EUUG

European UNIX® systems User Group

Spring '90 CONFERENCE Proceedings

23-27 April
at
Sheraton Hotel
Munich
West Germany

EUUG

European UNIX® systems User Group

Proceedings of the Spring 1990 EUUG Conference

April 23–27, 1990

Sheraton Hotel,

Münich, West Germany

ISBN 0 9513181 6 0

Further copies of the proceedings may be obtained from:

EUUG Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

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

ACKNOWLEDGEMENTS

Hosted by:	German UNIX systems Users Group	
Programme Chair:	Hans W Strack-Simmerman	iXOS Software GmbH
Programme Committee:	Daniel Karrenberg	CWI, CST
	Marc Nyssen	Dept of Medical Informatics, VUB
Conference Executive:	Ernst Janich	Nixdorf Computer AG
Tutorial Executive:	Neil Todd	GID Ltd, London, UK
Proceedings Production:	Dave Edmondson	Imperial College, London, UK
	Stuart McRobert	Imperial College, London, UK
	Nick Williams	Imperial College, London, UK
	Frank Taylor	Imperial College, London, UK

We would like to thank the German UNIX User Group for hosting this event, its Chairman Anton Gerold and Snoopy of iXOS Software.

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

— dme, sm, njw, fst.

UNIX Conferences in Europe 1977–1990

UKUUG/NLUUG meetings

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

EUUG Meetings

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

Technical Programme

Author Index

Multi-threaded Processes in CHORUS/MIX

François Armand

Fréderic Herrmann

Jim Lipkis

Marc Rozier

Chorus Systèmes
6, avenue Gustave Eiffel, F-78182,
Saint-Quentin-En-Yvelines, France
Tel: +33 1 30 57 00 22,
Fax: +33 1 30 57 00 66,

mg@chorus.fr

ABSTRACT

Interest in concurrent programming in recent years has spurred development of "threads", or "lightweight processes", as an operating system paradigm. UNIX-based systems have been especially affected by this trend because the smallest unit of CPU scheduling in UNIX, the process, is a rich and expensive software entity with a private memory address space. In this article we examine performance constraints affecting concurrent programs, including real-time applications, in order to understand and evaluate the demand for a new scheduling model. Although performance criteria differ sharply among various application domains, we conclude that a single thread model can provide efficient concurrent execution in a general-purpose operating system.

We describe the design considerations behind the thread-management facilities of CHORUS/MIX, a UNIX-compatible operating system built for distributed, real-time, and parallel computing. Mechanisms for processor scheduling and inter-thread synchronization must satisfy the needs of each of these three categories of concurrency. Extension of the traditional UNIX interface to the multi-threaded environment is an area of particular delicacy. CHORUS/MIX adopts novel approaches for signal handling and other UNIX facilities so as to ensure a smooth transition from sequential to concurrent semantics in applications.

1. Introduction

Concurrency has become increasingly important as a programming paradigm in recent years. It is a traditional tool in application areas where concurrent activities arise naturally, such as simulations, servers in distributed systems, and programs which interact with humans or hardware devices. With the recent availability of multiprocessors, parallel programming has become an important technique for increasing the speed of computation. Concurrency is most often invoked through programming of multiple threads of control that access a shared memory context. A variety of general- and special-purpose programming languages provide concurrency in their semantics.

Operating systems become involved in the support of concurrent programming for two reasons. First, applications are often written in low-level programming languages like C or assembler which do not support concurrency. Creation and management of execution threads must be arranged through run-time services of the operating system or system-level libraries. Second, and more important, multiprocessor hardware and real-time applications each impose requirements on the scheduling and resource management facilities provided by the operating system. The nature of the interaction between supervisory software and user programs, in light of these new requirements, has become a major topic of both academic and commercial operating system research in recent years. Potential solutions are constrained by considerations of compatibility, portability, and standardization. New features must be introduced into existing operating systems through graceful and non-disruptive extensions. It is important to distinguish the *interface* provided by a system from the *implementation* of the corresponding features. Ideally, a single interface — that is, a coherent set of operating system facilities — should support both conventional and concurrent programming, with or without real-time constraints, on uniprocessors and multiprocessors. The *implementation* of these facilities will vary according to the underlying architecture and specific system goals. Nonetheless, a uniform interface requires negotiation of conflicting performance requirements in several areas.

In this article we describe design issues and alternatives for thread management, including inter-thread communication, synchronization, scheduling, and exception handling. Many of these functions play prominent roles in concurrent language semantics, as well as in operating system design. Ada, for example, includes semantics for task creation, synchronization and exception handling. We will ignore language and programming issues, however, so as to concentrate on the operating system services needed to *support* concurrent programming environments. Focusing primarily on UNIX derivatives, we consider some of the approaches taken in existing multi-threading systems, including CMU's Mach [Coo87a, Tev87a], DEC's Topaz [McJ88a], and the SunOs Lightweight Process Library [Sun87a]. We present the thread interface designed for CHORUS/MIX, a UNIX-compatible real-time operating system which builds on the experience of earlier systems. The goal in CHORUS/MIX was to design a coherent set of thread management features which address the performance requirements of both multiprocessor and real-time programs in a unified manner.

2. Performance requirements and alternatives

Scheduling and resource management in the UNIX environment are defined in terms of the *process*, a rich software entity which incorporates a stream of instruction execution, a memory address space, an exception-handling environment, a set of access capabilities, and other information about the executing user program. Process creation and deletion are expensive functions, as is CPU context switching among running processes. Good performance in concurrent applications will be impossible to the extent that these functions are frequently invoked. An often-proposed solution is the addition of a new scheduling entity called a "lightweight process" or a "thread", which represents an executing stream (and perhaps a stack) but does not contain an address space or other resources. (This is the approach taken in Mach, Topaz, Amoeba and other systems.) Programs arrange concurrency by invoking multiple threads, all sharing a process' address space. This seems attractive in the UNIX world because existing program development tools, designed to compile and link sequential programs for a monolithic address space, can be extended to multi-threading without change, albeit with some awkward limitations (which we discuss in later sections).

However, adding a new unit of scheduling constitutes a fundamental change to the heart of the system interface, with repercussions in many areas. Extending signal handling, memory and resource management, and process control features into this new framework requires resolution of a number of difficult issues. Before taking this path, then, it is worthwhile to examine the performance issues that lead to the demand for lightweightness, and to consider alternative approaches. We will look closely at two application domains which are highly sensitive to scheduler performance — computations with fine-grain parallelism, and real-time systems.

Requirements for Multiprocessing

Performance of computation-intensive programs on multiprocessors is affected by two related factors.

- *Fine granularity.* As multiprocessors become available with increasing numbers of processing elements, the challenge of partitioning the computational work of an application to fully exploit the hardware resources becomes greater. Amdahl's Law states that the number of processors that can be effectively used to reduce the elapsed execution time of a computation is limited by the percentage of the program's execution that is spent in serial (or less parallel) sections. Serial execution is often a consequence of the high overhead of parallelization. If the time required for initiating and terminating parallel execution is much greater than that of the computations to be performed, then the extra available processors are of no use. The desire to parallelize fine-grain operations leads to a demand for very inexpensive fork/join functions. Fine granularity occurs in a wide variety of situations, including object-oriented programming and concurrent functional languages as well as numerical computation.

- *Flexible semantics.* In practice, the fineness of granularity attainable is a consequence of the semantics of the concurrent threads. In Ada, tasks are subject to priority-based preemption and elaborate rules for exception handling and termination. The run-time system may incur considerable bookkeeping expense when tasks are created or destroyed. By contrast, most Fortran systems with MIMD multiprocessing extensions create simple threads for parallel DO loops which must run to completion (cannot block or switch processors) and have little semantic baggage. Initiation of parallel execution requires a very small number of instructions in common cases. Other concurrent languages and tasking libraries used from C or assembler occupy the area between these two extremes.

In order to support a wide range of task granularities and adapt to various semantic requirements, thread management for multiprocessing is generally implemented in user mode — usually in general or language-specific libraries — rather than in the operating system. An apparent drawback is that two separate schedulers must be implemented, one at the system level and one in user level code. However, no centralized thread scheduler can alone handle the granularity needs that arise in parallel programming systems. Creation and deletion of threads in Mach and Topaz require several hundred instructions on a DEC Vax computer, according to measurements made by their respective designers [Tev87a, Swa89a]. The semantics of most concurrent languages allow threads to be created directly, without system intervention, far more cheaply. This is particularly true in the case of Fortran parallel DO loops, where fine granularity is often crucial. Thus reliance on supervisor-level threads limits the utility of a system for other than coarse-grain parallelism.

Requirements for Real-Time

Robotics, process control, and other real-time domains pose the following requirements.

- *Fast response.* A key metric of a real-time system is the maximum time required between the occurrence of a hardware or software event and the execution of the first instruction of application code that handles the event.
- *Scheduling control.* Real-time systems often rely on thread priorities to control CPU scheduling. For example, a thread that interacts with a physical process will be subject to tighter timing requirements, and thus assigned a higher priority, than one that is spooling or analyzing data. The thread semantics should include a system-wide priority level which may be changed by the user.
  An alternative, *deadline scheduling*, has been proposed in the research literature and used in specialized environments. Here, information concerning the real-time tasks' timing characteristics and deadline requirements is communicated to the scheduler instead of task priorities.
- *Determinism.* Response should be predictable as well as fast, since designers must accommodate the upper bounds on software operation timing. Thread priorities must be strictly enforced, by preemption when necessary. Thus at each moment in time, the highest-priority ready thread is active in execution. Extraneous context switches, interrupts, and other asynchronous activities can prevent tight upper bounds on time requirements.

Less demanding applications (i.e., without real-time constraints) may nonetheless make use of priorities or related features. In the next section we will consider ramifications on system design posed by these requirements.

Alternatives for Thread Management

There are three common approaches for thread management.

1. *User mode threads within a process.*

   A library package which multiplexes a single UNIX-style process to implement multiple threads can provide a useful concurrent programming environment. Threads execute as coroutines, with optional timer-based preemption. While the operating system kernel need know nothing about the threads' existence, concurrency can be increased if asynchronous kernel operations are provided. While an I/O system call is being processed or a page fault is being resolved on behalf of one thread, this enhancement would allow other threads to execute within the same process. UNIX signals can be used to inform the thread scheduler of the completion of the asynchronous operation. The SunOS Lightweight Process (LWP) Library is an example of this approach, though without general asynchronous system calls or asynchronous page fault resolution. Other implementations exist within specialized or language-specific contexts.

Threads within a single process cannot make use of multiple processors, nor can they satisfy the requirements of real-time applications. Within these limits, however, this is an attractive approach because it minimizes disruption to the operating system interface.

2. *User mode threads scheduled across multiple processes.*

   We concluded earlier in this section that parallel computation on multiprocessors is most effectively implemented with threads managed and scheduled in user mode. A mechanism quite similar to that of the previous paragraph suffices. Multiple processes — at least as many as the number of hardware processors to be used — are needed, instead of only one. These processes must provide access to some amount of common memory. In all other respects, use of "heavyweight" processes is acceptable because they will be created and destroyed very rarely, usually only at the beginning and end of program execution. Various UNIX versions have supported shared memory regions among processes for years — though sometimes in a manner which limits usability — and this is still a relatively minor extension to underlying process model. Although processes retain their individual address spaces, large amounts (in theory, all) of the address space may contain shared regions. Controlled sharing of other process attributes, such as file access capabilities, can be provided as well but is not strictly necessary [Edl88a].

   The processes are used as thread executors. Let us first imagine that each of the processes created by a parallel program is locked onto one processor and will never be preempted, by virtue of a special provision in the system scheduler. The various threads of the user program are then scheduled onto these process/processor pairs, just as in a standard operating system processes are scheduled onto a processor. A queue of executable threads is maintained in shared memory and accessed by each process/processor, executing the code of the user-level thread scheduler between executions of user program threads. The scheduler code might be loaded from a thread-management library or emitted by a parallel-language compiler. Scheduling overhead is minimized, and task granularity is limited only by the parallel language semantics and the machine architecture. Use of shared regions for thread-specific storage allows threads to context switch among processes with low overhead, if necessary.

   This is an attractive approach for general computation on multiprocessors, and it has been used in both commercial and research systems (for example, the Sequent Balance/Symmetry [Bec87a] and NYU Ultracomputer [Edl88a]. Because the lightweightness of the scheduling entity provided by the operating system is unimportant, fundamental revisions to the operating system paradigms are unnecessary. It may be beneficial, however, to add some provision for grouping the processes that act as thread executors for one program. Processes are not in general locked onto processors, and so process scheduling becomes an issue. A process aggregate or "container" would be useful so that processes within one program might be scheduled onto processors together. Another motivation arises in distributed systems that support process migration. For high-bandwidth communication over shared memory, the collection of cooperating processes should remain on the same (multiprocessor) node.

   Real-time requirements, however, are not as easily satisfied by user-managed threads.

   - Because the central scheduler is not aware of individual threads, it cannot recognize or enforce global thread priorities. A user mode thread scheduler might implement priorities, but only at the level of an individual job. Deadline schemes also require global thread scheduling.
   - Events recognized within the operating system kernel affect thread state. A hardware interrupt could cause a blocked high-priority thread to become runnable, but the system process scheduler cannot directly place that thread into execution. Instead it must pass the event to the user level scheduler, which adds an extra layer of overhead.
   - Both of these problems might be addressed by binding selected high-priority threads to specific processes and thus using process priorities to simulate system-level thread priorities. But we would begin to lose the advantage of lightweight threads. Context switching among processes is expensive on many architectures because address mapping hardware must be updated and often caches must be flushed.

     While user mode threads provide a simple and useful paradigm for multiprocessing, they do not appear to be able to support the combination of prioritized threads and tight response time requirements that arise in real-time applications.

