be

> *call fprint_tokr4(bru_out, var, '(e10.3)')*

The first argument is actually a pointer to a **Logical-unit Description** Block. The **ldb** level (sitting on top of the FORTRAN runtime) provides line buffering and margin and page control, but also more original features such as inactivation and mirroring. We are currently adding the possibility of directing output to line-structured memory files rather than FORTRAN files.

## 8.4. Coded Input

BRUGEL has no full-fledged coded input system, because most user input is done via the command interpreter. Limited facilities exist for question-and-answer-style input during execution of a command, and for reading card-format data files of known structures, such as energy parameters libraries and Brookhaven Protein Data Bank files.

## 8.5. Machine-independent Binary I/O

Our current installation is a network of UNIX machines and relies on the NFS protocol. It was considered necessary that binary files of known structure could be read by BRUGEL running on any machine in the network, regardless of the host that had produced them. To achieve this goal, binary files contain a label block, and data blocks contain size and type tags. The **bpp** binary I/O system currently supports only read and write of 'classical' FORTRAN arrays, because full, efficient binary I/O of arbitrary heap structures will be possible only through memory-mapped files.

Moreover, FORTRAN runtime libraries usually endow binary files with a record structure that we do not need, resulting in poor performance. When possible under a given operating system, BRUGEL uses raw file access (e.g. by direct calls to **read(2)** under UNIX).

## 9. What about methods ?

In the context of object-oriented programming, the term **method** conventionally refers to subprograms designated in the type description of an object to perform specific operations. We are only now considering their use in **bpp**, for several reasons:

- The accessibility of type descriptors at runtime already allows a fair amount of genericity, although limited to primitive types and operations.
- The implementation of the use of methods implies the writing *and maintenance* of a few subprograms, in assembly code and dependent on the compiler architecture. We have only recently decided that the added expressive power was worth the extra development costs.
- The coding style in BRUGEL has been strongly influenced by ideas from functional programming, and thus is not always compatible with object-oriented programming style as found in textbooks.

## 10. Efficiency

One may (and we had to) wonder whether the potentially very complicated statements generated by the pre-processor will not result in an unacceptably sluggish software. We have observed that performance measurements give very different results depending on the hardware used as well as on the task required of BRUGEL, and must also be assessed using different criteria.

### 10.1. Compute-Bound Tasks

On superminis, supermicros and workstations, the complexity of post **bpp** code does not seem to be a problem for optimizing compilers. Anyway, the bottleneck is usually floating-point computations, and the indirection overhead is mostly hidden.

On minisupercomputers, however, vectorizing compilers do not fare as well: the fact that all data is in a single array seems to make them overcautious; many loop-constant expressions are not factored out but rather vectorized. Although 'full vectorization' can be achieved, many unnecessary vector operations are generated. Since floating-point vector operations are executed quickly, the indexing overhead is strongly apparent. For the most compute-bound subprograms, the only effective solution has been to make the heap invisible by transmitting heap data to subroutines as 'standard' FORTRAN arrays.

### 10.2. Computer-Assisted Design

In this case, the true measure of efficiency is not raw speed (although BRUGEL is somewhat sluggish on low-end 68020-based systems), but efficient memory management, and the number of application-level features that could be added using the object-oriented approach.

## 11. Historical View and Conclusions

At this point, although some original features have been mentioned, one might well wonder whether this is not another case of 'reinventing the wheel'. The interesting aspect is that development started at a time when object-oriented techniques were far from being recognized software production tools. BRUGEL has evolved by selecting from various programming languages which features were promising enough *in this particular context* to warrant implementation. As mentioned in the introduction, each such feature had to compete, not only with others, but with the necessary priority to development of new application-level functionalities. This demonstrates, in a sense, that object-oriented programming is a natural approach to software development. But the *order* in which features were introduced is also indicative of their priority in practical cases: an object oriented command language came first, but dynamic memory allocation *with automatic reclamation* was the next step, while it is at best a secondary design goals in languages such as C++ and Objective-C.

Finally, do our initial reasons (re-evaluated in 1985) for sticking with FORTRAN and adding our own ingredients still apply ? Unfortunately, yes. VAX/VMS FORTRAN 77 is becoming a *lingua franca* but can not be called a standard, even *de facto* As far as we know, FORTRAN 88 is not to feature a pointer type. The only rich *and* strictly standardized language, i.e. ADA, is still generally expensive. And C-based object-oriented languages are still young.

As in the old Chinese curse, we *do* live in interesting times !

## References

[1] Allen, J., *Anatomy of LISP* , McGraw-Hill, 1978.

[2] Angell, I.O. and Griffith, G., *High-Resolution Computer Graphics Using FORTRAN 77* , McMillan, 1987

[3] Cox, B.J., *Object-Oriented Programming : An Evolutionary Approach* , Addison-Wesley, 1986.

[4] Delhaise, Ph. *et al.* , *Analys is of data from computer simulations of macromolecules using the CERAM package* , Journal of Molecular Graphics, Vol. 3, No. 3, September 1985.

[5] Ichbiah, J. *et al.* , *Reference Manual for the ADA Programming Language* , ANSI/MIL-STD-1815A-1983, Castle House Publications, 1983.

[6] Jensen, K. and Wirth, N. , *PASCAL User Manual and Report* , Springer-Verlag, 1975.

[7] Knuth, D., *The Art of Computer Programming, Vol. 1 : Fundamental Algorithms* , Second Edition, Addison-Wesley, 1982.

# Distributed Logic Programming

*Andreas Krall*

*eva Kühn*

*Ulrich Neumerkel*

Institut für Praktische Informatik
Technische Universität Wien
Argentinierstr. 8, A-1040 Wien, Austria
*andi@vip.uucp, eva@vip.uucp, ulrich@vip.uucp*

*ABSTRACT*

M-VIP (Multiuser Vienna Integrated Prolog) is currently under development at the Technical University of Vienna. M-VIP extends standard sequential Prolog by multi-user capabilities. It enables the development of shared knowledge base systems fully integrated in workable UNIX environments. In this paper we present the language extensions of Prolog to allow the concurrent execution of Prolog queries, initiated by different users and its implementations.

## 1. Introduction

Prolog is a fifth generation language mainly used in the fields of artificial intelligence, knowledge and software engineering. With Prolog implementations of expert systems, deductive database systems, natural language systems, compilers and program generation are realized. It is well suited for purposes of rapid prototyping as well as a test tool.

The success of Prolog stems from its declarative features. This means that one is much less concerned with the "how to" in favor of the "what". Essentially, Prolog programs are statements about the world. The execution of a Prolog program is initiated by a query on this world model. The execution of a program is the side-effect of proving the query.

Available Prolog systems are implementations of a sequential Prolog language. Moreover they are mostly restricted to main memory and incapable of handling large amounts of data. VIP (Vienna Integrated Prolog) provides the possibility of storing rules in their internal representation on disk and improves the efficiency of memory utilization on conventional computer architectures. A more detailed description of our Prolog implementation will be given in the next section.

There are several approaches to extend Prolog by parallelism. All of them are concerned with gaining a speed-up in proving a Prolog query by parallel execution of subqueries. These subqueries can be considered as separate processes that require some kind of synchronization.

In section 2 we will give a short introduction to sequential Prolog limited to the features we need to describe M-VIP as well as an overview of VIP Prolog. In section 3 the motivation for M-VIP and its specification are shown. A comparison to related work is given, including aspects of multi user capabilities in a procedural language (C) and in an object oriented system (GemStone). The close relationship to distributed databases is pointed out. In section 4 we describe the implementation methods feasible under UNIX: A distributed and a centralized approach. An analysis of their advantages is provided. Section 5 sums up with an outlook on future work.

## 2. Prolog

The programming language Prolog (PROgramming in LOGic) has been developed by Alain Colmerauer and Philippe Rousell in the early seventies. Prolog is based on Horn clause predicate logic. A Prolog interpreter is a program, which proofs Horn clauses, a subset of full first order logic with a fixed but efficient strategy.

The elementary objects in Prolog are terms. Terms are constants (atoms, integers, reals and the empty list), variables or compound terms. A predicate looks like a compound term, but it does not represent structured data, it is a predicate logic assertion. A Prolog program is a set of predicates. Predicates consist of a sequence of (Horn) clauses. A clause is either a fact or a rule.

During runtime the clauses of a program are stored in the database. In Prolog there exist two system predicates (builtin predicates), which manipulate the internal database: *assert* to insert a clause and *retract* for the deletion of a clause. In Prolog programs and data are treated in the same way. Facts can be asserted as data and then treated as a program. Terms can be interpreted directly with the builtin *call* (metacall). The internal database can be written to a library file – *store* – and loaded from a library file – *load*.

For sequential Prolog we developed the VIP system (Vienna Integrated Prolog) [5]. VIP extends standard Prolog by a module concept and uses a new implementation technique. In contrast to other Prolog implementations VIP is based on a different abstract machine (VAM- Vienna Abstract Machine). The VAM is an interpreter which gains the same speed as the compilers based on the WAM (Warren Abstract Machine). This is due to the fact, that the VAM avoids the parameter passing bottleneck and can make more optimizations during runtime, preserving locality of memory references.

### 2.1. DB-VIP

As previously mentioned Prolog is closely related to the relational database model and well suited for database applications. In particular Prolog is more expressive than existing database languages, as it provides the facility to model more complex data objects by using rules, recursive query formulation, as well as recursive data modeling, and the representation of extensional and intensional data. These powerful features of Prolog have been widely accepted, but to make Prolog a real database machine it needs the following extensions: improved control strategies, maintenance of very large amounts of data, shared data and multi user access. The first issues have been addressed in other work. See also the implementation of VIP-MDBS, a multi-user database management system under VIP-Prolog [6]. The last two issues are covered by the work of the VIP group presented here.

## 3. M-VIP

### 3.1. Introduction

The use of Prolog for implementing database and expert systems raises the need for multiuser capabilities. With the project M-VIP we have the intention to implement an efficient multiuser Prolog on existing computer architectures and operating systems. As the results in the research of fine grained parallel architectures are not suited for conventional systems, we decided to support the sharing of common databases only. Prolog in M-VIP is thus essentially sequential Prolog extended by primitives for synchronizing the updates of different users on shared databases. Communication is performed by shared databases which can be modified with the builtin predicates *assert* and *retract*. A locking mechanism enables the users to synchronize simultaneous updates on shared databases.

### 3.2. Language Extensions

The main design goal of M-VIP is to restrain the language extensions in order to keep the differences for the application programmer between single user and a multi user programs as small as possible. M-VIP contains only some additional builtins for synchronization.

For structuring the databases VIP contains a module concept. A module is a set of predicates, which belong in some sense together. A program contains modules which are stored in a library. The VIP-interpreter is started with a library. At runtime additional libraries can be loaded. In M-VIP the same library can be used by different interpreters at the same time. The interpreters can simultaneously modify the database in the libraries through *assert* and *retract*. To avoid synchronization problems there exist builtins for locking predicates and modules:

| | |
|---|---|
| *load(LibraryName)* | load a library and prevent access of other interpreters |
| *readload(LibraryName)* | load a library; the library is not modifiable. |
| *sharedload(LibraryName)* | load a library modifiable by different interpreters. |
| *unload(LibraryName)* | releases the library. |
| *store* | saves all modules in the main library. |
| *store(ModuleList)* | saves the modules of ModuleList in the main library. |
| *store(ModuleList,LibraryName)* | saves the modules of ModuleList in the library LibraryName. |
| *readlock(ModuleName)* | locks a module against updating. |
| *updatelock(ModuleName)* | locks a module against updates and reads by other interpreters. |
| *unlock(ModuleName)* | releases all locks for this module. |

The same locking mechanism exists for predicates:

*readlock(ModuleName:PredicateName/Arity)*
*updatelock(ModuleName:PredicateName/Arity)*
*unlock(ModuleName:PredicateName/Arity)*

In sequential VIP there are available only the *load* predicate and the three variants of the *store* predicates. If VIP and M-VIP use the same libraries, VIP programs have exclusive access to the library. The predicate readload is used for program libraries and sharedload is used for mixed data-program libraries.

The application programmer does not need to use locking predicates. If he knows that the computation is correct anyway, he can freely use *assert* and *retract* on shared libraries. But if no locking predicates are used, there is no possibility to determine, at what time the other interpreters are informed about a database change. If locking is used, it is guaranteed, that between an *unlock* or *store* and a *lock* the view between the different interpreters of the shared library modules is the same. In the following a short example for seat reservation for an airline database is given. The data is stored in the shared library module database. The relation flights contains the flight number, the date and the number of available seats. The predicate *reserve_seat* reserves one seat or writes an error message and fails.

```
reserve_seat(Flightnr, Date) :-
     writelock(database:flights/3),
     database:flights(Flightnr, Date, Seats),
     Seats > 0,
     Seatsm1 is Seats - 1,
     retract(database:flights(Flightnr, Date, _)),
     assert(database:flights(Flightnr, Date, Seatsm1)),
     unlock(database:flights/3).
reserve_seat(_,_) :-
     unlock(database:flights/3),
     write('no seats available'),
     fail.
```

This small example shows how to use the locking predicates. More complex database access mechanisms can be realized with this primitives with methods as described in [4]. Two problems occur when two interpreters want to lock the same predicate or want to update a locked predicate. In such a case one interpreter is served and the second must wait. A long wait can be caught by a timeout exception [3]. If two or more interpreter are mutually waiting for the same resources a deadlock exception is raised.

## 3.3. Comparison with related work

Naish et al. [8] incorporate a similar mechanism as our locking schema into Prolog. However they do not use low level locking primitives, but they provide a high level builtin based on complete transactions. Their transaction predicate can be written with our primitives. They also use special database predicates (*db_insert* and *db_delete*), which are not available in standard Prolog.

Incorporating parallelism into the Prolog language is still a field of research. In Concurrent Prolog [9] goals are seen as separate processes synchronized by special variables. This concept uses the notions of stream and channel for communication. But concurrency here only means the concurrent execution of goals within one single user query. Furthermore, the semantics of concurrent Prolog is completely different from the semantics of sequential Prolog and thus no longer related to the logic programming paradigm. Similar approaches are Guarded Horn Clauses and Parlog [9].