3. *Threads scheduled by the operating system.*

   Thus real-time considerations mandate the use of threads implemented and managed through the central system scheduler. In CHORUS/MIX, the memory address space and resource ownership functions remain with the process, but a process may contain an arbitrary number of threads, and threads rather than processes are scheduled onto hardware processors. With no intermediate thread-management layer in user mode, the system scheduler can enforce thread priorities directly and provide the deterministic scheduling required for real-time.

   On multiprocessors, in the absence of real-time constraints, system-level threads may be used as executors of the parallel program activities which are created and scheduled in user mode. A multi-threaded process acts as a "container" of the system threads which participate in execution of a concurrent program. We conclude that this system interface model can effectively address the needs of several different categories of concurrent applications.

3. Threads in CHORUS and CHORUS/MIX

CHORUS is a family of operating systems based on a minimal real-time nucleus which provides low-level services for distributed processing and communications [Roza]. The nucleus can be scaled to run on a variety of hardware configurations, including embedded boards, multicomputer and multiprocessor configurations, networked workstations, and dedicated servers. CHORUS operating systems are built as sets of independent, dynamically-loadable servers that rely on the generic services provided by the nucleus, i.e., thread scheduling, network transparent inter-process communication (IPC), optional virtual memory management, and real-time event handling. CHORUS-V3, the current version, was developed by Chorus Systèmes and has been commercially available since early 1989. Earlier versions were designed and implemented by the Chorus research project at INRIA between 1979 and 1986. Work on UNIX integration and compatibility in CHORUS began in 1984.

The physical support for a CHORUS system consists of a set of *sites*, interconnected by a communication network. A site is a tightly coupled grouping of physical resources: one or more processors, memory, and attached I/O devices. There is one CHORUS nucleus per site.

The *actor* is the logical unit of distribution of processing and of collection of resources in a CHORUS system. Actors in CHORUS are similar to *tasks* in Mach. An actor constitutes an execution environment, including a protected address space, for one or more *threads*. Each actor is tied to a single site. Within a site, threads of multiple running actors are scheduled by the nucleus as independent activities. Thus multiple threads of an actor may run in parallel on multiprocessor sites. Threads of the same actor may communicate and synchronize through shared memory. In addition, CHORUS offers message-based facilities which allow any thread to communicate and synchronize with any other, whether within the same actor, across actors, or across sites. Message exchange under CHORUS IPC may be either asynchronous or by demand-response, also called *remote procedure call* (RPC).

The CHORUS/MIX operating system is composed of a CHORUS-V3 nucleus in combination with a set of subsystem servers that implement a System V-compatible UNIX. Each UNIX process is implemented by one CHORUS actor; hence the multi-thread nucleus model extends naturally into the UNIX layer. The traditional UNIX services are augmented in CHORUS/MIX with facilities for distributed, parallel, and real-time computing. In order to distinguish the thread interface provided by the CHORUS nucleus (used primarily by subsystem programmers) from the thread interface provided by CHORUS/MIX (used by UNIX applications programmers), CHORUS/MIX threads will be called *u_threads* (short for "UNIX threads") in the remainder of this paper.

The u_thread management interface has been defined to satisfy two major objectives:

- to provide a low-level, generic interface which can satisfy a variety of needs, including support of existing thread interfaces and language-dependent packages. This was also one of the CHORUS nucleus objectives, and as a result the thread management interfaces at the nucleus level and UNIX level in CHORUS are quite similar.
- to have minimal impact on the syntax and semantics of UNIX system calls so that existing mono-threaded programs can easily become multi-threaded programs.

The basic u_thread management interface includes primitives for creating and deleting u_threads, suspending and resuming their execution, modifying their priorities, obtaining the identification of the current thread, and obtaining and altering the CPU context (register values, etc.) of a blocked thread. All of these services are low-level and incorporate a minimum of semantic assumptions. Policies regarding stack management and ancestor/descendant relationships among u_threads, for example, are left to higher-level library routines which are provided to make u_thread usage easier for programmers.

4. Synchronization

Threads running on a multiprocessor can coordinate directly through shared memory, using hardware synchronization primitives like *test-and-set* or *fetch-and-add* when necessary. Nonetheless, operating system services play an important role in synchronization. In this section we consider two basic forms of synchronization, mutual exclusion and event notification. More advanced coordination functions can be built on top of these, or through use of other system facilities. Ada-style synchronous rendezvous, for example, can be implemented within an actor (or process) using shared memory, or in a more general manner using the CHORUS RPC facility. Functions described in this section are identical in the CHORUS nucleus and in CHORUS/MIX.

Performance goals for synchronization functions again differ sharply between the two application domains of parallel computation and real-time.

- *Multiprocessing.* Most important is to minimize the overhead of synchronization. In particular, "easy" operations like obtaining a lock which is already free should require only a few machine instructions.
- *Real-time.* Overhead is still a consideration, but determinism is again crucial for real-time. When one thread releases a lock, it is assumed that the highest-priority waiting thread is scheduled. Some systems go further and allow a releasing thread to designate a specific target thread (of the same or higher priority) that is guaranteed to execute immediately.

Overhead on multiprocessors can be minimized through *busy-waiting* synchronization, in which a thread tests a condition or a lock in shared memory, and, if unavailable, continues testing as long as necessary. This is especially effective if the synchronization is satisfied immediately or very soon, because no context switching or other software overhead is introduced. However, when resources are not available busy-waiting can waste processor time, tie up the memory subsystem, and even lead to deadlock in some cases. Therefore we may want to switch the waiting thread off of its processor and queue it until the condition is satisfied. This is known as *blocking* synchronization. Various hybrids of busy-waiting and blocking have been proposed in order to combine the performance advantages of each [Ous82a].

In CHORUS, synchronization functions and implementation strategies for multiprocessing are left to user-level libraries and execution environments. The operating system kernel merely maintains the thread state (ready vs. blocked) and queues blocked threads, thus maintaining flexibility for the user layer to optimize for specific requirements or hardware features. As we shall see, the kernel interface is designed to ease the job of writing synchronization library code that is free of *race conditions*, timing-sensitive errors that can unpredictably cause the synchronization to go awry.

For real-time purposes, however, synchronization must interface directly with the thread scheduler. Hence we also provide a set of synchronization functions directly in the operating system nucleus. Only blocking functions are provided, since busy-waiting is difficult to reconcile with deterministic scheduling.

4.1. Semaphores

The *binary semaphore* or *lock* provides mutual exclusion for protection of shared data. A generalization, the *counting semaphore*, allows a fixed number (not necessarily one) of concurrent accesses to a resource. Support for these functions presents two demands to the CHORUS nucleus. First, we must support semaphores directly in the nucleus, so that real-time constraints may be met. Second, we need a race-free mechanism by which user-level semaphores or other synchronization routines may cause threads to be blocked and restarted by the nucleus. However, we can address both with a single nucleus facility.

The *P* and *V* (counting semaphore *obtain* and *release*) functions were defined by Dijkstra as follows [Dij68a]. The semaphore variable *sem* contains an integer counter.

```
P(sem)    – atomically: decrement the counter, check the result
          – if the counter value has become negative,
                    block the current thread
          – return

V(sem)    – atomically: increment the counter, check the result
          – if the counter value remains less than one,
                    awaken the waiting thread with the highest priority
          – return
```

The function of the P and V operations depends on how the counter is initialized. If it is set to one, then P and V implement mutual exclusion. P is executed on entry to a critical section, V on exit, and the result is that no more than one thread can execute in the critical section at a time.

However, if we associate a private semaphore with each thread, and initialize the counters to zero, then P and V can be used to perform the functions "block the current thread" and "release the previously-blocked thread", respectively. When any user-level synchronization routine checks a condition, perhaps busy-waits, and then determines that it is time to free the processor to execute a different thread, it invokes P on its private semaphore. Later, when the condition has been satisfied, the routine can resume the suspended thread with V on that thread's private semaphore. Use of the counting semaphore here lends robustness in the face of unforeseen timing circumstances on multiprocessors. Suppose that a decision has been made that the current thread must block but it has not yet performed its P operation, and meanwhile a second thread satisfies the relevant condition and calls V. We must ensure that when the P is finally executed, the first thread will not be made to wait forever for a corresponding V that will never occur. In fact, the V is effective regardless of the order of P and V. In this example it will increment the counter to one, and the subsequent P will immediately return without waiting. Thus the counting semaphore included in the nucleus for direct use in real-time situations also serves to provide reliable thread control for general user synchronization.

4.2. Event notification

Often threads must synchronize because one activity cannot logically proceed until it obtains data, or notification of an internal or external event, or other status information from a second thread. Mutual exclusion alone cannot provide event notification. For synchronizing access to discrete resources that occur in finite allotments, e.g. space in bounded buffers, counting semaphores can be used for "notify when not full" and "notify when not empty". But more general forms of event synchronization are useful in many applications. In the following subsections we describe two of the most popular mechanisms for event notification, and consider some of their strengths and weaknesses.

4.2.1. Condition variables

Topaz, the SunOS LWP Library, and the Mach "C Threads" library all provide some variation of Hoare's *monitors* with *condition variables* for event notification. A monitor is a package of functions protected with a binary semaphore. Under Hoare's discipline, each thread performing an operation on a shared object must first enter the corresponding monitor (i.e., lock the semaphore). Each condition variable declared within a monitor is associated with some logical condition which may affect threads' ability to continue processing. When a thread in the monitor needs to wait for a condition to be satisfied, it issues a *condition_wait*, thus suspending the current thread and also releasing the semaphore. Later, when another thread in the monitor satisfies the condition, it may issue either *condition_signal* to wake one waiting thread or *condition_broadcast* to awaken all threads currently blocked on the specified condition variable. Awakened threads automatically re-enter the monitor before proceeding. Thus the shared program objects which make up the condition being awaited are protected at all times with mutual exclusion.

The power of this paradigm — as well as its drawbacks — stem from the coupling of event notification with mutual exclusion. By defining composite functions as *atomic*, we can solve synchronization problems that would arise in use of more primitive functions. This arises in two areas.

- *Avoiding deadlock on condition_wait.*
  By specifying that the *condition_wait* operation release the monitor (semaphore) *atomically*, we can avoid an area of potential deadlock. If the *condition_wait* and semaphore release were independent nonatomic operations, then they would have to be done in some sequence. If the *condition_wait* comes first, then the semaphore would remain locked, and no other thread would ever be able to enter the monitor to satisfy the awaited condition. If the semaphore release comes first, then there would be a danger of a race condition similar to that discussed in Section 4.1: an intervening thread could issue the *condition_signal* before our original thread issues its *condition_wait*. Condition variables contain no counters, and no other form of memory, so the eventual wait might indeed wait forever because it has missed the corresponding signal. In lieu of keeping information in the condition variable, atomicity is used to solve the problem. The Topaz and Mach implementations of condition variables on multiprocessors guarantee atomicity in this case.

- *Deterministic notification on condition_signal*
  A second atomicity issue can effect real-time performance. Experience suggests that when a thread waiting on a condition is signalled, that thread is ready to do useful or critical work and should be scheduled before the signalling thread resumes and before other threads (of the same priority) are allowed to enter the monitor. Assuming that a thread issuing *condition_signal* or *condition_broadcast* holds the monitor semaphore, then it is important that the semaphore be released *atomically* with the signal or broadcast operation, then relocked before the thread proceeds. Otherwise, we find that neither of the two possible orderings allow optimal scheduling. If the semaphore is released first, then other threads waiting for the monitor may enter before the target thread is made executable, thus delaying its progress. If the condition signal takes effect first, then the signalled thread will awaken only to block immediately in attempting to re-enter the monitor. The penalty is several extra context switches.
  Some specialized real-time monitors enforce a guarantee that a signalled thread will execute immediately on invocation of the signal operation. Thus no intervening threads can enter the monitor to tamper with the logical condition that led to the issuance of the signal. The condition variable interface is sufficiently powerful to implement this stronger functionality if desired.
  However, no general-purpose thread system known to the authors includes an implementation of condition variables that enforces *any* form of atomicity on condition signal or broadcast.

Monitors and condition variables have serious limitations in other areas. One problem is immediately apparent: the monitor imposes serialization which may be logically unnecessary and may be detrimental to good performance on a multiprocessor. Designers of large parallel computers strive to avoid the need for mutual exclusion, using specialized hardware features like *fetch-and-add* in combination with software mechanisms. On such systems it is possible that a number of threads test a condition fully in parallel without threatening the integrity of shared objects. If the condition is satisfied, the threads proceed in parallel, otherwise they all block until a condition-satisfied broadcast allows them to resume — again, all in parallel. Using the condition variable interface described, there is no way to program the synchronization without spurious serialization.

Even where mutual exclusion *is* required, it may not always fit the model underlying the monitor/condition variable paradigm. Device drivers in CHORUS reside in subsystems, not in the nucleus, and event notification between interrupt routines and user-level threads is sometimes required. But mutual exclusion in this case is based on hardware interrupt masking, rather than software semaphores, and so condition variables are not usable without loss of performance. Further, there may be situations in which a condition is logically coupled to multiple semaphores rather than just one. Finally, condition variables may be difficult to use and understand in complex programs. There are subtle interactions between the condition and semaphore that can give rise to unexpected bugs or performance problems. (Birrell [Bir89a] discusses experiences in this area.)

Thus monitors and condition variables do not appear adequate for general-purpose thread synchronization.