A suitable comparison with procedural languages are the *fork* and *exec* calls under UNIX. The main conceptual difference to our approach is, that although a fork system call spawns up a new process, it is not coupled to its parent process via shared data. Communication only runs over pipes or even more loosely, over files.

GemStone [7] that is based on the object oriented programming paradigm, is an approach where multi user features are incorporated. Each user gets a shared copy of the so-called object table (*ie* the shared table) where he performs his queries and updates locally. Changed objects are overlaid into the shared table by using an optimistic concurrency scheme, that tries to solve conflicts on commit time. A disadvantage of this locking mechanism is, that eventually really complex transactions have to be undone.

Distributed databases, including distributed knowledge base applications are related to our approach, too. As our primitives are not concerned with a full transaction concept; recovery problems and more sophisticated locking mechanisms delayed on the application level [1]. The work of Carey et al. [2] deal with concurrent control and recovery mechanisms for a Prolog environment on the application level. They deal with concurrency aspects of the parallel execution of queries as well as with multi user queries. In contrast to databases, Prolog provides no modification command for already existing facts. Therefore so-called phantom facts can be generated during conflicting updates. A solution is provided by precision locking that is based on two phase locking but may set locks on groups of logical objects, specified by predicates. The approach is similar to our approach, as they also use read and write locks on queries and facts. But they do not consider rule locking. A similar work to provide consistent concurrency control for logic programming on the application level can be found in [11].

## 4. Implementation

### 4.1. Introduction

To enable an efficient manipulation of objects symbol oriented languages must convert the names of objects into an internal representation. Every Prolog system needs for atoms, functors and predicates a symbol manager. The main function of the symbol manager is the mapping from the text to the internal representation of the name. Our implementation uses a 32 bit representation, where 6 bit are used for a tag and 26 bit can be used for the atom index. Names with a length of up to nine characters can be stored directly in the atom index. The most common combinations of one, two and three character sequences are represented by an 8 bit encoding. Other strings are stored using conventional hash tables. In M-VIP a centralized and a distributed implementation of the symbol manager is possible.

### 4.2. Distributed Implementation

For the distributed implementation we assume a network of independent processors with memory communicating over a fast communication network, for example workstations connected by Ethernet. Each interpreter has its own symbol manager and library manager. Additionally each library has its own symbol manager and library manager. Therefore a library is not only a collection of Prolog clauses, it is also a process managing the stored data. An interpreter loads a copy of the library, converting the symbols in the library to the representation of the symbol manager in the library. The loader loads an identical copy of the library, generates a conversion table for the atom indices and replaces the indices in the memory. The library process manages the locking and the distribution of updates. In case of a *lock* a message is sent to the library process, which sends back an acknowledge or suspends the sending process. A suspended process is activated, when an *unlock* occurs, after a timeout, or when the process locking the resource is killed. An update in an interpreter is done immediately in the local copy and stored in a buffer. At *unlock, store,* or buffer overflow the update messages are sent to the library process, which distributes the updates to the other interpreter processes.

An advantage of this implementation is, that the interpreters can work independently on their own copies of the libraries. Synchronization is only necessary during updates. In this case a bulk of *assert* and *retract* together with the textual representation is sent to the other interpreter processes. A disadvantage is the waste of memory for the local copies of the library and the conversion time for the atoms in different symbol tables.

## 4.3. Centralized Implementation

For the centralized implementation we have single processor or shared memory multiprocessors in mind. The centralized implementation uses one global symbol manager for all libraries. All libraries and interpreters use the same internal representation of names. Each library has its own library manager. The interpreters are independent processes, but use the same copy of the library. The interpreters use shared memory for the storage of the library copy. Therefore updates in the library are transparent immediately to the other interpreters. The library manager is responsible for correct access and update to the library. Every time a symbol is entered or deleted (at input of terms, at updates in the database) the global symbol management is activated.

The advantage of this implementation is the efficient use of memory, if sharded memory can be used. In absence of shared memory the libraries must be copied or the different interpreters must be implemented as one process. Updating the library is very fast, because no messages must be distributed to other processes. A bottleneck could be the common access to the shared memory and the single symbol manager. But the symbol manager can be split into independent symbol managers, which are working on different parts of the global symbol space. Due to the fact, that most of the symbols are stored directly in the atom index, the calls to the symbol manager do not occur so often.

## 4.4. Comparison

It is difficult to predict the relative performance of the different implementations. The actual run time behavior depends heavily on the applications. There are applications which read and generate a lot of symbols during runtime, but do not make any updates on shared libraries. This is especially the case, when string manipulating programs are executed. On the other hand there is a lot of programs, which make large changes in the shared libraries, but do not generate any symbols during runtime.

It is also possible to combine the centralized and distributed implementations. The global symbol table can be used on single workstations and the distributed symbol table can be used for the communication of different workstations.

## Conclusion

We presented an extension of Prolog, M-VIP, which extends the usability of Prolog for multiuser applications. The demand for multiuser expert systems and knowledge bases and the use of Prolog as implementation language for such systems have been the motivation for this extension. We tried to keep this extension simple and close to standard Prolog and presented two different implementation methods. In the future we will test the usability of our concept on the database system DB-VIP and on multiuser expert systems.

## Literature

[1] P. A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison Wesley, 1987

[2] M. J. Carey, D. J. DeWitt, G. Graefe, *Mechanism for Concurrency Control and Recovery in Prolog – A Proposal,* First Int. Workshop on Expert Database Systems, Larry Kerschberg (ed.) The Benjamin/Cummings Publishing Company, 1986

[3] S. Dulli, e. Kühn, *The Concept of Exception Handling in VIP,* GULP 1988, Roma, 1988

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger, *The Notions of Consistency and Predicate Locks in a Database System,* Communications of the ACM, CACM 19/11, 1976

[5] A. Krall, *Implementation of a High-Speed Prolog Interpreter,* Proceedings of SIGPLAN 87 Symposium on Interpreters and Interpretative Techniques, SIGPLAN 27/9, 1987

[6] e. Kühn, T. Ludwig, *VIP-MDBS: A Logic Multidatabase System*, Int. Symposium on Parallel and Distributed Database Machines, Austin Texas, Dec. 1988

[7] D. Maier, J. Stein, *Development and Implementation of an Object-Oriented DBMS* in B. Shriver, P. Wegner, (ed.) *Research Directions in Object- Oriented Programming*, Cambridge, MIT Press, 1987

[8] L. Naish, J. A. Thom, Kotagiri Ramamohanarao, *Concurrent Database Updates in Prolog*, Proceedings of the Fourth International Conference of Logic Programming, Melbourne, MIT Press, 1987

[9] E. Shapiro, *Concurrent Prolog – Collected Papers*, MIT Press, 1987

[10] L. Sterling, E. Shapiro, *The Art of Prolog*, MIT Press, 1986

[11] U. Schreier, H. Wedekind, *Supporting Concurrent Access to Facts in Logic Programs*, Proceedings of the 3rd International Conference on Data and Knowledge Bases, Jerusalem 1988

# Performance Evaluation: The SSBA at AFUU

*Christophe Binot*

Hewlett-Packard

*Philippe Dax*

E.N.S.T

*Nhuan Doduc*

Framentec
*ndoduc@framentec.fr*

## 1. Introduction

The UNIX phenomenon has essentially influenced the data-processing world quite recently. Benchmarking is one of the most visible aspects of this evolution, and AFUU, our French UNIX users group with its SSBA clearly is a proof of its importance.

This paper should be read as a progress report. As a matter of fact it's the first one about SSBA, the acronym standing for **"Suite Synthetique des Benchmarks de l'AFUU"** . We're not going to mention any theoretical foundation of benchmarking; instead, you'll be informed about the genesis and the evolution of SSBA. More precisely, we're going to browse through a three part report: the first part will narrate the creation of SSBA, eg. the fifteen months period before its appearance, with some arguable reasons and some debatable decisions; the second part is truly an advancement report covering the first six months of its existence; the last part will outline some projected works for the next twelve months.

We might as well say that the paper will deal successively with the justifications and specifications, the alpha and beta testings, and the first customer shippings, of the SSBA.

Finally, with local conditions permitting, some numerical results may be shown by April time: we still hope we can collect sufficient, significant and coherent data by that time.

## 2. The SSBA: Why and what?

UNIX is deeply and definitely modifying the data processing landscape. As the most common denominator, it is the cause and the opportunity for lots of activities that otherwise would not occur: benchmarking is one of them.

Before UNIX, benchmarks did exist, but were sparse and unregulated. At best their usefulness was limited. In fact, given their constraints at those moments, even the most ardent benchmarkers must have had much trouble to justify the necessity and validity of benchmarks.

Now, with UNIX spanning over the whole range of hardwares, benchmarks are no more considered as happy happenings but recognized as real necessities of life, with true influence on many 'bread-and-butter' decisions.

The need for a good and accepted tool for system performance evaluation would still not be enough to trigger the creation of the Benchmark group of AFUU. *Why do the job when you can wait for it to be done?*. A second contributing factor is the fact that public existing results are clearly unsuitable for any use: they are commonly fuzzy, incomplete and even sometimes contradictory; furthermore, they are usually not documented and as such not reliable at all. The third factor is human: data processing managers and users, permanently flooded with new offerings every other day, do truly, and daily, need such a tool. The last decisive element is UNIX: with UNIX, not only is such a tool a need, but, even with so many different UNIXes, is such a tool feasible. Indeed, right now, the technique is appropriate and the economics can be shown to be good (some commercial benchmarks and money-making benchmarkers have already bloomed in Europe!). Of course, benchmarks suites do exist elsewhere, eg. MS-DOS world... but what about any excitement over a single user single tasking operating system running on the same family of chips?

The Benchmark group was born on March 87 as a forum of, initially, end-users vying for comments, hints and warnings, numerical whenever possible, from lucky-and-happy users and/or not-so-happy-but-experienced victims of fate. The cruise speed of one day long meeting every six weeks was reached at once; the morning sessions are filled with presentations from manufacturers and vendors and also with talks from big account users (about their needs or experiences) or academia workers; the afternoon sessions are devoted to internal works targeting the creation of SSBA.

The very first decision is truly opportunistic. We immediately recognize the need for a benchmark (suite) for each domain, such as graphics, database, realtime, transactional... to name only the most trivial ones. However we also think that the need is more urgent to get something done very quickly to help clarifying the water where small fishes, medium ones and giant monsters are born every five minutes. Looking back by now, this tactic has proven to be sound, as we'll see down here.

So the subject for the first release of what is to become SSBA is narrowed to, on one side, the most popular benchmarks, and on the other side, the easiest parameters of each hardware.

The most popular benchmarks are not necessarily the best benchmarks. Their most common and famous characteristic is their age which allowed them to be used in each and every circumstance. In fact, they shared also another essential characteristic: the easiness of use. The two most famous are, without contest, the Whetstone and the Dhrystone (although I would vote for the latter in term of popularity). We don't need at all to go in any detail about the meaning and validity of any of the pair. Too many papers have definitely analyzed them. All have concluded that their values are very limited but not a single announcement does NOT contain results from them. Worse even, some announcements may contain ONLY their results. They are not perfect, but so is our world, and this is the reason why this pair has earned its place in our suite.

The most trivial parameters are also not the most important ones about the hardware; we put emphasis on some of them that are easy to evaluate, such as raw central processor speed, disk throughput... but we do include too some measurements about cache effectiveness, linearity about workload...

We want also to balance between the weights of C and Fortran and, for this very first version, Pascal, Cobol... are temporarily left out.

In summary, we settle on the following public domain benchmarks: Dhrystone *(500000 iterations)*, Whetstone *(50000000 instructions)*, Linpack *(100x100, we're targeting general computers, we're not looking at peak performances from Class VII or above machines...)*, Utah, Byte, Saxer *(10 Mega bytes I/O)*, DoDuc... all of which are used as is. We add TestC as a very simple test for C compiler, Outils which includes usual UNIX commands used casually and daily by UNIX users... We use BSD, as it was intended to be used, to look at performance of memory management and system calls. We modified Musbus, not only because some of its functionalities have already been taken care of earlier, but because we want it to be flexible and customizable by informed users. Finally, *noblesse oblige* , we put ahead of these benchmarks the famous formulae by Bill Joy which, after all, is not a too bad MIPS measurement.

As you may already notice, the SSBA does not pretend to be a new benchmark: it wants to be a clear, common, easy-to-use and easy-to-understand one. The added value from our Benchmark group is just the work to collect, interpret, analyze valuable elementary benchmarks and assemble them so that the SSBA can be used, easily and quickly, by each and all of us.

## 3. The SSBA: The first six months.

Version 1.0, also known as *"Sainte Germaine"*, was born on June 15th 1988, after 10 working sessions of the Benchmark group. Attachment #1 is the announcement paper as well the user's guide. Overnight it earned warm welcomes from many horizons and so proves that it truly fills a real need with a timely release. It also was noticed at once by professional magazines with many favorable comments. But the most encouraging fact is that it is immediately used by big or prestigious organisms as part of their call for tenders.

However, because of the complexity of the shell script, but also because not every UNIX is complete or standard-compliant, some bugs or deficiencies were discovered very quickly, which proves that it was really and widely used, at the least tested, in many places.

Comparing the version 1.0 to an beta one, we quickly reacted and released version 1.01, *"Sainte Marina"*, on July 20th. During the 'summer vacations' in France, the rate of use did not decrease at all; instead, it even became the test tools for an important organization specialized in reporting and consultancy. Version 1.2, *"Sainte Odile"*, released on Dec 14th, is presently the last public one, not counting numerous internal releases 1.1x.

Another interesting factor to note is that very gradually, it gets definite attention from hardware vendors, which again reflects the heavy use by (sometimes important) potential buyers.

News about the SSBA even spread to outside of France, and not necessarily because of an announcement inserted in the Volume 8, # 1, Spring 88 issue of the EUUG Newsletter. We got enquiries from places as far as East Asia, and the SSBA was even sent to a few partners in the US.

Gradually, many results arrived to the Benchmark group but not all of them: some results are included as proprietary informations inside confidential papers from vendors which require recipients to get their approval before sending to our group. This situation was not really unexpected and we think that right now we have found a way to deal with it.

In summary, the modifications from Version 1.0 to Version 1.2 fall in three categories.

First there are the minor and unavoidable bug corrections regarding the shell script and the improvements about result managements and printouts.

Then there are two important bugs all two located within our Musbus-modified part. Ken McDonell, to whom the SSBA was submitted from a French distributor, has discovered, *paternite oblige*, the first. Briefly, our shell script, while preparing and monitoring the workload test, as customized for the SSBA, inadvertently destroyed some items before execution. The second bug was discovered and cared of by ourselves, saving a bottle of champagne to the happy discoverer of the third, not yet discovered, bug. As a matter of fact, the importance of the third bug is for us to judge, but we're really willing to pay the price for it: for this as for the whole quality of the SSBA, you can trust us.

The third kind of modification steems from feed backs and numerical as well as qualitative results of numerous executions. We've increased many initial values eg. 100000 to 500000 Dhrystones, 10000000 to 50000000 Whetstone instructions, 16 users instead of 8 in the Byte suite... We've also reimplemented our way to use the BSD suite not only to keep it within reasonable running time, but also to make it very UNIX flavors independent (eg. Version 7, System III, BSD, SystemV...).

The final results collected after the first six months of existence may be enumerated here: IBM 6150 (115,125,135), DEC VS(2000,3200,3500) and 6220, Pyramid 9810/9815, Cray2, Intel (301,302), Apollo DN(3000,4000,10000), Gould NP1 and PN(6000,9000), Apple Mac II, Sun 3(50,60,110,160,260) and 4(110,260), Philips/TRT P9070, Sagem SX900, Prime EXL316, ESD SDX2000, Nixdorf Targon35, Matra XMS7000 and MS1316, Jistral Jispac 4016, Bull DPX(1000,2000/20,2000/27,5000) and Questar 700/20, Intergraph (C100,300), Mips (800,1000,2000,120), Unigraph (3000,6000), Convex C201, HP 9000(350,360,370,550,825,835,850), NCR Tower (600,650), Siemens PC-MX2, Unisys 7000/40, Sony 1850, Telmat T3000 and STE30, Silicon Graphics 4D/70, and FPS 300 (aka Stellar GS 1000).

This list, taken from our Tribunix newsletter, issue # 24, January/February 1989, counts approximately 60 machines. However we need to point out that none of these has yet been publicly and officially released by our Benchmark group: some of them will be announced, after careful review, on the occasion of our Convention UNIX which stands in Paris by March 89.

## 4. The SSBA: What's ahead?

The future of SSBA may tentatively be divided in three parts.

On the very short term, up to the 89 'summer vacations', we're conducting as usual a two prong work.

On one side, we're still looking to improve the shell script (and to correct bug if any). Then we will release Version 1.21, *"Sainte Honorine"*, by February 27th, that will contain a change from some licensed source to public source, and Version 1.2x, *"Sainte XXXX"*, shortly afterward that is nothing more than the English version of 1.21. Version 1.5, planned for June 89, will introduce the "AFUU scale" from a multidimensional analysis work performed on the results collected on each machine.

On the other side, we're working to broaden the SSBA's scope. We're looking at a set of *language* benchmarks. We're still at a very initial stage where we think that it might be a good thing to check the quality and the solidity of the compilers (say, the accuracy for numerical computation, the behavior of optimizers...) provided that the set would not be too complex for our level, which, after all, aims at giving each user a simple tool for the initial part of his or her performance evaluation work.

We're perhaps going to introduce a brother of SSBA: the SDBA who should be the database management system benchmark suite mentioned above. About this SDBA, we're not going to give any detail right now: the suite has just been finished a few weeks ago and not even yet been alpha-tested thoroughly; *(it might be a good idea to present it at the next EUUG conference?)*. While we can consider the SDBA suite to be somehow on-time, thanks to much laborious work from our friends at the SGBD group, ( *SGBD* is the

French acronym for *S*ysteme de *G*estion de *B*ase de *D*onnees, i.e. data base management system) we may not say so about another work targeted at creating a graphics benchmark suite. However we still think that some day not too far we'll be able to introduce this suite since lots of works are already well on track.

Then at the medium term, up to next spring (1990), with the coming of OSF1 and SystemV release 4, we'll be predictably busying running new validation tests: are we going to see whether the SSBA and its siblings run well under those new comers? or are we testing the conformance of these new comers to our existing software?

The long term around the SSBA, no more than at any other place else, does not need to be planned precisely. Yet we can mention some directions of interest raised throughout our working sessions: supercomputing, (parallelism?), realtime and multiprocessing.

The evolution of the SSBA depends on two main factors. The first one is our capability to fulfill the goals set forth here above. The second lies with the users and the reactions induced by the use of the SSBA.

Although the SSBA behaves, up to now, rather quite correctly, many things still stand ahead. The SSBA is no more on the launch pad, it has taken off: many of us estimate that it's pretty good on its way toward it's cruise speed. Whether it will succeed as all of us wish it does remains to be seen, but we're quite committed to care about it for some more times.

At the Benchmark group, we think that the SSBA is one pleasant yet useful way to further the promotion of UNIX: we hope you to enjoy it and continue using it; just feed us back with your comment(s) or contribution(s) to keep it, enjoying and serving you as long as possible.

# Attachment 1

SUITE SYNTHETIQUE DE BENCHMARKS DE L'A.F.U.U. (S.S.B.A. 1.2)

Association Francaise des Utilisateurs d'UNIX
11, rue Carnot
94270 Le Kremlin-Bicetre, France
Tel: (1) 46 70 95 90+

"En toute chose, il n'y a qu'une maniere de commencer
quand on veut discuter convenablement: il faut bien
comprendre l'objet de la discussion."
Platon.

## MANIFESTE

La SSBA est le fruit des reflexions du Groupe de travail Benchmark de l'AFUU (Association Francaise des Utilisateurs d'UNIX). Ce groupe, constitue d'une vingtaine de membres actifs, d'origines diverses (universite, recherche publique et privee, constructeur, utilisateur final), s'est donne pour but de reflechir sur le probleme de l'evaluation des performances des systemes informatiques, de collecter un maximum de tests existants de par le monde, d'en dissequer le code et les resultats, d'en discuter l'utilite, d'en figer des versions et de les fournir sous la forme d'une bande magnetique avec commentaires et procedures diverses.

Cette bande est donc, a la fois, un outil simple et coherent pour l'utilisateur final comme pour le specialiste, permettant une premiere approximation claire et pertinente de la performance et serait aussi susceptible de devenir un standard dans le monde UNIX(tm). Ainsi est nee la SSBA (Suite Synthetique de Benchmarks de l'AFUU) dont vous voyez ici la version 1.2 (dite de la "Sainte-Odile").

## UNE DEFINITION

Programme Benchmark: "Un programme informatique standard utilise pour tester la puissance de traitement d'un ordinateur par rapport aux autres". Un programme benchmark peut etre concu pour evaluer des problemes generaux, appeles problemes benchmark, comme le traitement des fichiers, le tri ou les operations mathematiques, ou pour evaluer un probleme plus specifique qui tiendra plus compte de l'utilisation de cet ordinateur. La performance, comme la vitesse de traitement, peut etre evaluee et comparee a celle des autres ordinateurs testes avec le meme programme. Ce processus est appele un test benchmark et peut etre utilise comme un outil d'aide a la decision lors de l'acquisition d'un ordinateur.

## AVERTISSEMENT

L'evaluation des performances est une necessite dans le domaine informatique comme dans tout autre. C'est un probleme d'autant plus delicat qu'il n'a pas de solution mathematique exacte et on doit donc travailler par approximations. Cet etat de fait a conduit, malheureusement, a des exces ou a considerer ce probleme comme de la cuisine ou de la "bidouille". Un des moyens d'approche est la mise au point de tests benchmarks.

Il est facile de critiquer les benchmarks et de dire qu'ils ne signifient rien. Par experience, il s'avere que ceci est generalement le fait de constructeurs dont les machines n'obtiennent pas de bons resultats aux benchmarks. Le tout est de savoir de quoi l'on parle. Il ne faut pas reduire une machine a un seul chiffre comme cela a ete trop souvent le cas (le mips par exemple), mais il faut essayer d'en donner une image multidimensionnelle en utilisant plusieurs tests specifiques. C'est l'approche qui nous a guides ici.

La solution qui parait immediatement ideale pour l'utilisateur final est de prendre son application et de la faire executer, telle quelle, sur un ensemble de machines. Le probleme est que des que l'application evolue, il faut tout recommencer; d'ol la necessite de trouver des tests caracterisant certains types de problemes qui vont apparaitre dans la plupart des applications finales. L'usage de ces tests risquant d'etre perverti, il faudra en indiquer soigneusement les limites mais aussi les qualites reelles.

Il n'y a jamais eu et il n'y aura jamais de benchmarks capables de representer completement la charge de travail d'un systeme informatique en environnement reel (de nombreuses etudes visent a l'approcher). Cela depend d'un nombre important de facteurs et c'est une evidence de le signaler. Partant de cette constatation on peut rester assis en se disant que l'on n'arrivera jamais a rien... Nous avons choisi d'agir (de toute facon quelqu'un d'autre l'aurait fait) en fournissant, conscients de toutes les limites de notre approche, un outil pratique et aussi valable que possible.

Il existe actuellement environ 200 tests references dans le monde. On peut les classer en 3 grandes categories:

1) Les benchmarks dits "standards": Dhrystone, Whetstone, Linpack, Doduc, Byte, Spice, Euug, Stanford, Musbus, Livermore, Los Alamos..., publies dans des revues ou sortis de chez les gros utilisateurs (General Electric, Exxon...), dont le code a ete largement diffuse et parfois modifie, generalement admis par l'ensemble de la profession mais a propos desquels il regne la plus grande confusion quant aux versions, aux resultats, aux interpretations et a l'utilisation que l'on peut en faire.
2) Les benchmarks dits "commerciaux": AIM, Neal Nelson, Uniprobe, Workstation Laboratories..., largement documentes, soumis a des licences dont les tarifs sont generalement tres eleves, fournissant un service professionnel, mais donnant, en general, a peu pres le meme type d'informations que les tests precedents, avec des connotations plus applicatives (AIM: systeme, Neal Nelson: bureautique, Workstation Labs: technique) et un emballage soigne pour le decideur. Ces tests n'en sont pas moins faillibles.
3) Les benchmarks dits "internes" utilises par certains constructeurs (IBM, DEC, HP, ATT, Olivetti, NCR, Texas, pour ceux qui ont fait l'objet de presentations) pour simuler des charges de travail (pseudo-utilisateurs effectuant des taches classiques) et calibrer ainsi leurs systemes.

Nous avons recupere ou pris connaissance de la majorite de ces benchmarks dans les 3 categories. Nous les avons examines sur le fond et la forme. Nous les avons executes sous diverses conditions afin de les valider.

Les tests selectionnes ici l'ont ete apres mure reflexion et donnent a notre avis une image d'ensemble de la machine aussi complete que possible, dans un souci de rigueur et de portabilite.

Il y aura des evolutions vers des domaines plus applicatifs: graphique, temps reel, transactionnel, SGBD, langages,... La version 1.21 comportera des commentaires en anglais et en francais et sera portee dans les news sur le reseau USENET. La version 1.5 verra l'apparition de l'indice AFUU. La version 2.0 verra l'ajout de nouvelles fonctionnalites.

L'interet d'une telle demarche est qu'elle soit largement diffusee et adoptee par le plus d'utilisateurs possible. Envoyez-nous vos resultats, decouvertes de bugs, commentaires ou insultes a:

afuu_bench@inria.inria.fr

Les seuls resultats publies le seront sous le controle de l'AFUU, pour eviter de tomber dans le travers des boites a lettres electroniques ol tout le monde vient poster des resultats parfois douteux.

La SSBA version 1.2 caracterise dans un systeme informatique sous UNIX ou ses derives:

- la puissance CPU, la vitesse de traitement, les capacites de calcul;
- l'implantation du systeme UNIX dans son ensemble, le systeme de fichiers;
- les compilateurs C et Fortran, les capacites d'optimisation;
- les acces et la gestion de la memoire, la performance des caches;
- les entrees-sorties du disque, la performance du controleur;
- le comportement en multi-utilisateur vis a vis de taches significatives;
- un ensemble de parametres inherents au systeme.

## COPYRIGHT

La SSBA est copyright AFUU. C'est un logiciel "domaine public". Vous pouvez l'utiliser comme bon vous semble mais en aucun cas publier des resultats dans un organe de presse quelconque ou lors d'une presentation publique, sans l'accord prealable de l'AFUU qui en garantira la conformite, ceci uniquement dans le souci d'eviter les problemes connus observes avec d'autres tests.

L'AFUU degage toute responsabilite quant aux consequences du passage de la SSBA.

Les procedures, les commentaires, une partie du code, ainsi que l'architecture de l'ensemble et la mise au point ont ete concus et realises par Philippe Dax (ENST), Christophe Binot (LAIH) et Nhuan Doduc (Framentec).

## STRUCTURE DE LA SSBA

La SSBA comprend 12 benchmarks selectionnes comme cela a ete dit plus haut. Ces 12 benchmarks sont, en partie ou en totalite, issus des benchmarks suivants: le mips/Joy, le Dhrystone, le Whetstone, le Linpack, le Doduc, le Byte, le Saxer, l'Utah, le Mips, le Test C, le Bsd et la Musbus.

La SSBA est organisee en 16 repertoires sur un meme niveau. A chaque benchmark est associe un repertoire dont les noms sont respectivement: mips, dhry, whet, linpack, doduc, byte, saxer, utah, outils, testc, bsd, musbus. Deux repertoires supplementaires ssba et config servent respectivement a lancer la SSBA et a analyser la configuration de la machine et du systeme. Ces 14 repertoires font partie de la distribution initiale. Deux autres repertoires, install et results, sont construits durant la phase d'execution de la SSBA. Le repertoire install contient des fichiers parametre qui refletent le type du systeme UNIX et le choix, par la personne qui lance la SSBA, des compilateurs et des chaines d'options de compilation. Le repertoire results contiendra les resultats et l'historique (trace) des operations effectuees durant l'execution de la SSBA.