4.2.2. Latching Events

Another popular event notification scheme involves synchronization objects which we shall call *latching events*. These differ from condition variables in that each event contains a persistent state which has two possible values, *OCCURRED* and *NOTOCCURRED*. Three operations are defined on latching events. Here *ev* is an event variable.

```
event_wait(ev)
    – if ev is in state NOTOCCURRED, enqueue the thread and wait
    – return
event_signal(ev)
    – if ev is in state NOTOCCURRED,
            change the state to OCCURRED and awaken all threads enqueued on ev.
    – return
event_reset(ev)
    – if ev is in state OCCURRED, change the state to NOTOCCURRED
    – return
```

When an event is signaled, it stays signaled until reset. The deadlock problem in the *condition_wait* operation above has no analog here, since it makes no difference whether an event_wait is followed or preceded by its corresponding event_signal. Otherwise latching events are similar in operation and usage to the broadcast mode of condition variables. However, this is an independent, primitive mechanism. No linkage to any other synchronization mechanism is involved, though events may be combined with semaphores or other synchronization devices by the programmer as required. Latching events are more flexible than condition variables, and avoid problems of spurious serialization. However they have no analog to the signal mode of condition variables, and thus cannot entirely supplant the latter as a general notification device.

CHORUS includes sufficient tools to allow implementation of condition variables, latching events, or other synchronization devices at the subsystem or user level. The scheduler does not currently provide such a feature directly. Unfortunately, little experience exists to guide the selection of synchronization primitives in a general-purpose operating system interface. The situation is particularly chaotic in real-time systems, because there is no consensus either on synchronization functions, scheduling paradigms, or the relationship between the two. Current real-time monitors tend to provide a multitude of *ad hoc*, redundant facilities for thread coordination. Investigation and experimentation are continuing in the search for software devices that can address this problem.

5. Thread Scheduling and Responsiveness

Real-time applications require deterministic thread scheduling which can be controlled by the user; time-sharing environments, on the other hand, require time-slicing and dynamic adjustment of priorities for good response times. The CHORUS nucleus provides both modes.

Scheduling is preemptive: at any given time, the running thread is always the ready thread with the highest priority. Scheduling decisions are based on *absolute* thread priorities within a system-wide range of priority values. The absolute priority is calculated as the sum of the priority of the owning process and the *relative* priority of the u_thread within the process. Relative priorities are useful for tuning of priorities among various applications and system components without disturbing the scheduling of the u_threads within a component.

Above a threshold priority known as SLICE_PRIO, scheduling within a priority level is first-in first-out (FIFO); a thread that yields control or is preempted is placed at the end of the queue for its priority. FIFO scheduling provides determinism. Combined with the capability to alter thread priorities at any time, this gives real-time users full control over scheduling both within a program and across the system.

Below the SLICE_PRIO threshold, threads are subject to time-slicing. CHORUS/MIX arranges that all u_threads are within this range by default. Thus real-time application programmers must raise the priorities of their u_threads explicitly to avoid time-slicing. Threads of the UNIX subsystem servers in CHORUS/MIX execute above SLICE_PRIO and are not subject to time-slicing. However, a range of priority values is available above that of the UNIX servers. This allows real-time application u_threads to execute at higher priorities than the server threads and thus to preempt them.

In traditional UNIX systems real-time responsiveness is limited by the fact that system calls are noninterruptable. A high-priority activity might be delayed for a period equal to the longest-running system call before it can respond to an event, even though the issuer of the system call executes at a low priority. This problem is addressed in CHORUS in two ways. First, the bulk of UNIX system call processing is performed in subsystem servers, under the calling user thread or a subsystem thread. These are subject to normal priority scheduling, as suggested in the previous paragraph. Second, both the UNIX subsystem servers and the CHORUS nucleus are themselves multi-threaded. This design allows both UNIX and CHORUS system calls to be interruptable at any point. Currently, inter-thread synchronization within these components is relatively coarse, with the result that concurrently-invoked system calls are occasionally delayed. Finer-grain synchronization is being introduced as required for good performance.

6. Adapting UNIX System Functions

In several respects, UNIX system services exploit the fact that memory, resource ownership, and signal delivery are tied to the unit of CPU scheduling. In adapting these services for multi-thread processes, we would like to preserve the existing interface and function to as great a degree as possible. Maximizing compatibility will minimize the difficulties faced both in converting old programs and writing new programs to use concurrency.

In some areas compatibility is problematical, but the basic UNIX process management functions extend readily. The *fork* primitive creates a child process, but with only one u_thread, corresponding to the u_thread in the parent process that executed the *fork*. *exec* overlays the current process memory with a new program, which begins execution as a mono-threaded process. Other related functions retain their traditional semantics or are adapted in minor ways.

Concurrently-executing system calls within a process could interfere with each other in such a way as to violate the semantics of some or all. However, serializing system calls in each process would limit concurrency to an unacceptable degree. CHORUS/MIX multiplexes system calls within a process according to the following policies.

- A u_thread executing a system call never prevents other u_threads from running in user mode.
- Execution of extended (non-UNIX) CHORUS/MIX system calls relating to CHORUS IPC or u_thread management can be completely multiplexed
- UNIX system calls that wait for I/O or other external events, including *read, write, open, pause, wait,* etc.) may be multiplexed with any other system call after the calling u_thread has blocked. (These are precisely the system calls which may be interrupted by signals in UNIX.) Thus multiple u_threads may invoke concurrent I/O operations.
- Only one u_thread in a process may be actively in execution of a UNIX system call at any moment. This restriction is imposed to protect the semantics of the UNIX emulation. The result is that system calls which affect the entire process, like *fork* and *exec*, and some others which execute very quickly, like *getpid* (obtain process id), are serialized.

6.1. Signals – Exceptions and Asynchronous Events

UNIX uses *signals* to manage both synchronous exceptions caused by errors in user code (i.e., divide by zero), and for asynchronous events such as expiration of a time interval or completion of an asynchronous I/O operation. Most signals are initiated by the operating system kernel, but user processes may also send signals to each other using a standard system call. Delivery of a signal acts like a software analog of a processor interrupt. The normal flow of code is suspended and a user-specified *signal handler* routine begins execution in the same process context (address space). If the handler returns to the system, execution of the original program resumes where it left off. Some signal handlers instead branch elsewhere in the program, usually to a caller of the interrupted routine, and resume program execution directly. (The standard C library includes a package of routines, called *setjmp* and *longjmp*, for non-local branching that can "unwind" the call stack.) The user program may choose to ignore a certain signal instead of providing a handler. If it does neither, then receipt of a signal terminates the process in most cases.

The introduction of multi-threaded processes mandates a reconsideration of signal handling. This has proved to be the thorniest area in extending the UNIX semantics, especially as pertains to asynchronous signals (as opposed to program exceptions). Traditionally, signals are sent to processes, and signal handlers are declared on a per-process basis. Which thread should receive delivery of a signal directed to a multi-thread process? More fundamentally, one might question whether signals are an appropriate mechanism at all in a multi-threaded environment. The argument is that asynchronous interrupts are confusing, difficult, and bug-prone in user programs, especially when locks or other synchronization facilities are being used by the interrupted code. In many cases, signals can be replaced through use of multiple concurrent threads. Instead of initiating an asynchronous operation, performing other tasks concurrent with the operation, and awaiting the signal at completion, one can create a concurrent thread to perform the operation using simpler synchronous system functions which return control only after the operation is complete. Further, thread-to-thread notification and abort can be provided with a mechanism that is less intrusive than an asynchronous interrupt.

Signals play an important role in UNIX and must be extended for multi-thread programs in a compatible and graceful way. Nonetheless some designers have attempted to minimize reliance on asynchronous interrupts in user programs. In both Topaz and the SunOS LWP Library, a single thread is designated to receive signals of a particular type on behalf of all threads in the process. Signal-receptor threads are dedicated to this purpose; they do not perform computational work as well. Thus worker threads are immune from interruption, though exception handlers may be defined to handle synchronous program errors. In the future, Mach will use a similar mechanism. Currently, signals in Mach are handled pseudo-randomly by any thread in the process [Tev87a].

CHORUS/MIX adopts a new approach to signal delivery and management. The goal is simple: *each signal should be processed by (i.e., on the stack of) the u_threads which have indicated that they want to process that signal.* Each thread has its own signal context (signal handlers, blocked signals, etc.) and system calls used to manage this information affect only the calling thread. This design arose from considerations of common signal usage in existing mono-threaded programs, and the desire to keep the same semantics in multi-threaded programs.

- Many applications use the longjmp non-local branch to abort a computation loop on receipt of a signal. This can work correctly only if the handler executes in the thread which is to be interrupted.
- When a terminal user types the control sequence which aborts the current foreground process, a certain type of signal is sent to that process. In a multi-threaded process, where one thread writes to or reads from the terminal while other threads perform computation, the programmer should be able to decide which thread(s) will process this signal: certainly the thread that interacts with the terminal, but possibly some of the other threads as well.

Furthermore, the per-thread signal design seems to ease programmer difficulties and maximize flexibility.

- If signal handling were assigned to only one u_thread at a time, the programmer would have to manage the complexity of redispatching caught signals to other u_threads.
- While asynchronous interrupts are sometimes difficult to manage, concurrency also increases the level of complexity, and in fact the issues are similar. The programmer must deal with asynchronous access to shared variables, and the resulting race condition bugs, in both cases. Programming a simple signal handler can be much more straightforward than coordinating threads, especially in a section of program that is not otherwise using concurrency.
- Signals provide a low-overhead, minimalist mechanism that can be used to implement any desired semantics for a specific language or programming environment. Standard library signal handlers may transform a delivered signal into a message, create a new thread to respond to an asynchronous events, or pass on the signal to another thread, as desired.

In order to determine which u_threads should receive signal delivery, CHORUS/MIX distinguishes two types of signals:

(1) *Signals for which the identity of the target u_thread is non-ambiguous.* These are signals corresponding to synchronous exceptions, and also those issued in response to a system call issued by a specific u_thread. Signals for expiration of a timer interval or completion of an I/O operation are in this category. In these cases the signal is sent only to the u_thread concerned. If that thread has not declared a handler or opted to ignore the signal, the default action (usually termination) is taken for the entire process.

(2) *Signals which are logically sent to an entire process.* For some purposes a process is viewed as a single entity. Signals sent by device drivers (e.g., the keyboard "abort process" sequence), and signals directed to a process by a user program, are broadcast to all u_threads of the target process. Each u_thread which has declared a signal handler for the particular signal will receive delivery. If no u_thread either declares a handler or ignores the signal, the default action is again taken relative to the entire process.

Thus signal semantics and usage extend smoothly from the traditional UNIX mechanisms to the multi-thread environment of CHORUS/MIX. The per-u_thread signal status adds to the size of the state information that must be initialized and maintained for each u_thread. However, no extra expense is added to the u_thread context switch, so the real-time responsiveness of the system is not compromised.

Signals are used to perform one new function in CHORUS/MIX: an inter-thread interrupt facility. Any u_thread may send a signal to another u_thread of the same process using a new system call. Designers of previous thread systems have preferred to introduce new facilities for inter-thread notification. The *alert* facility of Topaz, for example, effectively sets a flag which is then polled by various coordination functions in the target thread. However, the signal interface already exists and cannot be suppressed. It is simpler to provide a single mechanism than to add a new concept whose semantics would complicate the signal semantics. Again, individual programming environments may adopt their own conventions for inter-thread interrupts.

6.2. Thread context

A final compatibility problem affects UNIX programs and libraries that are written in C. Variables in C programs which are declared as global or static are bound to a single storage location in a process. Hence in a multi-threaded process, those variables become globally shared. However, they are sometimes used in a manner that requires a separate instance of the variable in each thread. The best-known example in UNIX is the *errno* variable, which reports error codes returned from the most recently executed system call. Other examples occur in library packages which must maintain state information between successive calls in a thread. The C language makes no provision for thread-private instances of global or static variables. (Automatic variables are thread-private because each thread has its own private stack.)

There are a number of ways of addressing this deficiency.

- A different language, with support for private thread context, might be used.
- A pointer to a u_thread-private data area might be passed as an argument to all routines, either explicitly or implicitly as with C++ member functions. This solution is awkward and potentially expensive.
- Private u_thread information could be maintained at a fixed location in the process address space, which is remapped on each context switch. As discussed earlier, this could add considerable expense to the context switch and sacrifice much of the advantage of threads, especially in real-time applications.
- A nucleus facility could be invoked on each access to obtain the current thread identifier, which could then be combined with a hashing scheme to obtain the address of thread-private storage.
- A hardware register might be reserved to hold the address of this private area. This is not feasible on all processors, and in any case requires that compilers be modified.

The solution in CHORUS/MIX has been to extend the processor context with a set of *software registers*. These are implemented in the CHORUS nucleus; one register is provided per privilege level, as is done for stack pointers. The software registers are saved and restored at each context switch. In the usual implementation, with two privilege levels (user and supervisor), the added cost on context switch is equivalent to a copy of four pointers. Each u_thread may read and modify the value of the software register at the corresponding privilege level. Where possible, the reading of these software registers is implemented inline, without a trap to the system.

Several independent routines in a library may need thread-private static storage within a single thread. Hence a package is provided in the C library to manage multiple uses of an individual software register. Macros and inline functions are used to minimize expense and facilitate modification of library routines for multi-threaded use. No modification at all is required in user programs which merely access *errno* or similar standard variables.

7. Conclusion

Extending a general-purpose operating system to support concurrent programming requires careful attention both to performance requirements and to the compatibility and usability of the system interface. Within the framework of a distributed and parallel environment, the particular design goals for thread support in CHORUS/MIX were the following.

- *To satisfy real-time needs.* u_threads are implemented and scheduled by the operating system according to standard requirements of real-time computing: deterministic and responsive behavior with respect to priorities, scheduling, and synchronization. The CHORUS/MIX design differs substantially from that of other general-purpose thread-management systems.
- *To achieve a graceful integration with standard UNIX semantics.* Multi-threaded CHORUS/MIX processes are still UNIX processes; each standard UNIX feature has been extended in a coherent manner, as required. The major focus was on signal handling and parallel invocation of system calls. Our signal handling design is deterministic and gives the user full control over signal behavior in a multi-threaded program. This design is again a departure from earlier systems that augment UNIX with concurrency features.
- *To support parallel architectures.* u_threads can be used along with user mode scheduling to provide efficient parallelization of a wide range of granularities of tasks in compute-intensive applications.