L'ensemble de la SSBA, a son etat initial, comprend 226 fichiers repartis dans les 14 repertoires. La SSBA utilise 94 programmes source. Elle genere 92 resultats bruts et une description des principaux parametres systeme de la machine. Le volume occupe est de l'ordre de 1.4 Mega-octets. Il est conseille de disposer au moins de 15 Mega-octets sur un meme systeme de fichiers ol sera implantee la SSBA et de prevoir aussi un /tmp d'au moins 4 Mega-octets pour la faire tourner correctement.

Il est egalement necessaire d'avoir un systeme UNIX ou derive qui supporte imperativement les 43 commandes suivantes:

awk, bc, cat, cc, cd, chmod, comm, cp, date, dc, df, diff, echo, ed, expr, f77, grep, head, kill, lex, ls, make, mkdir, mv, nroff, od, ps, pwd, rm, sed, sh, sleep, sort, spell, split, tail, tee, test, time, touch, wc, who, yacc.

Commandes facultatives:

banner, hostname, logname, lpr, more, pr, shar, tar, uname, uuname.

Pour information, la SSBA 1.2 s'execute completement en 2h00 sur une machine "4 mips" non chargee.

Le profil fonctionnel des differents programmes est le suivant:

| CPU | SYSTEME | CALCUL | MEMOIRE | DISQUE | CHARGE |
|---|---|---|---|---|---|
| dhrystone | outils | doduc | seqpage | disktime | multi.sh |
| mips | forks | whetstone | randpage | saxer | work |
| bct | execs | linpack | gausspage | iofile | |
| testc | csw | float | iocall | | |
| | signocsw | fibo24 | bytesort | | |
| | syscall | | | | |
| | pipeself | | | | |
| | pipeback | | | | |
| | pipediscard | | | | |

Outre les fichiers de code et de donnees pour chacun des benchmarks, dans chacun des 14 repertoires il existe un jeu de 5 fichiers supplementaires. Si bench represente le nom fictif de l'un des benchmarks, pour un repertoire donne on aura:

| | |
|---|---|
| bench.doc | les commentaires relatifs a ce benchmark, |
| bench.files | la liste des fichiers mis en oeuvre, |
| bench.mk | le fichier Makefile, |
| bench.cf | le script-shell de configuration, |
| bench.run | le script-shell d'execution. |

Au cours de l'execution de la SSBA d'autres fichiers peuvent etre crees:

| | |
|---|---|
| bench.h | header cree par bench.cf, |
| bench.jou | trace des compilations, |
| bench.log | trace des operations lors de l'execution, |
| bench.kill | script-shell pour tuer le benchmark courant, |
| bench.tmp | fichiers temporaires, |
| bench.res | resultats locaux a ce benchmark. |

Que l'on se place au niveau general (repertoire ssba) ou au niveau local de chaque benchmark, les fichiers Makefile (bench.mk) comportent tous les memes cibles:

| | |
|---|---|
| conf | configuration, |
| compile | compilation, |
| run | execution, |
| sizes | taille des executables, |
| clean | suppression des objets, executables, journaux, resultats..., |
| readme | visualisation des documentations, |
| print | impression des Makefiles et des script-shell, |
| printall | impression de la totalite des sources, |
| tar | archivage format tar, |
| shar | archivage format shell-archiver. |

## MISE EN OEUVRE GENERALE

1 – Se placer sous le repertoire ssba, etre de preference sous sh,

2 – Editer le fichier ssba.def qui contient les commandes par defaut. Chaque entree de ce fichier est constituee de 2 champs separes par un ':' (deux points). A droite du ':' placez-y la commande ou la chaine d'options appropriee a votre systeme ou celle de votre choix (mais aucune option d'optimisation).

Par exemple:

| | | |
|---|---|---|
| C Compiler | :gcc | (par defaut : cc) |
| Options C Compiler | : | (par defaut : rien) |
| Fortran Compiler | :f77 | (par defaut : f77) |
| Options Fortran Compiler | :-f68881 | (par defaut : rien) |
| Printer | :laser | (par defaut : lpr) |
| Pager | :pg | (par defaut : more) |

Si le second champ est vide, c'est la valeur par defaut qui est prise (voir ci-dessus).

3 – Taper la commande de lancement de la SSBA:

```
 fiche     (pour remplir la fiche signaletique)
ou
 ssba.run&
```

4 – Se placer dans le repertoire results pour verifier le bon fonctionnement de la SSBA en visualisant le fichier ssba.log, journal historique des operations.

5 – En cas de probleme il est possible de tuer l'ensemble de la SSBA en tapant ssbakill (situe sous le repertoire ssba), si cette commande est sans effet essayer ssba.kill.

6 – Attendre la fin d'execution de la SSBA et analyser les resultats dans le fichier ssba.res, dont un condense se trouve dans le fichier synthese, situes tous les deux dans le repertoire results.

7 – Imprimer les fichiers ssba.log, ssba.res et synthese du repertoire results, puis les communiquer a l'AFUU.

## MISE EN OEUVRE LOCALE

L'execution locale d'un benchmark est possible a condition que les fichiers des repertoires install et config aient ete crees au prealable, soit par ssba.run& soit par ssba.ini (voir l'en-tete de ce fichier), suivi de unix.sh, suivi de make conf, et enfin dans config de make compile -f config.mk.

En effet, pour que tout benchmark puisse s'executer normalement, il est necessaire d'avoir calcule le parametre de granularite du temps 'HZ', puis de l'avoir introduit dans les outils de mesure du temps: config/chrono et config/etime.o.

Cette phase de pre-initialisation construit des fichiers qui seront utiles pour la plupart des benchmarks: install/hz.h, install/signal.h, install/*.cmd et install/*.opt.

Une fois cette operation preliminaire effectuee, la marche a suivre, pour executer localement un benchmark, est la suivante:

| | |
|---|---|
| make conf -f bench.mk | configuration |
| make compile -f bench.mk | compilation |
| make run -f bench.mk | execution |
| make sizes -f bench.mk | tailles |

## CONSEILS

- Pour les supercalculateurs et autres machines avec des options Fortran specifiques les rajouter directement sous editeur dans les fichiers whet.mk, linpack.mk et doduc.mk a la ligne LOCALOPT.
- Ne pas chercher a modifier quoi que ce soit, la SSBA pourrait ne plus tourner et de toute maniere nous nous en apercevrions a la sortie des resultats... Et puis c'est vilain!
- Ne pas paniquer devant les temps d'execution ou certaines erreurs.
- Si le "doduc" ne se compile pas il faut peut-etre augmenter la taille de la table des symboles: rajouter -Nn4000 a la ligne FFLAGS du fichier doduc.mk.
- Vous pouvez executer sans crainte la SSBA sous root (conseille pour les systemes qui n'autorisent pas plus de 25 processus par utilisateur), mais il serait preferable de l'executer en mode utilisateur normal, au total, en phase de simulation 8 utilisateurs la SSBA genere 46 processus.
- Pour les audacieux, dans le repertoire musbus, le fichier musbus.run, a la ligne nusers=8, il est tout a fait possible de mettre 16, 32 ou ce que vous voulez, il faut simplement s'assurer que le noyau de la machine le supportera.
  Dans le repertoire bsd, le fichier bsd.run, le parametre 1500 apres le "-p" de seqpage, randpage, gausspage, peut etre modifie en fonction de la memoire virtuelle disponible, par exemple 10000 pour 10 Mega-octets.
  D'autres parametres peuvent etre modifies tres simplement. N'hesitez pas, la SSBA est votre outil.
- Pour vos achats, faites executer la SSBA sur une machine fraiche chez le constructeur et examinez avec lui les resultats obtenus.
- Et surtout:

PRENEZ BEAUCOUP DE PLAISIR

Que la force soit avec vous!

Christophe Binot
Philippe Dax
Nhuan Doduc

Le 22 Decembre 1988

# An Interactive UNIX Spelling Corrector

*Philip M Sleat*
*Sunil K Das*

City University London
*phils@cs.city.ac.uk*
*sunil@cs.city.ac.uk*

*ABSTRACT*

When designing interactive information processing applications there are two important properties of the data files that must be considered. The first of these is the size of the file. The second is speed of access. An investigation was conducted into the most appropriate structures for the storage of dictionaries. Sequential, binary and hashing techniques proved to be inferior to tree-based methods which permitted data compression.

Tree structures for storing a dictionary file were used in writing an interactive spelling corrector because:

- tree-based storage methods were the only method to introduce data compression;
- access times were found to be better in comparison with the sequential search and comparable with binary chop or hashing; and
- it was discovered that there existed inherent spelling correction within the tree.

## 1. Introduction

A useful tool to aid document preparation within the UNIX programming environment would be an interactive, spelling *corrector*. The spelling *checker* currently supplied with UNIX systems is not interactive and could be more "user friendly". The user invokes *spell* by supplying the name of the document to be checked. The output is those words which are not in the dictionary and hence assumed to be misspelled. This spelling checker is cumbersome to use. The user must take the generated list of misspelled words and use a standard text editor to locate each one in the document to correct them. The spelling checker does not supply a list of suggested correct spellings; if the user is unsure of the correct spelling of a word, it must be looked up manually.

Files for interactive, information processing applications are often large. Ideally, a file should hold data in as compact form as possible. It is desirable that the structure of a file aids efficient searching to extract pertinent data. Access times should be as small as possible so that a given record in a file may be retrieved quickly. Moreover, the organisation should facilitate adding new data. Very often, a trade-off is made between the compactness of a data file and the speed with which records can be accessed. Generally highly compact data files have slow access times; the compressed nature of the file making searching algorithms complicated.

There are three simple, well known methods for storing and accessing data files [Knuth 1973]:

- linear, sequential storage and access;
- sequential storage, binary chop (logarithmic) access;
- storage and access by hashing techniques.

A fourth method, particularly appropriate to the storage of dictionary files, is based upon tree structures [Sussenguth 1963], and out-performs the other three in terms of economy of size and speed of access [Stanfel 1970]. The complete file is stored in a binary tree giving all the advantages of the binary chop. This tree-based, storage method has the advantage that the tree is a more compact data structure than the equivalent "flat" dictionary file; it is possible to read the whole tree into main memory to do the search. It is for this reason that this tree-based file is very quick to access.

This paper describes a student project [Sleat 1988] whose objectives were to:

- investigate and extend tree storage techniques to facilitate the compression of a dictionary file and speed up word access; and
- study the design of human-computer interfaces [Shneiderman 1987] to produce a "user friendly", interactive spelling corrector using the compressed dictionary tree.

A 36,000 word English dictionary was compressed from 320K to a dictionary tree of 117K; this compact file being just over one third the size of the original file. The spelling corrector, written in **C**, has proved both easy to use and efficient. The dictionary tree is quick to access; on an IBM PC average access times of 0.006 seconds per word have been achieved. Moreover, it was discovered that the dictionary tree has an element of "built-in" spell correction.

## 2. Dictionary Tree Structures

The tree terminology [Iverson 1962] used in this paper is illustrated in Figure 1. Letters A-K represent *nodes*, with A as the *root*. C and F are examples of a *leaf*. A-D-G-I is a *path* connected by three *branches* and therefore, has a *length* of three. The *filial* set of D is the set of nodes one path length away from D, namely {F, G, H}.

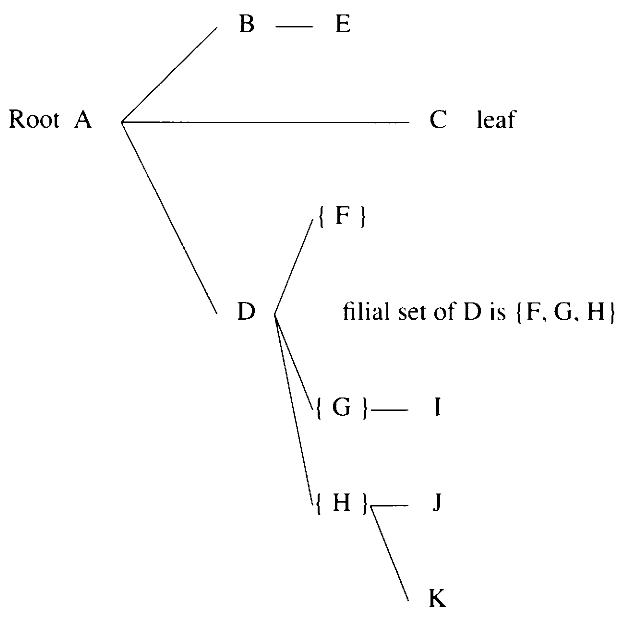


**Figure 1**: *Tree Terminology*

A convenient way to store an English dictionary using a tree structure is to have each letter of a word at a different node. Each of these letters then becomes one of the keys on which the words are scanned when checking if a given word is in the dictionary.

For some nodes, a node can be both at the end of a word and part way through a word. For example in Figure 2, the word "INSERT" is both a word in its own right, as well as a prefix for the words "INSERTED" and "INSERTING". Thus, to represent the end of a word diagrammatically, an asterisk has been used to flag the terminal letter of a word. The end of word is not necessarily at the end of a path, but all leaf nodes represent the ends of words.

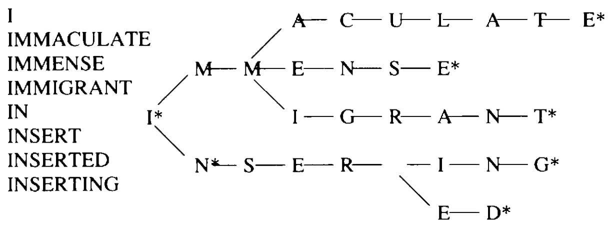


**Figure 2**: *Part of a dictionary and its tree structure*

As an example, how is the word "IMMIGRANT" searched for to see if it is in the dictionary (and hence spelled correctly)? Starting at the root "I" the filial set {M, N} would be scanned. Since "M" is the first letter in the filial set, this node's filial set is scanned looking for a match to the third letter in "IMMIGRANT", i.e. "M". The filial set of this "M" node is {A, E, I} so the "I" node is eventually located. From this point, the letters in "IMMIGRANT" can be checked off from the "G" onwards until the leaf at the end of the path is reached.

If the word being searched for is misspelled as "IMNIGRANT", it would be found that "N" was not in the filial set of the "M" node. Hence, the word is not in the dictionary tree and is potentially misspelled or requires adding to the addenda dictionary file.

## 2.1. Common Word Endings

In many tree structures, there are common subtrees which are stored many times, thus wasting space. For an English language dictionary tree, these subtrees manifest themselves as common word endings. For example, many English verbs can end with the suffix "ING". Other common word endings are "ATE", "TION", "ED". All of these common word endings are stored at the end of a path in the tree representation of a dictionary. The space occupied by a dictionary tree can be considerably reduced by replacing common word endings with a pointer into some predefined table as displayed in Figure 3.

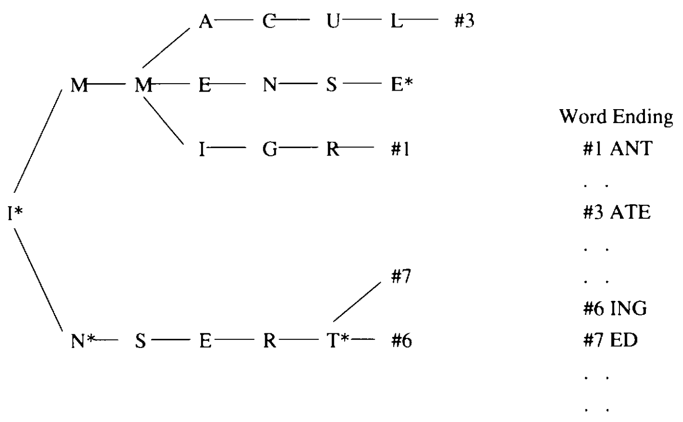


**Figure 3**: *Tree Structure Showing Word Ending Pointers*

The algorithm of Figure 4 was devised to build up a frequency table of word endings of up to four characters in length from the "flat" dictionary file. It could be used going forward in the dictionary file followed by going backward in an attempt not to "lose" a common word ending.

```
for (each word in the dictionary)
    if (word length > 4 characters)
       /* x is the character length of the word ending */
       for (x = 2; x <= 4; x++)
          if (word ending is in frequency table)
             frequency count ++;
       else if (there is space in the table) {
          insert word ending;
          frequency count := 1;
       } else {
          locate word ending with lowest frequency count;
          use this location for the new word ending;
          frequency count := 1;
       };
```

**Figure 4**: *Building a frequency table of common word endings*

Having generated a frequency table of common word endings, the most popular four letter word endings were chosen, followed by three and two letter word endings to give the best compression of the final dictionary tree. The frequencies for the three and two letter word endings had to be adjusted dynamically to reflect the fact that we had chosen some four letter word endings. For example, consider the word endings "TING", "ING" and "NG". Having selected "TING" to enter into the word ending table, the frequencies of "ING" and "NG" had to be adjusted because of the overlap.

A further possible source of saving space would occur if there are more common word beginnings than there are common word endings, and the words were stored backwards. A brief investigation into the English language established that there are more common word endings than beginnings. For the English language, it is sensible to store the words forward.

## 2.2. Ordering Nodes of a Filial Set

A major source of inefficiency in tree-based dictionary storage is the need to search the filial set of a node in a linear manner to see if a given character is included. Using the principle of *Zipfian* distribution on the nodes of filial sets would speed up this search. Zipfian distribution arranges the nodes of a filial set in order of greatest frequency. For example, suppose the most common and uncommon letters following the letter "A" in the English language are "N" and "J", respectively. The node "N" would be the first node in the filial set of "A" so that it would be checked first. The least frequently occurring letter "J" will be stored last in the filial set of "A". In this way, the most common letters are checked earliest to speed up the process of searching the filial set.

However, there is an inherent disadvantage associated with Zipfian distribution. If the filial set is stored alphabetically and the current letter in the filial set is higher in the collating sequence than the letter being searched for, then it is known that the letter is not in the filial set. With a filial set ordered according to Zipfian distribution, a search to the end of the filial set is necessary to discover that a given letter is not included. Therefore, using Zipfian distribution will be slower than alphabetic ordering for checking misspelled words. For correctly spelled words, a Zipfian based dictionary tree can be searched faster.

An algorithm was devised to sort an English dictionary according to a Zipfian distribution. For example, the following subset of words:

| | | |
|---|---|---|
| A | | A |
| AARDVARK | | ABOUND |
| AARDVARKS | | ABOUNDED |
| ABLE | would generate | ABOUT |
| ABOUND | | ABLE |
| ABOUNDED | | AARDVARK |
| ABOUT | | AARDVARKS |

because more words begin with "AB" than "AA" and more words begin with "ABO" than "ABL".

The algorithm for Zipfian sorting is as follows:

i Construct a dictionary tree from the dictionary file. As each node in the tree is passed through, increment a count associated with the node. This count is the frequency of use of the node.

ii Visit each node in the tree in turn (using a pre- order search). Sort the filial sets of each node according to the frequency counts.

iii Perform another pre-order search of the whole tree to re-construct the words in the tree. Write these words out to the Zipfian sorted dictionary file.

## 2.3. Reversing the Dictionary

Assume the word "INSERTING" has been misspelled as "INSTRTING". In suggesting a list of correctly spelled words, the wrong path of the dictionary tree would be examined. For example, words like "INSTRUCT" and "INSTRUCTION" instead of the correct word "INSERTING" would be retrieved. All of the letters up to and including the incorrect letter form the start of a legal word. With the misspelled word "IZSERTING", no suggested list would be generated at all. Therefore, when the letter in error is near to the end of a word, it is more likely that the correct spelling will be suggested.

If storage space is less of a constraint two copies of the compact dictionary tree could be retained. In the application described, this would still take up less space that the original dictionary used by **spell**. The first dictionary would be stored as a dictionary tree. The second dictionary would have all the words of the "flat" dictionary file reversed and re-sorted before compression. In this way, the words are stored in the dictionary tree backwards.

With words such as "IZSERTING", the position of the incorrect letter is known exactly. So, if a spelling mistake occurs at the beginning of a word, the word can be reversed and the **backwards** dictionary examined to generate a list of suggested correct spellings. When a spelling mistake occurs toward the end of a word, the normal copy of the dictionary tree can be examined whereas if it is near the middle of the word, the best strategy would appear to be to generate two lists of suggested correct spellings. The first list comes from using the misspelled word and the **forwards** dictionary while the second from using the reverse word and the **backwards** dictionary. These two lists can then be merged. This process should generate the correct spelling in the majority of cases.

## 3. The Spelling Corrector

The following **C** programs have been written:

zipdict.c

This program will take a "flat" dictionary file and re-sort it according to the Zipfian distribution.

supcom.c

This program will take a file sorted according to a Zipfian distribution and compact it according to the principles of the tree based storage method with common word endings, variable length pointers and Zipfian sorted filial sets.

examine.c

The algorithm to search the compact dictionary tree and detect the end of word boundaries has been implemented as a set of **C** functions which are in the program **examine.c**. To use the functions, an initial call must be made to *set_up_dict()*. This simply reads the dictionary tree into memory from the file specified by the string DICT_NAME.

When the dictionary has been set up, the function *correctly_spelt()* can be used. This function is passed a string of up to 30 characters representing word in upper case to be checked; e.g. *correctly_spelt("HELLO")*. The function returns 1 if the word is correctly spelled or 0 otherwise.

The spell correction algorithm also appears in the file **examine.c** as a function *find_correct_spelling()*. Three parameters are required: a string representing the incorrect word; a pointer into an array of strings to hold the suggested spellings; and the size of this array. For example the statement:

*find_correct_spelling("HELLJ", sugg_spell, 30)*

would fill the array sugg_spell with up to 30 suggested spellings.

## 3.1. The User Interface

The spelling corrector has a full screen presentation of the user's document. The document is displayed a page at a time under the control of the user. Each word that is unknown to the spelling corrector (*ie* those that are not in the dictionary), is displayed in *reverse video*. Therefore, the user can locate misspelled words quickly and decide upon the correct spelling based on the context of the complete sentence.

The page from the user's document is displayed in the middle 23 lines of the screen. The top line contains a copyright message. The bottom line contains a *function bar* giving a summary of the current keys that can have effect. The function bar changes according to the operation *mode* of the spelling corrector. The first mode is *word select* mode in which the cursor keys are used to move around the screen. The second mode is *word correct* mode in which the user can correct a word based on suggested correct spellings from the program. There are two function bars, one for word select mode and one for word correct mode. In word correct mode there are two ways in which a misspelling can be corrected. The first of these is where the user types in the correct spelling directly. The second is where the spelling corrector displays a list of suggested correct spellings from which the user can select the appropriate word.

## 3.2. Addenda Dictionaries

In a single user PC type environment, each user would have his or her own copy of the compact dictionary tree. Each user would be responsible for the maintenance of the dictionary. However, in a multi-user UNIX environment it makes no sense for each user to have a separate copy of the dictionary. Multiple copies of the dictionary would waste storage space.

Using a common dictionary implies its maintenance must be the responsibility of one person. Therefore, the interactive spelling corrector does not contain a facility for the user to add words to the main dictionary. If anyone could add words to the dictionary, there would be a danger that it could eventually contain misspelled words. Instead a user can invoke the spelling corrector with an optional **addenda** file. This will contain correctly spelled words not in the dictionary, for example names or acronyms. The spelling corrector will then check this addenda dictionary as well as the main dictionary. At any point the user may add a misspelled word to the current addenda dictionary by moving the cursor to the word and pressing the "A" key. Any future occurrence of this word in the document will not then be flagged as incorrect.

## 4. Conclusions and Acknowledgements

Techniques for file storage and access based on trees are definitely useful for certain types of file. The English language dictionary is a prime candidate for compression in this manner. Indeed any file that has many records with repeated keys would be suitable for storage in a tree. Likely additions to the tree based storage method such as the grouping together of all common subtrees and ordering of the keys within the filial sets were considered. Both of these additions gave significant improvements over the original tree structure.

As a practical example, an efficient implementation of a compressed English language dictionary has been coded. By careful examination of the source dictionary a compact representation of tree pointers was used. The compact dictionary tree occupied only 117K which in comparison to the original size of 320K for a 36,000 word dictionary file is a good achievement. It can be shown that this storage method was more suitable for dictionaries than more traditional methods. Not only did the structure lend itself to very fast access times but also the generation of correct spellings was easy using the nature of the tree. This dictionary is extremely quick to access; the presence or absence of a given word may be checked in, on average 0.006 seconds (on an IBM PC in the C programming language).

This compact dictionary was then used as the basis of a fully interactive, user-friendly, spelling corrector for UNIX and PC based systems. The UNIX version has been *termcapped*. The primary goals of the spelling corrector were efficiency of dictionary storage, efficiency of dictionary access and ease of use. The first two were achieved by extending the tree structured techniques mentioned above. The latter was achieved by careful consideration of the human-computer interface, the screen display and user input. The spelling corrector has proved a useful tool and is very easy to use.

The authors would like to thank the following members of City University Computer Science Department: Mohammad Nejad-Sattary and Aarron Gull for beta testing the spelling corrector and making many constructive suggestions to improve the HCI, and David White who provided the original idea for the project and for his academic guidance.

## 5. References

[Iverson 1962]
Iverson, KE, *A Programming Language*, Wiley, New York, 1962.

[Knuth 1973]
Knuth, D E, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Massachusetts, 1973.

[Shneiderman 1987]
Shneiderman, B, *Designing the User Interface*, Addison-Wesley, Reading, Massachusetts, 1987.

[Sleat 1988]
Sleat, P M, *Storage Retrieval Methods for Interactive Spelling Checkers*, Report - City University Computer Science Department, London, April 1988.

[Stanfel 1970]
Stanfel, L E, "Tree Structures for Optimal Searching," *Journal of the ACM, Vol 17, No 3*, July 1970

[Sussenguth 1963]
Sussenguth, E H, "Use of Tree Structures for Processing Files," *Communications of the ACM, Vol 6, No 5*, May 1963

# Context-Reflecting Pictures of a Database

*Agnes Hernadi*
*Zalan Bodo*
*Elod Knuth*

Computer & Automation Institute
Hungarian Academy of Sciences
Budapest

*ABSTRACT*

An experimentally implemented unusual database interface with a new idea of context managing mechanism is introduced to make multi-contextual data dialogues intersession resident and readily responsive to appropriate changes of the database. Dialogue contexts, called pictures, reflect the real data, so their universe is a complete information model (to be mapped to the database scheme). Moreover, pictures are the only means by which database access is brought about. A picture can be regarded as an entry form for typical data input, or a transient portion of information on something, or a detailed report on something to be kept safe, or a part of the database scheme from a given point of view, or a distinctive query specification etc. Operations on contexts are not associated with traditional database subfunctions (insert, delete, update etc.) but are all interpreted on pictures and provide all subfunctions in one pattern.

## 1. Introduction

We have focused on database applications requiring direct, flexible interaction to support ad hoc transactions over frequently changing and sometimes even improperly defined schemes in contrast to routine traditional database applications which handle relatively fixed (stable) database schemes, transactions, report structures etc.

A database interface inspired by office-, management- and personal information systems recently coming into view (such as Hyper-systems, the remarkable phenomenon of Macintosh etc. [3], [5], [7], [8]) has been experimentally developed at our institute for AT&T UNIX PCs.

While coping with the problems of a highly flexible user friendly man/machine interface facility we found it was possible to conceal the traditional concepts of schemes, data definition languages, data manipulation languages etc. and to provide a unified tool simultaneously serving various purposes for *entering and updating data, query interpretation, report generation* and *scheme manipulation* [4], [6].

In fact this interface facility supports simultaneous usage of multiple views or data contexts moreover makes these contexts intersession *resident, sensitive* to the current alteration of the database and *the only tool* for accessing the database.