Work is still in progress in two important areas. First, we must develop low-cost synchronization methods which allow programs to scale to large number of processors while satisfying the determinism requirements of real-time programs. Finally, we have entirely omitted the crucial area of debugging support in this article. Operating system features in support of debuggers (including remote debuggers) for multi-thread programs are currently under investigation and experimentation.

8. Acknowledgements

Many thanks to Vadim Abrossimov, Michel Gien, Marc Guillemont, Marc Maathuis, Will Neuhauser and Francois Saint Lu who have contributed, each with a particular skill, to the CHORUS/MIX u_thread management specification and implementation.

References

[Bec87a] Bob Beck and Dave Olien, "A Parallel Process Model," *Proc. USENIX*, pp. 83-101, September 1987.

[Bir89a] Andrew D. Birrell, "An Introduction to Programming with Threads," *Technical Report 35*, DEC-SRC, Palo Alto, CA, January 1989.

[Coo87a] Eric C. Cooper and Richard P. Draves, "C Threads," *Technical Report*, Carnegie Mellon University, Pittsburg, PA, March 1987.

[Dij68a] E.W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, ed. F. Genuys, Academic Press, New York, 1968.

[Edl88a] Jan Edler, Jim Lipkis, and Edith Schonberg, "Process Management for Highly Parallel UNIX Systems," *Proc. USENIX Workshop on UNIX and Supercomputers*, September 1988.

[Hoa74a] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, vol. 17, no. 10, October 1974.

[McJ88a] Paul R. McJones and Garret F. Swart, "Evolving the UNIX System Interface to Support Multithreaded Programs," *Technical Report 21*, DEC Systems Research Center, Palo Alto, CA, September 1988.

[Sun87a] Sun Microsystems, "Lightweight Process Library," *SunOS Release 4.0 Reference Manual*, Sun microsystems, Mountain View, CA, November 1987.

[Ous82a] John K. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proc. 3rd International Conf. on Distributed Systems*, pp. 22-30, 1982.

[Roza] Marc Rozier, Vadim Abrossimov, Francois Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frederic Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Leonard, and Will Neuhauser, "CHORUS Distributed Operating Systems," *Computing Systems Journal*, vol. 1, no. 4, pp. 305-370.

[Swa89a] Garret Swart and Roy Levin, "An Introduction to Threads and CMA," Presentation to IEEE POSIX P1003.4 Realtime Extension for Portable Operating System, DEC Systems Research Center, Palo Alto, California, January 1989.

[Tev87a] Avadis Jr. Tevanian, Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young, "Mach Threads and the UNIX Kernel: The Battle for Control," *Technical Report CMU-CS-87-149*, Carnegie Mellon University, Pittsburg, PA, August 1987.

Distributed Computing in Heterogeneous Environments

Herman Moons,

Pierre Verbaeten

Katholieke Universiteit Leuven
Dept. of Computer Science
Leuven, Belgium

herman@cs.kuleuven.ac.be

Ulf Hollberg

IBM European Networking Center
Heidelberg, Germany

ABSTRACT

Distributed computing systems have received considerable attention in the last decade. Unfortunately, current research efforts are often restricted to homogeneous environments. There seems to be little attention for real-world installations, where heterogeneity is mostly the rule, rather than the exception.

The *Distributed Academic Computing Network Operating System*, DACNOS for short, presents a general solution for running distributed applications in heterogeneous networks. The DACNOS extends local guest operating system services to provide homogeneous networking functionality. This is achieved by a virtual global object space, to which each user has access from within his native environment.

This paper describes the architecture of the DACNOS network operating system, with special emphasis on its incarnation on the UNIX platform. It presents the DACNOS solutions to the problems of communication, access protection and data representation in a network of cooperating heterogeneous systems. DACNOS implementations currently exist for VM/CMS, VAX/VMS, PC-DOS, OS/2 and UNIX System V derivatives.

1. Introduction

When looking at the current situation in the computing world, the following observations can be made. Major computer users (e.g. banks or universities) have built up a heterogeneous collection of hardware and software, which represents a large investment. Large organizations experience a growing need for the sharing of information and resources.

The Distributed Academic Computing Network Operating System (DACNOS) provides a general framework for distributed computing in such a heterogeneous environment.

- On the one hand, DACNOS offers an extensive set of new primitives, that form an excellent basis for the development of *distributed applications*. These primitives handle the subtle problems of data representation and communication, so that the programmer can concentrate on their particular application.
- On the other hand, DACNOS extends the local guest operating systems in a way, that the same program can use local and DACNOS services. This *coexistence* enables a special class of applications to provide *transparent access* to remote resources through the native interfaces of the local system. As a result, existing software will continue to run as before, but now has access to shared remote resources throughout the network. Furthermore the interface to the guest operating system remains unchanged, so users don't need to learn yet another system. In section 6, we will present as an example the Remote File Access (RFA) component of DACNOS; it implements transparent access to remote and potentially heterogeneous files.

In this paper, we present the architecture of the DACNOS system, with particular emphasis on its realization in the UNIX environment. Section 2 gives an overview of the DACNOS components. In section 3, we discuss the *portability environment* that allows for easy migration of DACNOS to different types of guest operating systems. The next section presents the *Remote Service Call*, which provides the means to perform truly distributed computing in a network of heterogeneous systems. Section 5 introduces DACNOS system services like the distributed directory and the accounting services. In section 6 we illustrate how transparent access to remote resources can be achieved. We end the paper with an evaluation of the DACNOS design.

2. The Architecture of DACNOS

From the user's point of view a DACNOS network consists of a completely connected graph of *logical nodes*. Logical nodes are abstractions of physical machines, with two important properties: they have a unique network address, and they are completely contained within one physical machine. The developer of DACNOS for a specific type of guest operating system decides how logical nodes map onto real objects.

The DACNOS kernel is present on every logical node in the network. To facilitate software development, the kernel is designed following a layered approach. This has two obvious advantages: each layer forms a self-contained unit, and can be designed and implemented by a separate team. Hardware and guest system dependencies are dealt with in the lower layers, so that upper layer software is completely portable.

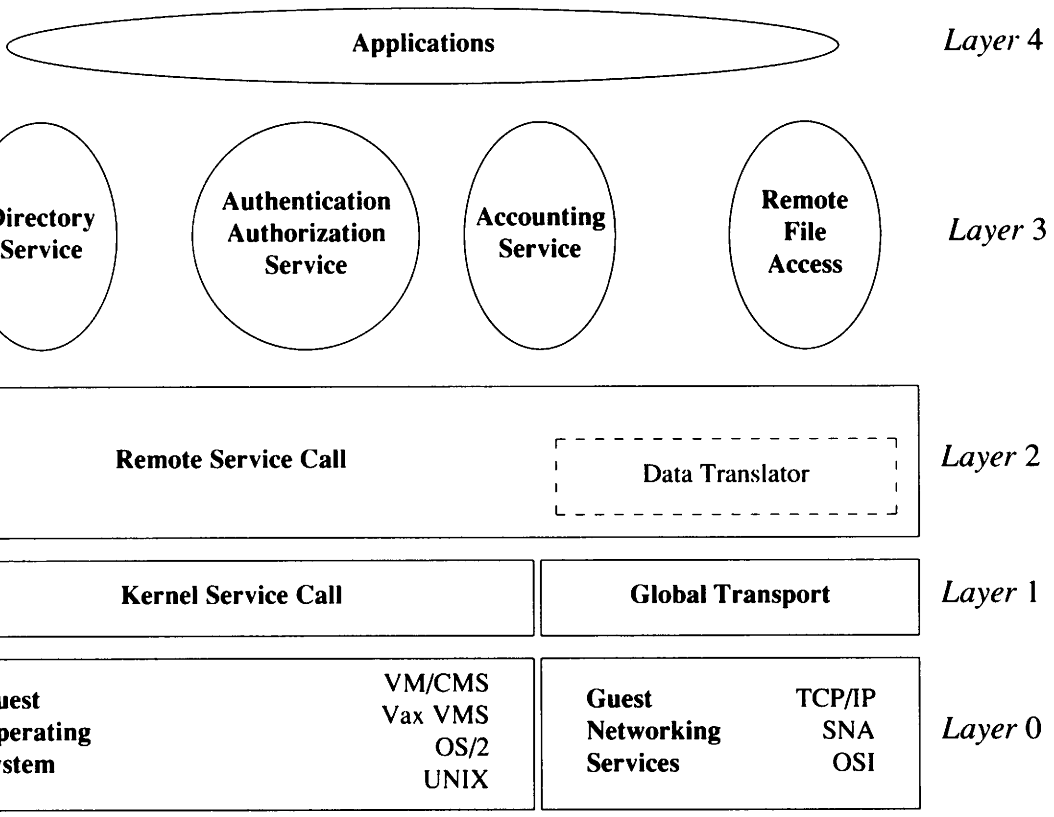


Figure 1: *The DACNOS system components*

As shown in Figure 1, the DACNOS architecture consists of five layers:

- **Layer 0: Guest Operating System**

  This layer is of its very nature machine and guest system dependent. The key point here is that DACNOS supports a wide variety of layer 0 components. Hardware environments go from personal computers (like the IBM PC/AT or PS/2), over departmental computers (like the DEC Microvax or SUN 3/50), to mainframes (like the IBM 3090). Guest operating systems range from single-user single-tasking ones (like VM/CMS) to multi-user multi-tasking systems (like UNIX or VAX/VMS). This heterogeneity is a distinctive characteristic of the DACNOS project, which sets it apart from other research projects in the area of distributed systems.

- **Layer 1: Portability Environment**

  The purpose of the Portability Environment is to mask the heterogeneity of layer 0, by presenting upper layers with a *homogeneous interface to operating system services.* The presence of this layer simplifies portation of the DACNOS to other guest systems. Ideally, only the Portability Environment has to be reimplemented, while the other DACNOS components require only recompilation. Layer 1 consists of two subcomponents: Kernel Service Call (KSC) and the Global Transport (GT). The KSC is responsible for providing operating system services in a host independent way. GT is responsible for the transmission of messages over underlying network services.

- **Layer 2: Generic Support for Distributed Processing**

  This layer introduces the notion of a *shared global object space with access protection,* as a proper substitute for globally shared memory. It consists of two subcomponents: the Remote Service Call facility (RSC), and the Presentation System (PS). RSC is basically a protocol machine, that regulates access to objects residing at logical nodes, and provides the illusion of one shared global object space. The RSC is guest system independent, because it uses only layer 1 services. As a result, porting RSC to a new type of system is a straightforward exercise. The Presentation System is needed to handle the problems of data description and data transformation that are inevitable in a heterogeneous environment. It consists of a compiler and an interpreter. The compiler takes user-supplied data descriptions, and converts them to an internal form. The internal form is used by the RSC protocol machine and the PS interpreter to convert data, where necessary.

- **Layer 3: DACNOS System Services**

  This layer consists of a number of servers that offer system management services and resource sharing. These servers don't have to reside at each logical node. It suffices when they are available at some place in the DACNOS network. The more important System Services are *Remote File Access, Authentication/Authorization Service, Accounting Service* and the *Directory Service.* They all use layer 2 and local services, which greatly simplified their implementation.

- **Layer 4: Application Programs**

  This layer contains all user-developed software that makes use of the DACNOS services. The design and implementation of distributed applications is greatly simplified because the DACNOS kernel takes care of the idiosyncrasies associated with communication, object location, access protection and data presentation. Equally important, user-software that relies solely on DACNOS primitives is totally portable across heterogeneous guest systems.

3. The Portability Environment

The goal of the *Portability Environment* (PE) is to achieve portability by separating the bulk of the DACNOS modules from the guest operating system. The PE is a thin layer, that provides the services needed from an operating system in a system independent form. It consists of two subcomponents: the *Kernel Service Call* (KSC) and the *Global Transport* (GT).

KSC is responsible for providing operating system services in a host independent way. It provides light-weight processes (threads), together with efficient inter-thread communication and synchronization mechanisms. The interface to KSC is expressed in terms of KSC objects and operations on these objects. A detailed discussion can be found in [Sta88a] and [Sta87a].

Because UNIX systems have no support for multiple threads in a single address space, we implemented a light-weight process package with preemptive priority scheduling. Rescheduling occurs whenever a service call is made, or when a UNIX signal arrives (e.g. a SIGALRM signal when a timer expires). KSC objects are stored in a shared memory segment. With this organization KSC objects have a lifetime independent of a single UNIX process.

Global Transport provides a simple unified interface for the reliable exchange of messages between heterogeneous logical nodes. It offers a maximum of connectivity, integrating local and wide area networks being run with private and/or public protocols. The GT interface is seen as two KSC ports: the *SendPort* (SP) to which messages can be sent, and the *ReceivePort* (RP) from which a logical node can receive messages. All the details of routing the messages to their intended destination are taken care of by the Global Transport System. Messages contain a header that specifies the address of the destination logical node. The addresses conform to the OSI layer 4 Transport Service Standard [Int85a].

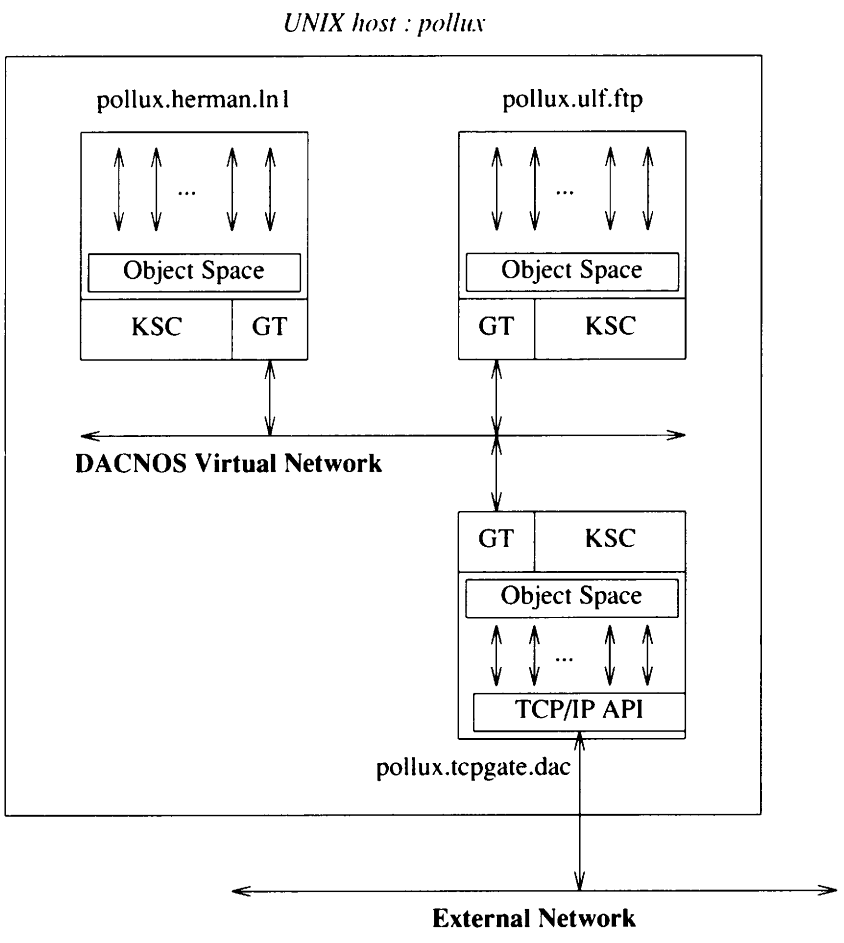


Figure 2: *Logical Nodes in a UNIX environment*

On UNIX, they are of the form *"host.user.node"*. The *host* part of the address is used for routing between different machines. Multiple logical nodes can coexist on the same host. Nodes on the same machine communicate through a *DACNOS Virtual Network*, implemented by means of a device driver. Communication with other machines is realized by means of a gateway logical node, that connects the Virtual Network to external networks. The *user.node* part of the GT address is used for distinguishing logical nodes connected to the Virtual Network.

Figure 2 shows a schematic view of the organization of logical nodes within a single UNIX host. Communication with the outside world is achieved through a gateway that connects to TCP/IP.

4. Generic Support for Distributed Processing

The *Remote Service Call* (RSC) component of the DACNOS kernel provides a uniform interface for the development of distributed applications based on the *client/server* model. From the programmer's point of view, RSC maintains objects in the network in the same way they are available on the local host. RSC thus provides the functionality of a local operating system in a distributed environment. To achieve this, the following features were incorporated into its design:

- location, presentation and access transparency of objects,
- asynchronous cooperation,
- enforcement of node autonomy and resource usage monitoring.

Distributed Object Sharing and Access Protection

The RSC design philosophy is based on the idea of *object sharing*. The absence of shared memory in a distributed environment has frequently been identified as a major inconvenience [Car86a, Che86a]. RSC provides an alternative for shared memory by implementing the concept of *distributed object sharing*. Certain RSC objects can be shared between distributed processes, analogous with the sharing of memory in the local case.

When sharing objects in a distributed environment, rigorous access protection mechanisms have to be integrated into the design. In RSC, an object can only be accessed if the necessary *access rights* are held by the process requesting the access. An access right to an object can be thought of as a "pointer", which can be passed to other processes in the distributed system instead of passing the object itself.

RSC provides two complementary sharing semantics, that control the transfer of access rights between client and server:

- Explicit *OFFER* and *SHARE* operations represent a long-term binding between client and server. The server uses the *OFFER* operation to make an object available to other processes in the DACNOS network. Clients issue a *SHARE* operation to get access to the offered object.
- Access rights passed in a *Service Call* represent a short-term binding between client and server. In this case the access rights carried with the request are lost as soon as the request has been processed and returned back to the client.

The owner of an RSC object has full control over all accesses to his objects. Granted access rights can be revoked at any time of the client/server relationship by means of *RETRACT* and *DELETE* operations.

Kernel Objects and Operations

The DACNOS RSC component provides a basic set of objects, from which user-defined, abstract objects can be built. In RSC, the *Port* object provides the means for hiding the implementation of a service behind a well-defined interface. Some of the RSC objects have the *event attribute*. Processes can wait "at these objects" for an object specific event to occur. The objects provided by the RSC kernel are summarized in the following table.

Object	Function	Sharing	Event
Process	issues operations on objects	-	-
Port	is a queue for service requests. Multiple clients may concurrently request the same service, multiple servers may provide the service	1:n	Carrier or Notice arrived
Carrier	passes a service request with temporary access rights to a server	1:1	end of service
Notice	is a one-way message	-	-
Window	is used to grant access to virtual memory	1:n	-
Account	keeps information about resource usage and dispatching parameters. It identifies clients to servers in processes and carriers	1:n	-
Lock	provides synchronisation of processes	1:n	complete
Event-List	supports wait on a collection of events	-	1 event complete

Figure 3: *RSC Kernel Objects*

- The *Process* is light-weight and bound to a logical node. The *Lock* is a globally sharable binary semaphore. The "one-way" *Notice* was introduced because in some situations the always "two-way" *Carrier* would be a waste of resources.
- The *Port* is the "meeting point" between clients and servers. It has two queues: the request queue and the waiting-processes queue. Several requests (Carriers or Notices) may be queued at a Port, and multiple Processes may be waiting at the Port for incoming requests. A Port can be made public by OFFERing it to clients, or it can be passed privately in a Service Call with predefined access rights. In the first case the Port represents the abstraction of a public service (e.g. a file server). In the second case the Port is typically associated with a high level object (e.g. an open file).
- The *Carrier* object represents a *Service Call* to a server. It is analogous to the parameter list of a local procedure call. The Carrier may contain access rights to sharable objects that the sender makes available to the server for the duration of the call. After servicing the request, the server will return the Carrier to its origin (at which time it loses access to the objects that were sent with the Carrier). The client can wait for the return of the Carrier.
- The *Window* object represents a contiguous section of main memory. A Window can be shared, and is used to exchange large amounts of user data between cooperating Processes by means of the *READ-WINDOW* and *WRITE-WINDOW* operations. Windows are typed, which makes it possible to transparently handle the data conversions that are unavoidable in a heterogeneous environment.
- The *Account* object makes it possible to do accounting in a distributed system, where local accounting services are no longer sufficient. An *Account* is automatically created at Process creation time, and is appended to all Service Calls issued by this Process. The server shares the Account, making it possible to charge the provided services to the client's Account.
- The *Event-List* is available to provide Processes with a "multiple-wait" facility. The Event-List may hold several event-type objects. The first *complete* event in the Event-List will resume a waiting Process.

The RSC objects are manipulated through well-defined RSC operations. Some of these operations are applicable to several types of objects. Each object has an associated *CREATE* and *DESTROY* operation. Access rights to sharable objects can be obtained through *OFFER/SHARE* operations, or through a Carrier by means of a *REQUEST* (a Service Call) to a Port. For event-type objects there is a *WAIT* operation. Depending on the object type there exist several other object-specific operations. Details on these operations can be found in [Ebe88a].

The Service Call

We are now in a position to take a closer look at a typical service request cycle, involving Ports, Windows and Carriers. A Carrier is a structured message representing a service call, carrying parameters (by *value* and by *reference* and an *Account* object. The request cycle normally starts by a client that gains access to a server port. This is illustrated in figure 4a.

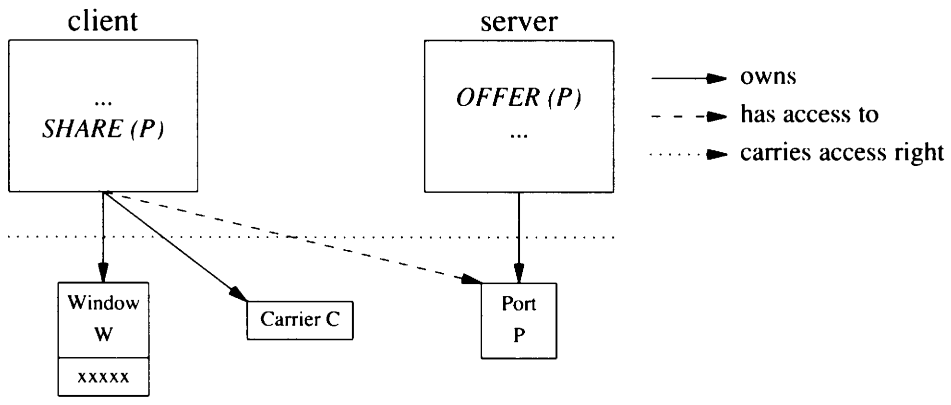


Figure 4a: *Service Call Setup*

In the example, the server makes its Port P public by means of an *OFFER* operation. The client gains access to the server's port by means of a *SHARE* operation. From that point on the actual service call can proceed. The client places access rights – in this example "write" – to its Window W in the Carrier C (a reference parameter). It *sends* the Carrier to the server Port. When the Carrier arrives, the server wakes up and receives the Carrier. It can now extract the Window access rights from the Carrier. The situation at this point is illustrated in figure 4b.

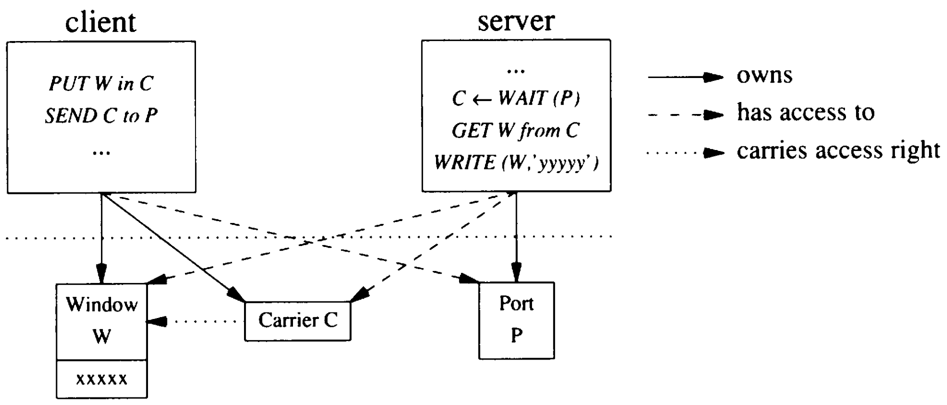


Figure 4b: *Service Call Request Processing*

The server now shares the client's Window W, and can change its contents. After the processing of the service call is complete, the server *returns* the Carrier. At this moment the server loses all access to objects contained in the Carrier. The client (waiting for the Carrier to return) wakes up, and can read the modified Window contents. The situation at this point is illustrated in Fig 4c.

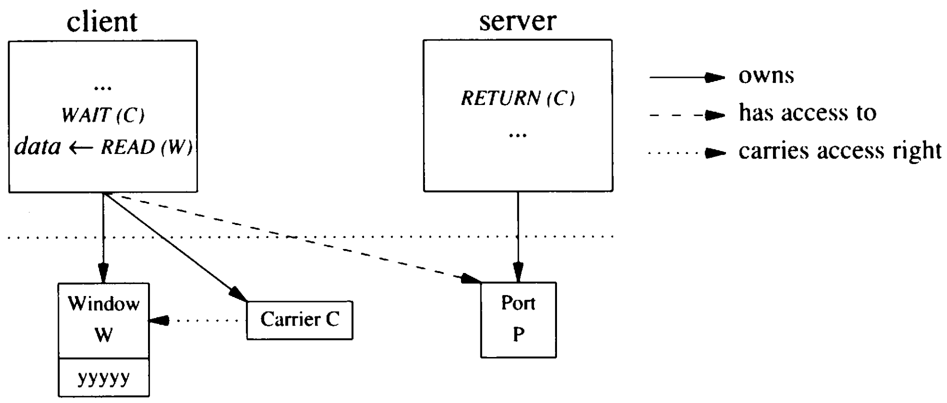


Figure 4c: *Service Call Terminated*

It is important to note that each request is represented by a separate Carrier object. This means that the client can individually control each request it makes. The client may even *revoke* a particular request at any time, which will delete the access rights contained in the Carrier.

The asynchronous service call, together with the multi-wait Event-List and the multiple waiters per Port, turn the RSC interface into a very flexible set of tools for the implementation of general distributed applications. A detailed discussion of all aspects of RSC can be found in [Gei86a, Ebea] and [Ebe88a].

Data Representation

The object oriented nature of the RSC significantly eases the presentation management. Format descriptions for sharable objects are attached to an object when it is created, or are dynamically associated with the object by means of appropriate RSC operations. The syntax description for service calls, i.e. presentation information on the objects in a Carrier, is bound to the Port object of the server. Whenever data are communicated between heterogeneous nodes, the RSC kernel uses the data descriptions to invoke the corresponding data transformation routines (transparent to the application).

Application designers use the NATRAS data description language [Wil87a, Ebe87a] to describe the format of the information to be communicated. NATRAS is derived from ASN.1 [ISO8825a], a description language introduced by CCITT and accepted by ISO for the description of application layer protocols. A compiler is available to translate NATRAS descriptions to the internal form used by the RSC kernel. Figure 5 gives an example of a personnel record in Pascal, together with the corresponding NATRAS description.