Last but not least this facility is easy to understand, to learn and putting it to use is quick, requiring no programming knowledge.

## 2. On the Data Model of the Experimental System

We have developed an experimental system called Cooperative Databases (Co DB) [1]. This relies on an ultimately simple data scheme. In order to allow us to keep our attention intently fixed on the problems of that interface facility we have chosen a completely unsophisticated and practicable data model. Namely

- We apply a binary relationship model [2] with no subtyping, however, relationships are non-directed many-to-many.

- Types and relationships are maintained automatically by entering their instantiation, and destroyed utterly on deleting their last instantiation. Instances of types are called atoms, and that of relationships are called connections.
- Two constituents are put together to form an atom that is to say a *<type>* class-description appointing the type to which the atom belongs and a *<value>* :

  <type><value>

  Both constituents are character strings although implementations may have restrictions laid on them. There are no arithmetical operations interpreted on values and for the time being we don't even plan to introduce them.
- A connection is an unordered pair of atoms belonging to a particular relation, so three constituents are put together to form a connection like a *<relname>* and an unordered pair *(<type1><value1>, <type2><value2>)*. So it seems a relation is made up of a *<relname>* taking the place of *role* and an unordered pair of existing types *(<type1>, <type2>)*.

  Relation names are – they may even be empty ones – character string objects which are unique for any given pair of types. Two binary relationships are considered identical if and only if all three of their corresponding constituents are identical (disregarding the order of types).

  A type might as well be related to itself, and an atom might be a constituent of any number of connections within a given relation. Two connections are considered to be identical if and only if all three of their corresponding constituents are identical (neglecting the order of atoms). Accordingly the same pair of atoms might be connected in as many relations as one could desire and still appearing once at most in a given relationship.

## 3. The Way of Displaying Data and Context

We believe it helps to view a binary relationship as a paragraph of two lines marked by the indentation of the second one.

Accordingly the first line displays an atom or a type and the second one the related atom or type preceded of course by the relation name if it is not actually an empty string. So a relation *<relname>(<type1>, <type2>)* or a connection *<relname>(<type1><value1>, <type2><value2>)* may appear in either form shown by Figure 1.

```
(i)     type1:
             [relname:] type2:
    or
        type2:
             [relname:] type1:


(ii)    type1: value1
             [relname:] type2: value2
    or
        type2: value2
             [relname:] type1: value1
```

**Figure 1:** *Equivalent ways of displaying (i) a relation and (ii) a connection*

As an example let us explore a database that records gods, heroes and mythical subjects of Scandinavian and Teutonic mythologies. To represent this we select three types such as *god*, *armed_with* and *takes_part_in*, and two sorts of relations *R1* for the relationship between *god* and *armed_with*, and *R2* for the relationship between *god* and *takes_part_in*. Information about god Odin should appear in one of those forms in Figure 2, depending on our taste or purpose. We will see later, that any number of indentations are allowed.

```
(i) god: Odin
      [R1:] armed_with: Gungnir the spear
      [R2:] takes_part_in: cosmogenesis


(ii) god: Odin
      [R2:] takes_part_in: cosmogenesis
      [R1:] armed_with: Gungnir the spear


(iii) armed_with: Gungnir the spear
      [R1:] god: Odin
        [R2:] takes_part_in: cosmogenesis


(iv) takes_part_in: cosmogenesis
      [R2:] god: Odin
        [R1:] armed_with: Gungnir the spear
```

**Figure 2:** *Alternative reflections of the same information*

Notice that no matter how we name *R1* and *R2* they remain redundant and even disturb us in understanding the represented connections. If both relation names were empty strings each it would be quite similar to the traditional way of jotting. That's the reason why we allow relation names to be empty strings.

## 4. Pictures as the unified tools of interaction

A *picture* consists of an arbitrary number of hierarchically indented lines displaying atoms, types and relation names occasionally. More formally a picture is a forest of *picture lines* with the definition of

```
<picture line> := [<relname>:] <type>: <domain>
<domain> := BLANK | <value> | <expression>
```

where <expression> is a selection criteria (e.g. a regular expression in terms of UNIX), and BLANK stands for *any* or *all* (no selection criteria).

In root lines no <relname> may appear. In non-root lines however theoretically a <relname> always appears at the very most it is empty (theoretically present but invisible).

In a manner consistent with the above definition each picture line has a unique *parent line* unless it is a root line.

### 4.1. Validity and other characteristic states of pictures

Informally speaking a picture is called *valid* if all the types, atoms, relations and connections referred to by any of its lines exist.

A valid picture is *filled* if each indented line in it possesses the following property: if its parent line contains a value, then it enumerates all the values connected to the atom displayed in its parent line by the named relationship.

A valid picture is *saturated* if each line in it which has at least one indented line, also has all possible indented lines in this very picture.

An empty picture trivially possesses all of these characteristic states.

## 5. Operations on Pictures

Any kind of user action can be carried out by using the appropriate operations.

### 5.1. State Transformations

To carry out picture state transitions and report generation we provide four transformations each of which acts on the whole picture.

#### VALIDATE

All the types, atoms, relations and connections appearing in the picture spring into existence if they have not existed in the database (see definition of valid picture).

#### FILL

The picture is to be converted into a filled one (see definition of filled picture).

All the values fitting into a given place will be listed. In case of domain expression only values satisfying the expression will be included.

#### SATURATE

The picture is to be completed to become saturated (see definition of saturated picture).

#### EVALUATE

All feasible valid sequences satisfying every single domain specifications are to be determined.

Each value displayed on the picture is regarded as a restriction. Feasible pathes between two lines displaying atoms must fit on both ends. Reports are typically generated by this operation.

### 5.2. Operation modes

In order to reduce the number of *depicting* operations operation modes were introduced. These serve as distinctive marks of thc particular classes of effects. There are three operations modes:

| | |
|---|---|
| FREE | mode serves for temporal depicting with no effect on the database. |
| CHECK | mode is the default mode for all valid pictures. This mode serves the purpose to develop our view on data. Therefore in this mode no operation has update effect and operations violating the validity constraint are refused. |
| ENFORCE | mode provides the only way to alter the database. In this mode the VALIDATE transformation is called after each depicting operation the result of which violates the validity constraint. |

The FREE $\rightarrow$ CHECK mode transition is refused whenever the picture is not valid. Other mode transitions are never refused. The FREE $\rightarrow$ ENFORCE mode transition is equivalent to a call of the picture state transformation VALIDATE, and as such must be confirmed.

### 5.3. Depicting Operations

These operations act on the selected part, that is on a subtree, a line or a token of a picture. They may or may not change the database itself depending on the current operation mode, however, according to the *What You See Is What You Get* paradigm no invisible change may occur. The whole interaction is supported with forms and icons requiring no syntactic knowledge of the user.

On selecting a subtree, we speak about a weak subtree if no restriction applies to the selection. But if the remainder must still constitute a picture, we speak about a strong subtree, see Figure 3.

#### REMOVE(strong subtree)(option)

The selected subtree disappears recursively from the picture. In the ENFORCE mode data objects are to be deleted too. Table 1 lists the options and actions taken.

```
god: Tor
   [equated_with:] god: Donar
   [derivation:] Teutonic: Punra (Donner)
   meaning: god of thunderbolts
   armed_with: Mj"olnirthe  hammer
   [place_of:] residence: Trudheim
      [in:] Asgard: ___________________________
   [father:] of: ____________________          |
      [son:] god: Magni                |       |
      meaning: strong                  | (i)   |
      [son:] of: _____________________|       | (ii)
         [mother:] giantess: Jarnsaxa          |
      [son:]· god: Modi                        |
         meaning: fearless ___________________|
   [married_to:] goddess: Sif
```

**Figure 3:** *Example of (i) a weak and (ii) a strong subtree*

| Option | Actions taken |
|---|---|
| with root (default) | The whole selected subtree disappears. In the ENFORCE mode each atom included in the selected subtree is to be deleted with all their connections. Corresponding types and relations can be destroyed utterly. No other inductive effect is taken. |
| without root | It differs from the default option in that the root line of the subtree does not disappear. In the ENFORCE mode each atom displayed by the root lines of the disappearing part is to be disconnected from the atom displayed by the root line of the selected subtree. The corresponding relation can be destroyed utterly. |
| disconnect | The whole selected subtree disappears. In the ENFORCE mode the atom displayed by the root line of the selected subtree is to be disconnected from the atom in its visible parent line, if there is any. The corresponding relation can be destroyed utterly. |

**Table 1**

**CLEAR(weak subtree)**

All the domains in the selected subtree are to be made blank. Identical lines with no indented hierarchy are only to be displayed once. It never alters the database.

**MOVE(strong subtree)**

The selected subtree is to be moved. The subtree disappears from the source picture. This operation can be used in an inter-picture sense too. See PASTE for terminating a MOVE.

**COPY(weak subtree)**

The selected subtree is to be copied. The source picture remains unchanged. This operation can be used in an inter-picture sense too. See PASTE for terminating a COPY.

**PASTE(line)(option)**

This operation terminates a COPY or MOVE operation. The subtree to be copied or moved is inserted into the picture according to the option specified. In the ENFORCE mode database update requires confirmation. Table 2 lists the options and actions taken.

| Option | Actions taken |
|---|---|
| after (default) | The copied/moved subtree is to be inserted after the selected line (skipping of course all the lines marked by a longer indentation). The root line of this subtree is to be marked by the same indentation as the selected line. |
| before | The copied/moved subtree is to be inserted prior to the selected line, and its root line is to be marked by the same indentation as the selected line. |
| under | The copied/moved subtree is to be inserted immediately after the selected line. Its root line is to be marked by an indentation as compared to the selected line. |

**Table 2**

**ADD LINE(line)(option)**

A line is to be created and inserted into the picture according to the option specified. In the ENFORCE mode database update may occur.

Strings to be displayed in the inserted line should be entered through a form asking for them. Menus of tokens already entered into the database are available. Table 3 lists the options and actions taken.

| Option | Actions taken |
|---|---|
| after (default) | The created line marked by the same indentation as the selected one is to be inserted after the selected line (skipping of course all the lines marked by a longer indentation). |
| before | The created line marked by the same indentation as the selected one is to be inserted ahead of the selected line. |
| under | The created line marked by an indentation as compared to the selected one is to be inserted immediately after the selected line. |

**Table 3**

Suppose we have a picture including the portion of Figure 4. If we want to enter some other information about the god Odin let us say the derivation of his name right after the line displaying it, then we have to select (mark) either the line displaying the name of Odin and to ADD LINE under it, or the line displaying his arm and to ADD LINE before it, see Figure 5.

```
 . . .
god: Odin
 armed_with: Gungnir the spear
 takes_part_in: cosmogenesis
 . . .
```

**Figure 4:** *A portion of a picture*

If we want to display the name of god Tyr right after the information about Odin, we can select the line displaying the name of Odin and ADD LINE after it see Figure 5.

```
  .  .  .
god: Odin
 [derivation:] Old_Icelandic: odinn
 armed_with: Gungnir the spear
 takes_part_in: cosmogenesis
god: Tyr
  .  .  .
```

**Figure 5:** *Result of the two ADD LINE operations on the picture in Figure 4*

**UNFOLD(line)(option1,option2)**

Our view of information displayed in the selected line is to be opened out by revealing all the lines which could come up as indented ones according to the database's content. It never alters the database. Table 4 lists the options and actions taken.

| Option 1 | Actions taken |
|---|---|
| natural only (default) | If the selected line displays an atom then all the atoms connected to it should be displayed. If however the selected line does not display an atom, all the relations interpreted on the type in the selected line should be displayed. |
| full | All the relations interpreted on the type in the selected line are to be involved neglecting whether they have any connections referring to the atom in the selected line. |

| Option 2 | Actions taken |
|---|---|
| non-repeating | The parent line of the selected one is not to be displayed between the indented lines. |
| repeat parent | Even the connection or relation between the selected line and its parent line is to be involved. |

**Table 4**

For the indented lines of the selected one which are already in the picture the following rules apply:

- Already existing lines will not be repeated;
- If the selected line displays an atom and the already existing indented one displays a blank domain then this blank domain will be filled with appropriate values instead of repeating this latter line;
- If the indented line contains a domain expression, it remains unchanged, and an other indented line will be inserted with the same relation and type names displaying values or not depending on the selected line.

**ADD VALUE(domain)(option)**

A new atom of the type specified in the line of the selected domain is to be inserted. If the domain did not contain a value no option is offered. The required value will be inserted in that very domain, however, it must not contradict the selection criteria, if there is any specified.

If the selected domain already contains a value, the new value is to be inserted according to the option specified.

In the ENFORCE mode database update may occur.

The value to be displayed in the selected domain should be entered through a form asking for it. The menu of values already entered into the database are available. Table 5 lists the options and actions taken.

```
 .  .  .
god: Odin
 [derivation:] Old_Icelandic: odinn
 armed_with: Gungnir the spear
 takes_part_in: cosmogenesis
god: Tyr
 [derivation:] Teutonic: Tiwas
 [equated_with:] god: Tiu
                      Saxnot
                      Mars
 god_of: battle
 .  .  .
```

**Figure 6:** *After unfolding the line which displays the name of Tyr (with default options)*

| Option | Actions taken |
|---|---|
| after(default) | The line containing the new atom will be placed right after the indented hierarchy of the line displaying the selected domain. The skeleton of the indented hierarchy of the selected line if there is such a hierarchy at all, will be inserted after the line containing the new atom. This skeleton contains all relation and type names but domains remain empty. |
| before | The line containing the new atom will be inserted above the line displaying the selected domain. |

**Table 5**

```
                         .  .  .
god: Odin
 [derivation:] Old_Icelandic: odinn
 armed_with: Gungnir the spear
 takes_part_in: cosmogenesis
god: Balder
 [derivation:] Old_Icelandic:
 armed_with:
 takes_part_in:
god: Tyr
 [derivation:] Teutonic: Tiwas
 [equated_with:] god: Tiu
                      Saxnot
                      Mars
 god_of: battle
 .  .  .
```

**Figure 7:** *Result of the ADD VALUE operation with an argument displaying the name Odin and the option after*

**DELETE VALUE(domain)**

The selected domain must contain a value which is to be abandoned. In the ENFORCE mode the atom is to be deleted with all its connections. No type or relation can be destroyed, however.