Pascal description

```
const
        NamLen = 5;

type
        Name = array [NamLen] of char;

Person  = record
                        name : Name;
                        first_name : Name;
                        age : integer;
                end;
```

NATRAS description

```
PersonnelRecord DEFINITIONS  ::=
BEGIN
    CONSTANT
            NamLen ::= 20
    TYPE
            Name ::= SEQUENCEOF (NamLen) CHAR(ASCII)

    TRANSFER Person ::=SEQUENCE (*
                                    name Name
                                    first_name Name
                                    age SHORTINTEGER
                                *)
END
```

Figure 5: *Example NATRAS data description*

5. The DACNOS System Services

In this section we give a short overview of the following DACNOS system services: Authentication & Authorization, Directory and Accounting.

Authentication/Authorization

In a networking environment with hosts owned by different organizations, access control to resources is of prime importance. Each time a service is requested by a remote client, the server must be able to verify whether the invoker has the right to do this (authorization). This is only possible if every DACNOS user can be identified in a unique and non-forgeable way (authentication).

To this end, a network-wide protection mechanism is integrated with the DACNOS system. The *Authentication/Authorization Server* (AAS) checks the identity of a DACNOS user at login time (using appropriate measures like passwords). A successful login results in a *<userid,nodeid>* pair that is stored within the AAS, and remains in place until the user leaves the system (logout).

The Authorization Service maintains lists of *user groups*. Each user group contains the userids that belong to this group. When a user issues a service call, its logical node address is passed as a hidden, non-forgeable parameter with the Carrier. The server maintains an *access list* that describes the user groups that are authorized to issue requests. The server now presents the logical node address of the client, together with the access list to the Authorization Service. The latter authorizes the request if the userid corresponding to the logical node address is present in the access list. A more detailed discussion of the AAS service can be found in [Mat88a].

Directory

The DACNOS system contains a universal *Directory Service*, capable of storing knowledge about the services and states that are present in the distributed system [Mat88b]. A number of Directory Servers (DS) are active in the network. Every logical node is assigned to a particular DS. As a result, the network is divided into *domains*. Directory entries have unique names, consisting of a domain name and a unique subname within this domain. Each domain contains a *name service* that ensures the uniqueness of names within its domain.

Directory entries are tuples of arbitrary structure. Each tuple has a type and attributes. Tuple, type and attributes can be protected. Searches in the directory are performed by means of predicates over tuples, analogous to queries in a relational database.

Accounting

With each service call, a hidden Account object is passed to the server. The server can then accumulate resource usage data into the client's Account object. After a service request, the Account object is returned to the client together with the Carrier. A copy of the account is passed to the Account Server, that uses the accounting information to bill the clients [Har88a].

6. Transparent Access to Remote Resources

The Remote File Access (RFA) component of DACNOS offers transparent access to remote files throughout the heterogeneous networks of DACNOS. Remote files can be mounted in a way that allows access to them through the interfaces of the local file system.

RFA Servers implement a homogeneous global file system. They offer storage (*file store*) for global files, and mediate access to *published* files of the local file system. Published files are owned by the server until they are retracted. RFA Servers can be configured by their administrators.

In RFA, global files have hierarchical names. Files are grouped together into *file sets* by common prefixes of their names. Clients of RFA mount file sets into their local environment (see command "*mount UKA.DAC02...*" in figure 6).

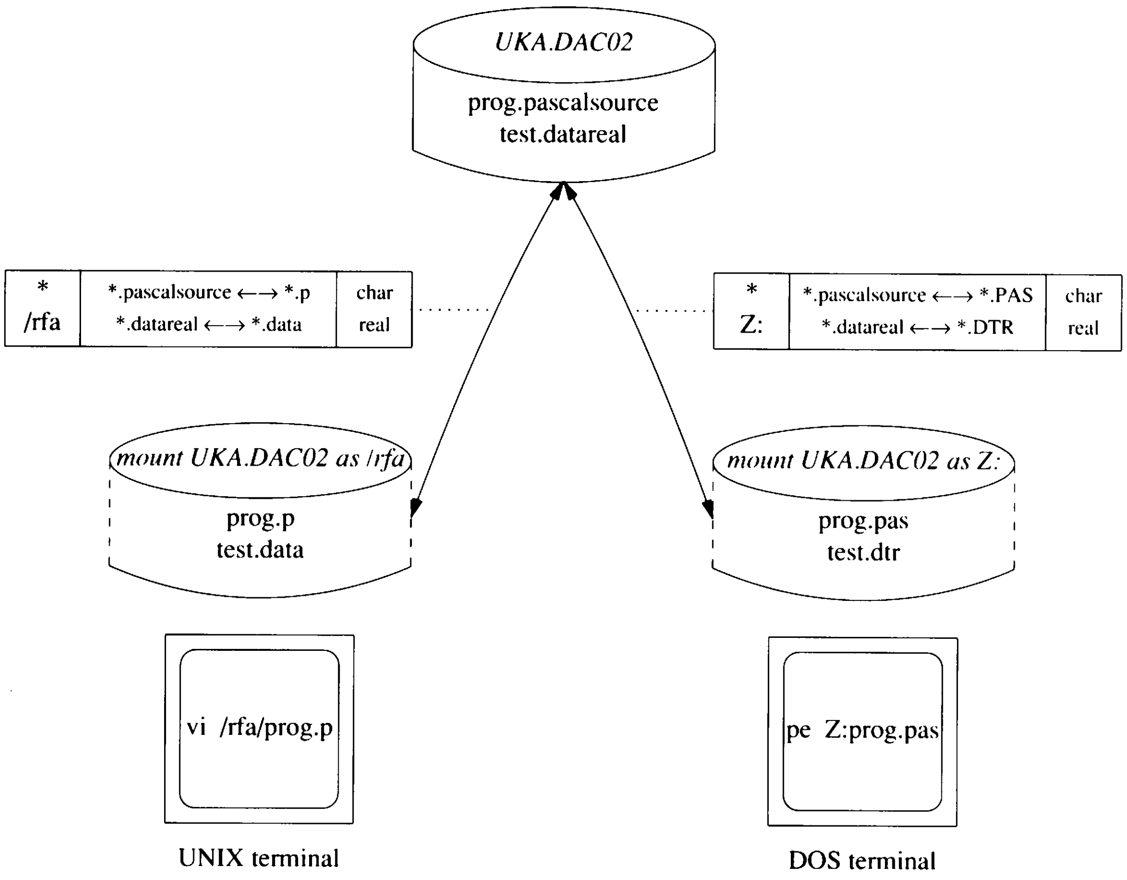


Figure 6: *RFA name translations*

RFA File Sets are represented by the corresponding counterparts of the local systems: (sub)directories in UNIX and VMS, disks in PC DOS, or minidisks in CMS. At mount-time, the suffixes of the global names are translated into local file names, that conform to the syntax of the local file system. The translation process is controlled by a set of user defined rules as sketched in the boxes of Figure 6 [Hol88a]. Then, remote files appear as local files with local names and are accessible through the local file system interface.

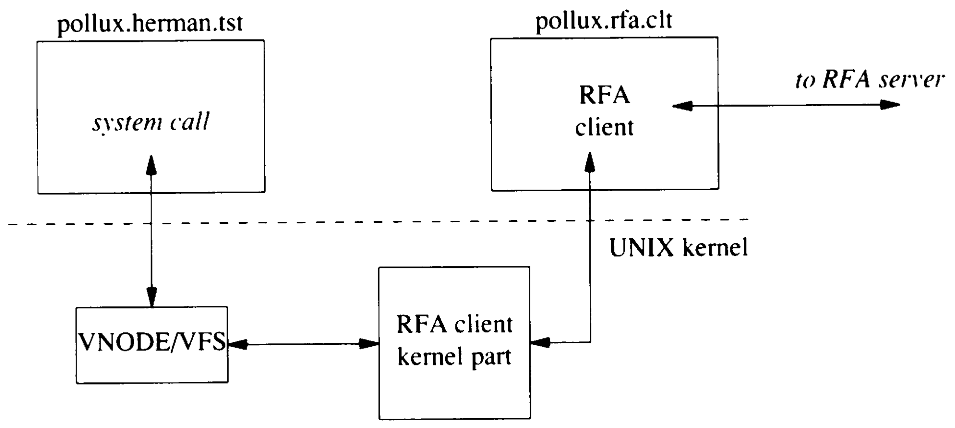


Figure 7: *RFA implementation on UNIX systems*

This transparent access can only be achieved, if the local file system supports extensions or interception points, like the *vnodes* interface of some UNIX implementations [Kle86a]. In some other cases, a similar support can be build into the local system; this has been done for PC DOS, VM/CMS and VMS. On UNIX systems the vnodes interface can be used as shown in figure 7.

File system calls referring to remote files via local names are routed to the vnodes client component of RFA. This code interacts with the RFA client process, which in turn communicates via RSC with the RFA server processes. The results are passed back to the user via the vnodes interface and the UNIX kernel.

7. Evaluation and Conclusion

The DACNOS system has been designed to ease the development of distributed applications by adding functionality to existing local operating systems. A distinctive characteristic of the DACNOS is that it provides general solutions to the problems of distribution, access protection and data presentation, that would otherwise have to be solved by application designers again and again.

Compared to other research projects, the DACNOS approach provides a more comprehensive way to tackle the *"distributed computing"* problem. The V-kernel [Che86a] focuses on efficient communication, but does not provide primitive objects that are suitable for building abstract high-level objects. Desperanto [Mam85a] has similar goals as RSC, but does not provide a general concept for passing protected abstract objects to other processes. Closest to the DACNOS design goals come the Cronus [Gur86a] and HCS [Bla87a] projects. These systems also attack the problems of heterogeneity and preservation of existing interfaces. Differences exist in the degree of transparency, enforcement of access control and support of resource management.

The experience with the DACNOS prototype proved that the development of distributed applications is almost reduced to the complexity of local program development. A number of non-trivial end-user applications were developed to demonstrate this statement. A first application offers access to an SQL/DS database from any node in the network. A second one provides a Remote Procedure Call service to access the ACRITH FORTRAN subroutine library. Using the application program interface of RSC and the System Services, these distributed applications were developed by students within a few months.

References

[ISO8825a] ISO/DIS 8825, *Information Processing Systems — Open Systems Interconnection: Specification of basic encoding rules for Abstract Syntax Notation One*, International Organisation for Standardization, 1985.

[Bla87a] E. Black, et al., "Interconnecting Heterogeneous Computer Systems," Technical Report 87-01-02, University of Washington, Dept. of Computer Science, 1987.

[Car86a] N. Carriero and D. Gelernter, "The S/Nets Linda Kernel," *ACM TOCS*, vol. 4, no. 2, pp. 110-129, 1986.

[Che86a] D.R. Cheriton, "Problem-oriented Shared Memory: a Decentralized Approach to Distributed Systems," *in Proceedings of the 6th International Conference on Distributed Computer Systems*, IEEE, 1986.

[Ebe87a] H. Eberle, K. Geihs, and B. Schoener, "Data Presentation Service Workbook," DAC Technical Memorandum, No.27, IBM European Networking Center, 1987.

[Ebe88a] H. Eberle, K. Geihs, and B. Schoner, "Remote Service Call: Object and Operation Reference," Internal Report, IBM European Networking Center, Heidelberg, September 1988.

[Ebea] H. Eberle, K. Geihs, A. Schill, B. Schoener, and H. Schmutz, "Generic Support for Distributed Processing in Heterogeneous Networks," in *HECTOR: Heterogeneous Computers Together: A Joint Project of IBM and the University of Karlsruhe*,, ed. G. Krueger, G. Mueller, vol. II.

[Gei86a] K. Geihs, H. Eberle, B. Schoener, and M. Seifert, "Distributed Object Sharing in Heterogeneous Environments," Technical Report No. 8610, IBM European Networking Center, Heidelberg, 1986.

[Gur86a] R.F. Gurwitz, M.A. Dean, and R.E. Schantz, "Programming Support in the Cronus Distributed Operating System," *in Proceedings of the 6th International Conference on Distributed Computing Systems*, IEEE, 1986.

[Har88a] G. Harter and K. Geihs, "An Accounting Service for Heterogeneous Distributed Environments," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 207-214, San Jose, California, June 13-17, 1988.

[Hol88a] U. Hollberg and E. Kraemer, "Transparent Access to Remote Files in Heterogeneous Networks," in *HECTOR: Heterogeneous Computers Together: A Joint Project of IBM and the University of Karlsruhe*, ed. G. Krueger, G. Mueller, vol. II, Springer-Verlag, 1988.

[Int85a] ISO Open Systems Interconnection, "Transport Service Definition," Document IS8072, 1985.

[Kle86a] S. R. Kleinman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," in *Usenix Conference Proceedings*, Atlanta, 1986.

[Mam85a] S.A. Mamrak, D.W. Leinbaugh, and T.S. Berk, "Software Support for Distributed Resource Sharing," *Computer Networks and ISDN Systems*, vol. 9, pp. 91-107, 1985.

[Mat88a] B. Mattes, "Authentication and Authorization in Resource Sharing Networks," in *HECTOR: Heterogeneous Computers Together: A Joint Project of IBM and the University of Karlsruhe*, ed. G. Krueger, G. Mueller, vol. II, pp. 126-139, Springer-Verlag, 1988.

[Mat88b] B. Mattes, "Directories and Orientation in Heterogeneous Networks," in *HECTOR: Heterogeneous Computers Together: A Joint Project of IBM and the University of Karlsruhe*, ed. G. Krueger, G. Mueller, vol. II, pp. 110-125, Springer-Verlag, 1988.