**EDIT EXPRESSION(domain)**

A selection criteria is to be defined or modified. Editing the selected domain is refused if it contains a value (see EDIT TOKEN). No value may be specified (see ADD VALUE).

**EDIT TOKEN(token)**

One of the strings displayed in the selected line is to be altered either in the picture containing the selected token (FREE mode), or in the database, and in all pictures displaying this very string (ENFORCE mode). This operation is unavailable in the CHECK mode.

Respectively either the relation or the type is to be renamed or the value is to be changed keeping all the connections. Editing the selected domain is refused if it is either empty (see ADD VALUE) or contains an expression (see EDIT EXPRESSION).

## 6. Galleries

Our pictures are stored in a special directory called Gallery which supports transactions dealing with pictures.

### 6.1. Picture Qualification

Pictures stored in a Gallery are qualified but their quality can be altered any time.

- A *Sketch* is a picture which need not be valid. Pictures to be depicted in FREE mode, or having become invalid are always requalified to this quality. This is the quality of created pictures too.
- A *Property* is a valid picture which however can become invalid and requalified to Sketch as the database changes.
- A *Protected* picture may never turn invalid. Depicting operations in ENFORCE mode on any picture violating this restriction are refused.
- A *Master Piece* moreover may not be changed at all. Depicting operations in ENFORCE mode on any picture violating this restriction are refused.

Beside these there are two standard, read-only pictures in each Gallery. The picture *Types* contains all the existing types. The picture *Scheme* contains all the existing relationships. These pictures or any of their parts can be copied freely however.

### 6.2. Gallery Organization

A Gallery consists of two main parts

- the *Exhibition* in which all pictures are updated according to the EDIT TOKEN operations and checked up on being valid or not; and
- the *Archive* in which pictures are not maintained at all.

### 6.3. Transactions dealing with pictures

Each Gallery belongs to a single Co DB. On opening the Gallery its Exhibition-menu is displayed. At request the Archive-menu is displayed too but in a separate window. From these menus the user can access the picture transactions namely:

- Create Picture
- Open Picture
- Delete Picture
- Copy/Move Picture
- Rename Picture

- Requalify Picture

All pictures have to be created except the standard ones. Opening a picture the operations on pictures are available. From an opened Gallery any number of pictures can be opened simultaneously. The other transactions work roughly in a way as can be expected.

## 7. Implementation issues

As we have already mentioned, Co DB is experimentally developed for AT&T UNIX PCs. The implementation exploits

- The hierarchic file system of UNIX;
- The multiple window management capability supported by TAM routines; and
- The manipulation of abstract objects at the operation system's level provided by UA.

We manipulate two abstract objects at the operation system's level: the *Co DB database* and the *Gallery*.

Commands assumed to be applicable to all ordinary abstract objects of this level such as create, open, close, delete, move, copy, rename are also defined on both of these objects.

Apart from the fact that each Gallery refers to a single Co DB, any number of Galleries can be associated with a Co DB. On opening a Co DB the menu of Galleries associated with it is displayed.

## 8. Summary

We have expounded a new, non language oriented approach of interactive database interface. This approach by matching modern requirements gives naive users demanding database supply altering dynamically an easy-to-use tool to interact directly with database.

We introduced the concept of picture which are user friendly abstract objects, having no fixed structure. Their content is transient, and they serve as a unified tool for accessing the database.

Our approach also provides a consolidated mechanism to draw computer aided comparison between independent databases containing diverse data, and to settle database communication protocols aiding interaction between them.

## References

[1] Z. Bodo, A. Gal, J. Gyenese, A. Heppes, A. Hernadi, E. Knuth, P. Rado, L. Ronyai, *co DB – Common Base Definition, Reference Manual Version 8.*, Computer and Automation Institute, Hungarian Academy of Sciences, Budapest, December 1988

[2] G. Bracchi, P. Paolini, G. Pelagatti, *Binary Logical Associations in Data Modelling*, in: J. M. Nijssen (ed.), Modelling in Database Management Systems (Proc. IFIP TC2 Conference, Freudenstadt), North-Holland, Amsterdam, The Netherlands, 1976.

[3] A. E. Cawkell, (ed.), *Information Technology and Office Systems*, North Holland, 1986.

[4] A. Hernadi, Z. Bodo, E. Knuth, *A Different Interactive Interface for Database Management Systems*, Proceedings of the 11th Int. Seminar on Database Management Systems, Seregelyes, Hungary, October 3-7, 1988, pp. 85-94.

[5] F. R. Hopgood, et al. (ed.), *Methodology of Window Management*, Springer Verlag, 1986.

[6] E. Knuth, A. M. Vaina, Z. Bodo, A. Hernadi, *Beyond data crunching: A new approach to database interaction*, Proc. of the 3rd Austrian_Hungarian Informatics Conference on 'Beyond number crunching', Retzhof, Austria, September 14-16, 1988, pp. 91-104

[7] G. E. Pfaff (ed.) *User Interface Management Systems*, Springer Verlag, 1985.

[8] *SUN Microsystems, Programmer's Reference Manual for the SUN Window System* SUN Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

# A Simple Guide to Porting the X Window System

*Lee McLoughlin*

Department of Computing,
Imperial College,
London,
UK.
SW7 2BZ
*lmjm@doc.ic.ac.uk*

*ABSTRACT*

The X Window System is the *de facto* graphics standard for UNIX workstations. The authors of the X Window System have gone to considerable lengths to allow it to be ported easily.

The server provided on the X distribution tape has some presumptions about the target system. The further from this model the harder it is to port X.

This paper only gives guidelines about how to port X by explaining where the major problems are and pointing to examples in the sources of code to act as a template for your own port. Only version 11 of X is discussed.

The author is a researcher in graphics interfaces. He has ported X release 10 to the Whitechapel MG-1 and HLH Orion and ported X release 11 to the HLH Orion 1/05 and has advised on other ports.

## 1. Introduction

The X Window System has been adopted by most manufacturers of UNIX based workstations as graphics standard. There are many reasons for this but two of the major advantages of X are:

- A large base of available applications
- The standard distribution tape, though not public domain is under a very generous license, is highly portable. So it is considerably cheaper and easier to make X available on a workstation than most alternatives.

In a presentation of this size it is not possible to give more than an overview of the porting process. It is intended mainly to act as a guide to lead the reader through a complex system. You are expected to have access to the sources for X. Pointers are given to other papers to read to give a more detailed description of the porting process.

### 1.1. What is X?

X is a pixel based graphics system for workstations that uses a *client-server* model. X was developed at MIT and *X Window System* is a registered trademark of MIT.

It is pixel based in that all the graphics requests are specified in terms of pixels, which are the dots on the screen, not millimeters or some other machine independent quantity.

In some graphics systems each graphics program directly accesses the screen and scribbles on it. Under X only one program, the *server*, can do anything with the keyboard, pointer (normally a mouse) or screen. Client programs connect to the server, usually using some kind of TCP/IP mechanism, and request it to do things: draw lines, draw characters, read events and so on. Because a client can use TCP/IP to make connections it does not need to be on the same machine as the server.

## 2. What is an X Port?

The X Window System splits into two parts:

1. The server
2. The clients

A full port of X will include both parts. However is it meaningful to just port one part. If you were to design and build an X terminal which plugged onto an ethernet it would only need the server side. A mainframe would then only need the client side and some X terminals.

For sake of completeness this paper will cover both.

## 3. What does X Port to?

In theory the MIT distribution of X will port to anything with a C compiler. In practice this is not really the case. The MIT distribution is slanted towards a particular kind of workstations. The further your target is from that the harder it will be to port to it.

The ideal target to port to has the following characteristics:

- 4.2/4.3BSD Unix system
- The screen can be mapped into user memory space and changed just by writing to that memory.
- There is special hardware for the cursor that automatically tracks the mouse and the cursor image on the screen is held separate from the screen memory.
- The mouse and keyboard generate events which are stored in a circular queue. The queue is mapped into user memory space and reading an event removes it from the queue.

If you are porting to a simple monochrome system then there should be no problems. If you aiming at colour then you must bear in mind that the sample server is slanted towards screens with 32, or less, bits of colour and expects pseudo-colour (a colour table lookup is used) not true-colour (the pixel directly represents the colour).

This is not to say that X cannot be ported to system that do not match up with the above ideal machine, it will just be harder. In some cases a lot harder.

A much more subtle problem may be caused by having a high resolution screen. Since X is pixel based the output of most programs will only look right if the pixel size is close to that of the screen that it was originally written for. Screens with a resolution of about 90 dots per inch (DPI) ± 15 DPI should have no problems. Very high resolution screens will take a lot more work to support since either the clients will have to be changed or the server may have to scale all dimensions passed to it. Either would involve several months of work.

Many X applications also presume that you have a three button mouse even though there is no such requirement in the X server. This can be faked easily given a two button (pressing both down can represent the third) or given four or more buttons. If you only have a one button mouse then it would be best to try and get the manufacturer to add more.

## 4. What Do You Need?

Firstly you need a copy of the MIT distribution tape. If a new (minor) release of X has just come out then the people are MIT are likely to be very busy people. Regrettably they are also in America. Unless you too are in America this means dealing with customs and other such people who like to delay tapes. Fortunately most of the bigger archive sites in Europe should be able to supply you with X.

While you are getting this there are a few other bits to collect. Firstly the official patches to the standard distribution tape, these are normally available from the same place as you get your main X tape. You need to have access to either the X mailing list (contact *xpert-request@expo.lcs.mit.edu* ) or the X network news group *(comp.windows.x),* to find out whether you should install the official patches (there have been occasional problems).

Another useful item to find at the same time is the *Purdue Speedups* by Gene Spafford. These are a set of enhancements to the standard MIT distribution to increase its performance. Mostly they are aimed at a monochrome server – but it is hoped that they will be extended for colour as well. Again they should be available from a larger archive site.

Once you have all this you will require two major and costly resources. Firstly you will require the disc space to load down and work on X. X is big. To work on the basic server and clients you will require about 50 Megabytes – if you are neat and tidy and remove documentation after printing and clean up the client source directories once they are installed. An untidy person will need about 100 Megabytes.

The second resource is much harder to quantify. It is programmer time. As a programmer working on such a project you will require a detailed understanding of how the graphics hardware on the target works. You will also need to know a fair bit about the X server – but it is well documented. You will certainly need to be an experienced C programmer. You really must acquire these skills and knowledge before attempting a port.

If the target is fairly close to the ideal then it should only be about about one man month to have a system up and running (though not necessarily in a shippable state). If it is not a 4.*x*BSD system then add a few months, or so, to overcome all sorts of problems (name length in the C compiler, lack of TCP/IP to name but two). If the screen cannot be memory mapped then add several months as all the sample code presumes this and so little of it will be usable. If you do not have a hardware cursor you should be able to use the code for a software cursor provided by Sun Microsystems in their server. If the mouse and keyboard do not generate events into a memory mapped queue then add a month or so to fake up something similar – generally this is not very hard, just full of awkward details.

If you have some clever graphics hardware then add a couple of months to add in code to use it. But this can, of course, be done once the base release is up and running.

You will find that a good symbolic debugger is an essential. So is patience, an Orion 1/05 has a performance of over 5 Mips but even on that it can take over two minutes for `dbx(1)` to load in all the symbols in the X server.

## 5. Useful but not essential

I found it very useful to have access to a working X system. It allowed the clients to be tested before my own server was working. Once my own server was working clients could then be run to both and the screens compared.

A printed copy of the X documentation is also useful. The `troff` source for all the documentation available is on the standard distribution tape. Once printed it amounts to about 800 pages! Personally I bought a commercially printed set – depending on whose you buy, this is probably cheaper than printing a copy out yourself.

## 6. The First Steps

Once the tapes are read in and the patches applied, sit down and read the documentation. Since there is rather a lot of it and most of it is aimed at applications programmers and not at programmers porting X you should be able to get away with just reading (or at least having available for reference) :–

1. *Godzilla's Guide to Porting the X V11 Sample Server* by David Rosenthal, Adam de Boor and Bob Scheifler.
2. *Strategies for Porting the Xv11 Sample Server* by Susan Angebrannd, Raymond Drewry, Philip Karlton, Todd Newman and Bob Scheifler.
3. *Definition of the X Server Porting Layer* by Susan Angebrannd, Raymond Drewry, Philip Karlton, Todd Newman and Bob Scheifler.

The next step will be to get familiar with what is on the X tape. Some edited highlights are show in table 1.

You'll notice on wandering around the sources that wherever you find a `Makefile` there is a file called `Imakefile` beside it. This is a template with which the real `Makefile` is generated using `util/imake`. The global configuration for `imake` is in `util/imake.includes/Imake.tmpl` which is used to configure the makefiles on a per-target basis.

To add a new target machine to `Imake.tmpl` copy the format of the existing entries (they are all very similar). You will then need to add the machine specific macro file. Again you should be able to copy and lightly edit one of the existing macro files in the `includes` directory, such as: `util/imake/includes/Vax.macros`. All the various options are detailed in the `README` file in that directory

| | |
|---|---|
| `clients/` | source for the various client programs |
|     `xterm/` | the standard terminal emulator |
|     `xclock/` | a clock program |
|     `xev/` | show input events |
| `contrib/` | vast number of unsupported programs |
| `demos/` | demonstration programs |
| `fonts/` | the font manipulation programs |
| `lib/` | the libraries used by the client programs |
| `rgb/` | colour database support |
| `util/` | Generally useful programs |
|     `imake/` | the makefile generator |
|     `imake.includes/` | rules used by imake |
|     `patch/` | program to apply updates |
| `server/` | source for the server |
|     `ddx/` | Device Dependant X |
|         `cfb/` | generic Colour Frame Buffer code |
|         `mfb/` | generic Monochrome Frame Buffer code |
|         `mi/` | Machine Independent frame buffer code |
|         `snf/` | Server Natural Format (generic font code) |
|         `qvss/` | Drivers for a DEC workstation |
|     `dix/` | Device Independent X |
|     `os/` | Operating System dependant bits |
|     `include/` | Various include files required by the server |

**Table 1**: *X source tree*

Most workstations are based on a 32 bit architecture. If your target machine is different, say 64 bit, or if there are some restrictions on alignments of network structures you will need to change some of the definitions in `X11/Xmd.h`.