[Sta87a] R. Staroste and H. Eberle, "Kernel Service Call: a Multitasking Facility for Heterogeneous Environments," Technical Report No. 8701, IBM European Networking Center, Heidelberg, 1987.

[Sta88a] R. Staroste, H. Schmutz, M. Wasmund, A. Schill, and W. Stoll, "A Portability Environment for Communication Software," in *HECTOR: Heterogeneous Computers Together: A Joint Project of IBM and the University of Karlsruhe*, ed. G. Krueger, G. Mueller, vol. II, pp. 51-79, Springer-Verlag, 1988.

[Wil87a] G. Wild and M. Zoeller, "Darstellungsproblematik fuer heterogene verteilte Systeme," in *Proceedings of the GI/NTG Conference on Communication in Distributed Systems*, Aachen, West-Germany, 1987.

AIX Version 3 on the RISC System/6000 Family

Iain Elliot

IBM Deutschland GmbH

ABSTRACT

The Advanced Interactive Executive (AIX) Version 3 UNIX kernel was designed to be portable across a wide range of architectures from the Intel 80x86 microprocessor family up to the IBM System/370-XA architecture mainframes. In several areas, however, the AIX Version 3 software was optimised to make full use of the RISC System/6000 architecture features; in particular, the virtual memory management, the device handlers and the compiler code generators. This paper reviews the virtual memory management subsystem and the radical alterations made to other parts of the kernel to support it. These concepts may be considered as the key to understanding the AIX Version 3 kernel on the RISC System/6000 computers.

The memory management hardware is described in detail, the implications of the hardware are then examined and the resulting software features are shown. Finally, the benefits for the user are outlined.

1. Introduction

The AIX Version 3 kernel is a new UNIX kernel which has been designed and built by IBM over the last 3 years. The major design objectives were [Cera]:

- to provide an operating system base for the 1990s, based upon UNIX, but using the latest software technology
- to support and take advantage of IBM's new RISC System/6000 architecture and hardware
- to ensure that the system design and structure allows portability to other architectures.

Although these objectives are reflected throughout the entire AIX kernel, two sections in particular were extensively modified to support the second objective; the memory management subsystem and the device handlers. The device handlers are specific to their particular devices and are not of such general interest, the memory management, however, has ramifications throughout the kernel and will be examined in detail.

The RISC System/6000 computer family has an unusual memory management architecture with:

- *An extremely large segmented virtual address space.*
  This very large, easily remapped, address space has enabled many more objects to be memory-mapped than is usual in UNIX kernels.
- *Several different type of segments.*
  Apart from the normal "working space" type of segment there is also support for persistent segments such as file data, for journalling of segments which gives database-like facilities and for client segments from remote machines.

These features will be examined in some detail after an overview of the memory management hardware and software architectures.

2. Memory Management Architecture

2.1. Hardware Features

The RISC System/6000 memory management unit [Groa] takes a 32 bit program address and expands it through 16 segment registers to a 52 bit virtual address which, after virtual address translation, becomes a 32 bit physical address. This is shown in figure 1.

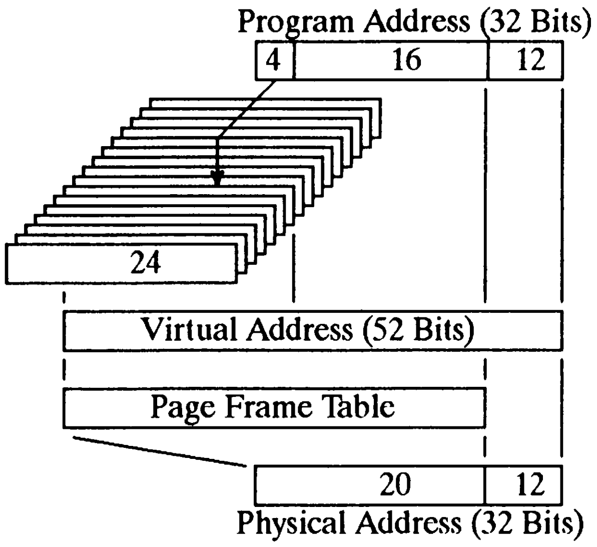

Program Address (32 Bits)
4
16
12
24
Virtual Address (52 Bits)
Page Frame Table
20
12
Physical Address (32 Bits)

Figure 1: *Memory Management Hardware Architecture*

The top four bits of the program address are used to reference one of 16 segment registers. If the most significant bit of the segment register is a 1 the reference is to I/O space and no further address translation is carried out. If, however, the most significant bit is 0 the contents of the segment register are concatenated with the remaining 28 bits of the program address to produce a 52 bit virtual address split into more than 8 million segments of 256 MByte each.

The Page Frame Table provides the page address. It is arranged as a conventional hash table with indexes into the main page frame table. As a result page frame table entries are only required for real pages with only one entry being required per real page. In order to accelerate the translation process the most recent translations are retained in a translation lookaside buffer.

There are several flags in the page frame table which are used and set by by the hardware. Firstly there are dirty and referenced flags which are used in much the same way as in most other memory management systems. Additionally there are lock bits whereby the hardware prevents processes from writing to a locked line (128 bytes) in a segment with the appropriate attributes. These locks may be used for serialization and to provide database-like memory in journalled segments.

In general, memory accesses with address translation switched on are as fast as memory accesses without address translation. All parts of the kernel are, therefore, memory-mapped (including interrupt handler vectors) apart from a few state variables defining the address translation status.

2.2. Segment Types

The RISC System/6000 system defines several different types of segment each of which is used for different purposes:

- *Working storage.*
  This corresponds to traditional virtual storage. It is used for transient data such as a process's stack. Paging is done to the paging area in the normal manner.
- *Persistent storage.*
  This is used for data which must survive a reboot, e.g. open files and filesystem metadata. In this case paging is done directly to and from the file system.
  There is also another type of persistent storage segment with database-like attributes. This is a journalled (persistent storage) segment. In this case alterations to the segment are made atomically using the hardware-supported line lock feature mentioned above and the changes are logged in a log file to ensure consistency.

- *Client storage.*
  This is used for data from a remote machine; a typical example is a file opened using the Network File System. Paging is done across the network and is handled by a special strategy routine.
- *Log storage.*
  This is similar to persistent storage and is used to support journalled segments. In this case, however, the memory manager handles disk I/O directly in order to ensure the database-like semantics.

2.3. Data Structures

All the data structures used by the memory management hardware and software (and throughout the AIX Version 3 kernel) can be extended dynamically. At system startup time a segment is allocated to the virtual memory manager for the majority of its data structures. This is sufficiently large to support system sizes up to 2**49 bytes. In addition a separate segment is allocated to the page frame table with further segments being allocated as and when required.

The major data structures used by the memory management hardware and the software are:

- *Segment Information Table (SIT)*
  The SIT is a table of segment control blocks used by the software to facilitate accessing the segment control blocks.
- *Segment Control Block (SCB)*
  This is the main data structure for describing the individual segments. It contains fields defining the type of segment, how it is to be paged and where it is to be paged to, and various other pieces of information, such as segment size limits.
- *Page Table Area (PTA)*
  This is a table of external page tables.
- *External Page Table (XPT)*

  The XPT is the major software table used by the memory management subsystem. It defines where pages should be paged to. The XPT varies depending on the type of segment.

  Working storage segments XPT entries have a disk address to which the page should be paged, several page protection bits, a "to be zeroed" or "finished with" bit and one bit to define whether the page is shared or not.

  Persistent storage and log segments do not have an XPT. For these segment types the relative offset into the segment is used to determine where the page should be paged to in secondary storage. For instance, the second page of a file must be paged to the second block of the file on the disk. As will be described later, copies of the file system metadata are kept in journalled segments and these in-store copies are used by the pager to expediate this translation.

  Client segments also have no XPT. In this case the page is sent to a strategy routine for dispatch to the client machine. The address of the strategy routine is defined in the SCB.
- *Hash Anchor Table (HAT)*
  The HAT maps a hashed virtual address to an index into the page frame table. The HAT grows as required in an attemt to ensure as near optimal indexing as possible.
- *Page Frame Table (PFT)*
  The page frame table contains the mapping between a virtual address and the associated physical address. Additional information contained in the table includes the changed (dirty), referenced and locked bits and several page protection flags.
- *Paging Device Table (PDT)*
  The PDT contains pointers to those device drivers which are required to read or write pages for each device used by the memory management subsystem.

2.4. Software Architecture

The memory management software is split into five distinct subsystems:

- the segment manager — is responsible for the allocation and modification of complete segments (the SIT and SCBs)

- the virtual page manager — manages the mapping between virtual pages and pages in secondary store (the XPT)
- the page frame manager — ensures that the page frame table used for mapping between virtual and physical addresses is kept up to date
- the page fault handler — is invoked when a page fault occurs. It reads the faulted page in from secondary storage using the XPT
- the persistent storage manager — provides database-like facilities for the filesystem.

More details of these subsystems may be found in [Techa] which also describes the data structures and algorithms used to far greater depth than is possible here.

Because of its unique nature the persistent storage manager will be described in rather more detail.

2.5. Persistent Storage Manager

The persistent storage manager (PSM) is the part of the memory management system responsible for journalled persistent storage segments. Through the use of database-like techniques and the line lock feature of the memory management hardware the PSM ensures that these segments are maintained in a consistent state .

When a process wishes to alter a journalled segment, it first requests a transaction identifier from the PSM. The process then requests that the PSM lock manager lock those parts of the journalled segment it wishes to alter. As the process makes the changes, the PSM log manager intercepts the changes writing them into a separate log segment. The process then relinquishes the transaction identifier and the PSM commit manager writes the log directly from its own buffers to secondary storage. The log now has a complete list of the changes. Required by the process and, if the system crashes, the changes are replayed from the log.

The PSM now writes the logged changes to the journalled segment, after which the locks are released. The changes are now written to disk following which a logsync record is written to the log, signifying that the journalled segment is now consistent with the log record.

This process is outlined in figure 2.

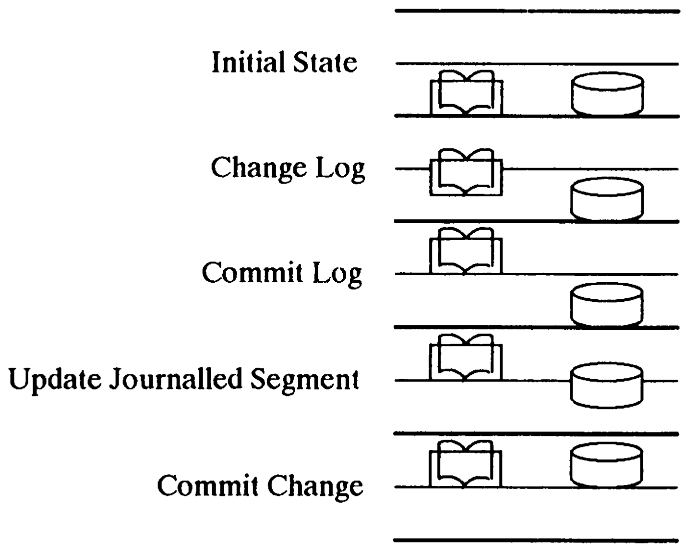


Figure 2: *Stages in Changing a Journalled Segment*

The PSM is currently totally inaccessible from the user level. There are no system calls to reference or modify its behaviour.

2.6. Paging Deadlock Avoidance

The memory manager detects if available paging space is running low and takes appropriate action. At the first level it issues a warning message to the console suggesting that the paging space be increased with the *defineps()* system call. At the next stage it kills the lowest priority task(s) which are using paging space and, at the final stage, if deadlock becomes inevitable, the memory manager issues a kernel panic at a threshold just large enough to complete panic processing.

2.7. Segment Register Allocation

Six of each process's sixteen segment registers are allocated automatically when a process is started:

- process text (read-only)
- process data (read-write)
- primary kernel segment (hidden)
- secondary kernel segment (hidden)
- I/O devices (hidden)
- shared libraries (read-only)

The remaining ten segment registers are allocated by the kernel as requested by the process through various system calls. The user process is generally not aware that it is accessing additional segments (for instance, a file read is semantically equivalant to previous read() system calls but now uses a segment) and the kernel handles any segment swapping required in order to access all of the objects in the process's address space.

3. Consequences

From a software viewpoint, the major characteristics of the memory management architecture are:

- exceptionally large virtual address space
- extremely high number of unique segments
- economic support of sparse address maps
- rapid access to segments outside of the program's address space by changing the segment register
- database-like memory accessible from the kernel
- support for networked pages

The more than 8 million segments each containing 256 MBytes far exceed the number required to support all of the usual memory-mapped objects for any reasonable number of processes. The spare segments are therefore used to memory map additional objects, such as all open files and the file system metadata.

3.1. File Mapping

File mapping is one of the most interesting uses to which the spare segments are put. When a file is opened a new persistent storage segment is created and allocated to the file. A file read operation then becomes a memory copy from the file's segment to the process's data segment. Similarly, a file write operation becomes a memory copy from the process's data segment to the file's segment.

The advantages of this approach over the traditional buffer cache are:

- the performance bottlenecks associated with the buffer cache no longer exist
- file system data is only written once in real memory (from the process segment to the file segment) rather than from process space into kernel space into the buffer cache
- more efficient use of real memory as the buffer cache is no longer required and there are only up to two copies of the data in memory instead of three
- the approach is more general and flexible.

3.2. Networked File Mapping

The client data segments are used for the Network File System (NFS) implementation in AIX Version 3 and could, potentially, be used for other networked file systems. As with local files, when a remote file is opened a segment is created and allocated to it. In this case, however, a client segment type is used instead of a persistent storage segment. The segment is then paged across the network rather than read in its entirity, as was the case in AIX Version 2 and some other versions of UNIX.