## 7. The Client Side

In addition to the above general changes only one other change will normally be required for the client side. One of the X libraries supplies some higher level functions. One of these functions draws a simple bar chart of the current load average on the machine. The source is in `lib/Xaw/Load.c`. Generally this file will compile without any changes but then not be able to work out the load so the bar chart will always be flat. On most systems with Sun Microsystems NFS you will need the same code that `sun` require.

You should now be able to type

```
make World
```

in the top of the X tree and it should all rebuild. Except that since the server hasn't been ported yet it will fall over when it reaches it.

If you do not require the server then you should have a:

```
#define BuildServer             NO
```

in the machine dependent macros file for `imake`. In which case `make` will not attempt to build the server or font files.

## 8. Porting the Server

If the target machine is close to the ideal then the port breaks down into two parts: input and output. For output you should be able to use either the generic monochrome or generic colour frame buffer code: `mfb` or `cfb`.

Loosly what goes on is that output is driven through some complex device descriptions which contain pointers to routines to perform various functions. For a given graphics screen a routine is called to fill in the description and initialise all the pointers to routines specific to that screen. To save you from writing your own routines two tiers of output routines are provided. The most generic, and so the slowest, is `mi`. These are the Machine Independent drawing routines and merely require some low level get pixel and put pixel operations. But because `mi` is so slow `mfb` and `cfb` were written. Both still use some `mi` routines for convenience. You should be able to use either `mfb` or `cfb` as a base for all your graphics output.

Input is generally more of a problem since memory mapped queues of input events are not common. Normally some form or `ioctl` or `read` system call is required to get the event. There is an additional requirement due to the way that the server waits for input from all the possible clients. It does this in the sample server by sitting in a `select()` waiting to be awoken when something interesting happens. You have to find some way of breaking out of the select when mouse and keyboard events become available. Some hooks are provided to do this.

## 8.1. The Steps in Porting

The first thing to do is to update the server specific configuration files. The first file is `server/include/servermd.h`. Here you will find definitions which give the bit and byte order of the screen, the alignment of bits in the screen and, the almost defunct, glyph padding value. You may also care to change the `VENDOR_STRING` definition to reflect your system.

Next change the `Imakefile` in the server directory to know about your target machine. Normally this involves adding the name for the target `ddx` directory to `ALLDDXDIRS`, a variable containing the name of the library containing all the screen specific code that you will write, and adding a conditional to ensure that a define of `X<machine>Server` is present. Add this symbol to the `ALL` definition. For the HLH Orion 1/05 these changes resulted in:

```
          ALLDDXDIRS = ddx/snf ddx/mi ddx/cfb ddx/mfb \
                       ddx/hlh ddx/cfb

                 HLH = ddx/hlh/libhlh.a

#ifndef XhlhServer
#define XhlhServer /* as nothing */
#endif

ALL =  XhlhServer
```

Because I had removed all the other machine specific drivers to save space I shortened the `ALLDDXDIRS` and `ALL` lists to reflect this.

You then need to add a block of rules for your target. Most of the existing sets of rules are similar, here is the version used for the Orion 1/05.

```
#
# hlh server
#
HLHOBJS = ddx/hlh/hlh_init.o $(FONTUTIL)
HLHDIRS = dix ddx/snf ddx/mi ddx/mfb ddx/cfb ddx/hlh os/4.2bsd
HLHLIBS = $(HLH) $(CFB) $(DIX) $(BSD) $(SNF) $(MFB) $(MI) $(EXTENSIONS)
#
# -lG is the StarPoint graphics library.
# Can also add -p or -pg if profiling all part of the server.
HLHSYSLIBS = $(SYSLIBS) -lG -pg
XhlhDIRS = $(HLHDIRS)

ServerTarget(Xhlh,
      $(EXTDIR) $(FONTUTILDIR) $(HLHDIRS),$(HLHOBJS),$(HLHLIBS),$(HLHSYSLIBS))
```

The `OBJS` rule lists the file containing the input and output initialising functions and the font handling routines. Since all the screen specific routines are usually held in a library (in the above example it is the library given by `$(HLH)`) none of it will be loaded unless required. So at least one part must be kept apart and reference a key routine in the library to cause it to be loaded.

## 8.2. Writing a Server

The best way to get a server up and running on your particular target is to take whichever existing server matches most closely your machine and use that as a base. It will be a subdirectory of `ddx` so copy it into its own directory and start changing all the names (both filenames and variable names) to match your machine.

Deciding on which existing driver to use is always going to be a problem. The bottom line is that you have to read through them all to see. Do remember that the input routines are normally a lot more complex that the output side. So choose a similar input system first.

### 8.2.1. Rebuilding the Server

After each stage in the porting process you will need to rebuild and test the server. The quickest way to do this is to initially rebuild all the libraries by going to the `server` directory and calling `make`. Since the server isn't written yet it will fall over, but hopefully not before rebuilding all the libraries. If it does not rebuild them all then do each one by hand.

After each change in the machine specific directory rebuild the library in that directory by typing `make`. Then go back to the main `server` directory and do

```
make loadX<machine>
```

for example:

```
make loadXhlh
```

## 8.3. Starting Input and Output

The `dix` part of the server will call two routines look like:

```
void
InitOutput(screenInfo, argc, argv)
        ScreenInfo *screenInfo;
        int argc;
        char **argv;

void
InitInput( argc, argv )
        int argc;
        char *argv[];
```

`InitOutput` has two main jobs, filling in `screenInfo` and calling `AddScreen` to add in each screen, which is presumably either given as an argument or found by a call to a hardware probe. The routine passed as the first argument to `AddScreen` will open the required screen and possibly perform some other initialisation then call either `mfbScreenInit` or `cfbScreenInit` to setup rest of the screen.

`InitInput` has to call `AddInputDevice` on to initialise the each input device, normally just mouse and keyboard. Once initialised they have to be registered by calling: `RegisterPointerDevice` and `RegisterKeyboardDevice`.

Example of this can be found in: `ddx/dec/qvss/init.c` and `ddx/sun/sunInit.c`.

## 8.4. Simple Output

Once you have picked a system, copied it and renamed appropriately, comment out the input system and concentrate on the output side. This should be the least time consuming part of the system to get working and will give you something you can demonstrate.

Do remember that the device independent parts of X will call your device specific functions via pointers held in various structures. So when commenting out the input side be sure to leave something for these pointers to access.

If your target has a colour screen you may find it easier, initially, to port `mfb` to it rather than use its colour capabilities. This will avoid having to write much colour map handling. Simply fill in the colour map such that everything is either black or white. The sample server in the `ddx/dec/qvss` directory would then serve as a good guide as it is one of the simplest of all the servers. Look in particular at the screen initialising routine `qvssScreenInit` and how it uses `mfbScreenInit`.

Without hardware support you will find that `mfb` is an order or magnitude faster than `cfb` so once it is working it is worthwhile keeping around as an option.

On a colour system once monochrome output is working you can then upgrade to colour output by using `cfb`. Most of the steps required by `cfb` are the same as those for `mfb` except that you must also add colour map handling for your hardware. A good example of this is in: `ddx/sun/sunCG3C.c`. In common with many of the sample colour servers this Sun driver can be compiled with `STATIC_COLOR` defined. `Cfb` must also be compiled with this defined and then it will not attempt to modify the colour map. This allows `cfb` to be tested without having to write the colourmap handling. Personally I found the colour map relatively straight forward to handle and did not use `STATIC_COLOR`.

Although this may appear to be a somewhat simplistic overview of output you should find in practice that the once the hardware has been initialised and its details passed down to the either `mfb` or `cfb` it is complete. There is no need to write line drawing, text output, area filling or any other graphics of those kind of graphics routines as they are all done for you.

### 8.4.1. Testing Output

There are some clients that do not require input, try `clients/xclock` and `demos/ico`. When starting them give a fixed geometry (including location) because without input working you will not be able to specify it.

## 8.5. Simple Input

Input, as has already been mentioned, is somewhat more complex. A technique I have found useful is to map the available event mechanism into the kind of circular queue that the server prefers. On most systems mouse and keyboard events can be treated separately. So to start with just attempt to handle mouse events as these are the simplier of the two to handle under X.

### 8.5.1. Mouse Input

There are four states an input device can be set to. For examples see `qvssMouseProc` in `ddx/dec/qvss/qvss_io.c`. First it can be `DEVICE_INIT` in which case the device has to be initialised. At this stage the mouse button map is also initialsed to show how many buttons are present and the functions to call to get movement events and to change mouse characteristics. This is all done by calling `InitPointerDeviceStruct`. If you are faking an input event queue then you should also call `SetInputCheck` on entering this state and pass it the addresses of the queue pointers that are changed when events are added to the queue. When the two values these pointers point to are different then the `dix` side of the server will know that there are input events to process.

A device can also be turn on by changing to `DEVICE_ON`. If you have a file descriptor that you can give to a `select` to show when input events are available then at this point you should call `AddEnabledDevice` with it. Similarly on getting `DEVICE_OFF` pass it to `RemoveEnabledDevice`.

Finally a device can be closed with `DEVICE_CLOSE`.

When it knows it has an input event to deal with `dix` will call `ProcessInputEvents`. This will have to convert each available event from the native machine format into what X expects and pass it to `dix` to deal with.

Converting the input event is normally is mapping the fields from the hardware or operating system into similar fields in an `xEvent`. For example see `ProcessInputEvents` in `ddx/dec/qvss/qvss_io.c`. The problems with mouse input are normally mapping coordinates, the hotspot and timestamping. Mapping coordinates will normally just entail converting the supplied coordinates so that `0,0` is the top left of the screen. The point that represents the mouse is its *hotspot*. On a cross shaped cursor the hotspot will be the center of the cross. So if your mouse coordinates are in terms of one of the corners of the box that the cursor is in you will have to map the coordinates to take this into account. Timestamping shouldn't really be a problem since you could simply call the dix routine `GetTimeInMillis()` and stamp each event that way. `GetTimeInMillis()` makes a system call to find the time. However this may prove to be expensive. You may have a timestamp on the event that could be used, think about switching to this once the server is basically working.

### 8.5.1.1. Testing Mouse Input.

To test mouse input there is a useful program: `clients/xev` It puts up a window containing a sub-window then will give a running commentary of any input events concerning that window.

## 8.5.2. Keyboard Input

Keyboard input is slightly more complex since it was designed to allow for all the various kinds of keyboards that are available. Basically the server will try to map the absolute scanline codes generated when keys go up and down into real keys. So it obviously has to handle the special keys itself, such as control, shift and shift-lock.

### 8.5.2.1. The Keyboard Map

If you cannot get at the keyboard scan codes but only at the mapped keys you will have to perform some kind of reverse mapping to fake up the original sequence of scan codes that the server expects. Consider a key like capital-`A` which is really something like: `shift` key down, `A` key down, `A` key up, `shift` key up.

The server needs to know about the map of the keyboard scan codes. This map is to convert the scan code into the key label actually printed on the key. This is broken down into two parts. Modifier keys, such as shift and control, and regular keys, such as the alphanumerics and keypad keys. A good example is `GetLK201Mapping()` in `ddx/dec/lk201/lk201.c`. A modifier map is filled in with the particular modifiers available and the main key area is then filled in. The main area is really a two dimensional array. One dimension is the number of keys (given by `minKeyCode` to `maxKeyCode`, the minimum and maximum scanline values) and there can be more than one symbol on each key. For example on most keyboards in the main key area the `5` key also has `%` on it, which you can get at by shifting. This key would have an array entry like (this is borrowed from the HLH specific keyboard codes):

```
        XK_5,           XK_percent,      /* 35 */
```

So the keyboard map is `mapWidth` deep, in the above example mapWidth would be two.

Note that the alpha keys would have entries like:

```
        XK_A,           NoSymbol,        /* RIGHT */
```

*NOT*

```
        XK_a,           XK_A,            /* WRONG!! */
```

The reasoning is a little obscure but if you look at most keyboards you'll find that the alpha keys only list the capitalised version of the key on the keycaps and it is that that the key map is specifying.

### 8.5.2.2. Getting Key Events

A keyboard device can be set into the same four states as the mouse, so most of the discussion on mouse input is also applicable here. If you look at `ProcessLK201Input` in `server/ddx/lk201/lk201.c` then you will see that apart from converting the event details (up or down) into a what the server requires it also detects special keys going up and down and turns on or off the appropriate LED's.

### 8.5.2.3. Testing Keyboard Input

The same program used to test mouse input `(clients/xev)` can also be used to test keyboard input. Do remember to check out each key, since the map entries for keys look all alike it would be easy for a simple typo to have crept in.

## 8.6. You are Finished!

Now input and output are both working so you have now successfully ported X!! Before letting users loose on it is a wise idea to test out all the various X clients to ensure that they all work.

## 8.7. Optimisation

Once the base release is working you may wish to optimise it. Normally there are two reasons for doing this:

1. You have special graphics hardware which you would like to make use of
2. The performance is slow

Before you start optimising you must find some objective way of measuring performance. The two most common ways of doing this are by running benchmark programs and by profiling the server. Personally I use a profiler and run X gathering statistics under a normal user environment. This should then show what are the most intensely used routines. Normally these will be text output, in particular the terminal emulator font: `fixed`, area filling, zero width lines and tiling.

What steps you take to optimise will largely depend on what your architecture is like and whether you have special graphics hardware. There are some machine independent optimisations available, the most widespread is the *Purdue Speedups* mentioned earlier. These enhance the performance of `mfb` and by following the same ideas you should be able to improve `cfb`.

**European UNIX® systems User Group**

Owles Hall, Buntingford, Herts SG9 9PL, England
Tel: + 44 763 73039
Facs: + 44 763 73255
email: euug@inset.uucp

ISBN 0 9513181 2 8