The advantages of this approach are:

- network traffic is lower as only those parts of the file required on the local machine are passed across the network

- startup time is faster as the entire file need not be transferred before processing resumes
- memory usage is improved as only the required parts of the file are copied onto the local machine.

3.3. File System Metadata Mapping

As stated above, file system metadata is also mapped into virtual memory. The following segments are created for each filesystem:

- *superblock*
  The superblock is held in a journalled segment.
- *disk block allocation map*
  This bit map indicates which blocks on the disk are in use. It is not journalled and may need to be reconstructed after a machine is restarted.
- *inode table*
  The inode table is held in a journalled segment.
- *inode allocation map*
  This map is similar to the disk block allocation map, but is used for the inode table. It may also need to be recreated after a system restart.
- *inode extension table*
  Some extensions to inodes are required to support the security features of the AIX Version 3 kernel, for example the access control lists. These are kept in a journalled segment.
- *indirect block map*
  The indirect blocks in the file system are kept in this segment. There is no parallel to this segment on the disk and therefore is always recreated at system startup time.

The first consequence of this structure is that, barring catastrophic media failure, the filesystem (but not the contents) is guaranteed to be consistent at all times. A power failure or similar problem will never result in a broken file system. The second, less obvious, benefit is that various kernel subsystems, including the memory management system, can access this information very much more rapidly than in conventional UNIX designs.

4. Summary

The RISC System/6000 family has an unusual memory management system with:

- an exceptionally large virtual address space
- an extremely high number of unique segments
- economic support of sparse address maps
- rapid access to segments outside of the program's address space by changing the segment register
- database-like memory accessible from the kernel
- support for networked pages.

These features are used in the AIX Version 3 kernel to support memory-mapped files with considerably enhanced performance, remote mapped files which reduce network traffic and startup time on remote operations and finally a database-like filesystem guaranteed to remain consistent under normal operating conditions.

References

[Cera] Dan Cerutti, "AIX Version 3 Software Overview," in *RISC System/6000 Technology Book*, IBM Corporation, Forthcoming.

[Techa] *AIX Version 3 Technical Reference Manual*, IBM Corporation, Forthcoming.

[Groa] Randy D. Groves and Richard Oehler, "RISC System/6000 Processor Architecture," in *RISC System/6000 Technology Book*, IBM Corporation, Forthcoming.

Cooperation Support for UNIX-Based Industrial Applications

Holger Herzog

Burkhard Stork

Siemens AG,
ZFE F2 SOF 4
Otto-Hahn-Ring 6,
D-8000 Munich 83,
FRG

sto@ztivax.uucp

1. Introduction

Decentralized factory automation splits complex central control into smaller components that are easier to survey. Access to information is quicker if the information is available on the site. Decentralized factory control demands communication systems or distributed systems that support cooperation between distributed application programs. Therefore distributed process automation systems tend to encourage a great deal of interdependence among machines.

Technical approaches vary. Already existing factory automation technologies span from basic industries to technical and administrative offices. On one hand, there are pure communication systems in compliance with the ISO/OSI reference model. On the other hand, you can find distributed operating systems representing a complete cooperation capability. Between these two extremes there are many different stages, e.g. the well-known systems for network operating facilities. This variety of solutions is due to application requirements, however, there is an increasing demand to substitute communication by cooperation. This article illustrates this trend by showing research results of our group. Our main research efforts are distributed cooperating systems. All prototypes developed are UNIX-based.

The indicated trend is part of the standardization process. These developments are rounding off the ISO/OSI track. Examples are the standardization for network management and other upper layer services. The endpoint of the development from interconnection to interworking, or in other words, from communication to cooperation, is marked by the ISO and ECMA standardization process for Open Distributed Processing (ODP).

This article is structured into four steps that show the indicated trend and that describe different kinds of interdependence in factory automation systems:

- communication in Automation Systems
- centralization in Communication Systems
- isolated Transparent Functions
- cooperating Systems.

For each step we give a short description on how this step fits into the trend, and we follow up with a case study. The case studies describe experiences with the reusability in programing and architectural issues. The Requirements of factory automation heavily affect the design of the case studies.

2. Reusability in Programming

Reusability of software is not a well defined term. While [Big84a] mentioned two kinds of reusability principles: Composition and Generation, [Jon84a] distinguished among five types of reusability:

- reusable architecture
- reusabale design
- reusable data
- reusable programs and common systems
- reusable modules.

The last two types of reusability could be associated with composition. The first three types stretch the term reusability to:

(1) transferable structures (reusable architecure)

(2) software design aiming at reuse (reusable design)

(3) the requirement of a standardized data exchange format to integrate the reusable components (reusable data).

The components can be standard routines gathered in a library or complete tools. Therefore, the term reusable data includes data exchange by a data base and the pipe mechanism of the UNIX operating system. In the following case studies we use the classification of [Jon84a] to describe our experiences with reusability. [Her89b] discusses the effects of the international standardization on reusability.

3. Functional Structure of Automation Systems

Figure 1 shows an automatization pyramid that is widely used to illustrate computer structure within CIM and process control systems [Dil86a]. The components are:

- management and production planning
- production control and coordination
- process control
- local control and
- sensors, drives, and regulating units.

At the lowest level of a CIM system, sensors, measuring instruments, and drives pass information to the almost stand-alone processes, which monitor and control the intelligent tools and machines, transportation devices and robots. Today, slow and simple field buses are used as connections. This layer demands realtime properties (reaction times in the order of 10-100ms). Overall processing times are decreased considerably by distributing the tasks over several processing units.

At a higher level, several simple systems or machines form larger functional units. Machine tools, robots, and assembly lines form a production cell; different operating and monitoring devices form a control unit. Realtime is still a requirement at this level. Several suppliers offer systems for these tasks with different parts connected via coaxial cables. These systems do not allow horizontal communication within the entire production process. The units are still separated and work independently. Here, standard local networks would improve productivity considerably.

The controlling layer coordinates the production units and provides central file services. Realtime requirements are no longer significant. The ability to handle great quantities of data is more important: files containing planning or quality controlling data are usually about 10 MByte. They provide the connection between planning and management on the one hand, and production lines on the other. The use of base bands still dominates this area, but in the long run, one will have to change to more efficient broad bands. Units on this central controlling layer monitor and supervise all activities in a network.

A system that supports planning has to provide facilities for management tasks as well as for technical offices. Especially technical office automation has moved towards new support technologies. The available tools for construction of subassembly parts, simulation of systems (like robot control), programing, documentation, and test are not really integrated: CAD-applications still have the problem that different data storage systems can only communicate via special transformers. Two main aspects are:

(1) lack of universal data storage systems,

(2) lack of normed and reliable back-up mechanisms (like version management for software development, data bank systems for management and engineering, documentation systems for offices and development).

This model neglects business functions and tasks, such as staff and personnel management, estimating costs, and book-keeping, and more production oriented functions, such as aquirement and sales. Although there are no clear-cut boundaries between these tasks, only a basis consisting of a homogeneously accessible and distributed system with easily adaptable interfaces will allow the realisation of a Computer Aided Industry (CAI). Such an architecture must integrate company-made automation systems, support hardware, like data base machines [Hil89a], and software solutions of other suppliers. An open system must support efficient solutions for special tasks: the job of integrating different systems must not result in an excessive overhead for local solutions.

4. Communication in Automation Systems

Many systems structured this way are based on the ISO/OSI reference model or are derived from functional standards like MAP. OSI-compatability, however, does not necessarily provide features for cooperation of different components. Protocol standards are usually overloaded with options and a variety of possibilties to combine several parameters, which are needed to ensure joint operation of two systems via communication. Moreover, integrating existing applications that do not support OSI-functionality (like those only accessible on a mainframe) require implementing specific services for communication.

Generally speaking, the possibility of communication within a distributed system leads towards decentralization of tasks and thus demands data exchange. Together with the heterogeneous hardware used in automation technology this requires a standardized data interchange format and a standardized data representation format for display devices. As far as the technical office is concerned, IGES is widely used to transfer data between CAD systems. ODA/ODIF has been developed for the administrative office. Figure 2 shows possible data exchange relations enabled through our ODIF exchange kernel.

Protocols in layer 7 of the ISO/OSI model cannot be formed as those of the lower layers, because they may have to support very specific requirements for a class of applications. On one hand, there are many stable solutions available (the Telematic services of the public network). On the other hand, demands increase for tools and applications that support protocols in the field of office communication and automation technology. The Office Document Interchange Format is one component needed in both industrial spheres.

Case Study: ODIF (Office Document Interchange Format)

The exchange of documents between applications or users, respectively, in a heterogeneous distributed system requires a standard format of data interchange. Using ODIF (Office Document Interchange Format) on the basis of ODA (Office Document Architecture) every application can read and edit arbitrary documents (text and graphic) in its execution environment. This is realized through an ODIF-format-compiler. A text formater for the layout structure (header, footer, page number, etc.) is under development.

ODIF-documents can be interchanged. Text is information for human usage. It can consist of the following elements: symbols, line graphics, and raster graphics. Text elements form the contents of a document. An ODIF-document also contains information on internal organisation, such as logical structure and layout of a document.

Each application, e.g. editor or printer, that works with a document will use this document in a specific form which is customized for this application. Whenever one application wants to pass a document to another application it has to transform this document into standard ODIF-format. The receiving application then has to change the standard ODIF-format into the application specific format.

Each tranformation (dependent on various devices or text systems involved) needs a specific converter. To avoid this time-consuming process, we have developed a converter core system, that can be adopted to specific attributes of the documentation application. The converter is syntactically described by an attribute grammar (a well-known technique in compiler construction) that generates the actual conversion program. This allows fast production of new converter types, or update to new ISO standards features (which occur frequently). The generating process is implemented by a meta compiler, that produces data stuctures and source code in C from the ODA document description: the central conversion procedures can be reused. Following the classification of chapter 2 the meta compiler is a reusable program while the ODIF is one example for reusable data.

5. Centralization in Communication Systems

The variety of systems in use, hardware as well as software, is due to the broad spectrum of applications. It is therefore necessary to support their use by provision of a standardized data management, and by concentration of resources, and hence by realization of central services. Suitable candidates for such services are functions that are frequently used by all users/applications in the network, e.g. a database system, a mailing procedure, or the general accessibility of a mainframe computer. Another well-known example is the file service to represent network-independent files as a homogeneous hierarchical tree.

Development of services is trend setting in the architecture of application systems. It allows for concentration of commonly used resources. A client-server-solution that embeds centrally required functionality reduces the development effort significantly. Application development can be mounted directly on top of a software structure (which means efficient modularization), and systems functionality (communication mechanisms, synchronization procedures, etc.) is generally provided.

Such systems tend to be extremely complex, and neither software nor hardware elements can be configured manually. Network Management will be a task of its own. It has to be attached to central functions within a complex environment, such as the system administration service. Network configurations in the area of industrial manufactoring show a very high degree of heterogeneity of the connected machinery (figure 1). Different applications have different requirements. A network management system has to provide the following functionalities:

- design, configuration, and modification of a network
- dynamic control of the network
- fault analysis within the network.

After a thorough analysis of the various requirements, we worked out a generally valid concept for network management. This task is put into practice by prototypes of a system administration facility tailored towards networks used in mining, and a configuration system for automation technology [Leh88a]. The user interface of this configuration system is based upon a sophisticated and flexible user interface toolkit that was developed in our group and is compatible with the X-Window standard [Sto87a].

Case Study: CONSENS

CONSENS is a configuration-shell to generate expert systems for configuring Local Area Networks. The goal is to create expert systems for each network topology through integration of specific configuration knowledge into the shell. The task of these specific expert systems is to design, configure, and manage Local Area Networks. The basic objects within the expert systems are the physical objects of a network-topology. Networks are built by synthesis of the basic objects to more complex objects, join these sub-parts to sub-networks and then join these sub-networks (layered view). The expert system asserts the consistency and completeness of each configuration.

The configuration-shell CONSENS is a knowledge-based system with a strict separation of the configuration-knowledge and the inference machine. The user has to build specific knowledge bases to generate an expert system for a new network-topology. Therefore, each expert system derived from the shell is knowledge-based. Central components can be reused as tools within different contexts to support the design, configuration, or development process by "simply" adopting the knowledge-based approach, i.e. reusing suitable parts of the knowledge base and general inference mechanisms. Therefore the toolkit is a set of reusable programs.

The interaction with CONSENS is realized via a graphics editor to manipulate network representations. With icons (they represent any physical unit) the user is able to depict any network configuration through different levels of abstraction. A toolkit is used to gain a uniform view for all applications within the system. This is reflected strongly in the architecture of the entire software. Used as a concept, by all application developers, it has the tremendeous advantage of easily gaining conformance and technical integration at the user level.

6. Isolated Transparent Functions

A central service can be used from all other components in the system. For the user this looks like a realisation of a single user system with an additional new service. The same is true for the realisation of isolated transparent functions with the difference that the new functionality is hidden. Isolated transparent functions are in this sense a further development of the central services and the second step in the direction of cooperation in factory automation systems.

Services are software technological structures. They are autonomous units in most respects and can be called up using simple interfaces. They will usually handle their tasks without interference or manipulation from the calling application. This allows the use of services in heterogeneous environments. An example for a service is the mail service which was implemented on a Siemens PC-16/20 containing an Intel processor. It can be used from a Siemens SICOMP WS20 environment, which uses either UNIX or MS-DOS as an operating system.

If, however, applications need to influence the execution of the software, this crude distributed concept of services is no longer sufficient. It is necessary to supply basic mechanisms for cooperation on the different system layers, such that there are no problems with the communication interchange of the different system components. These parts of the program can only function correctly, if there is a compulsory standardization of the processing of this information. This requires the modification of existing applications. The advantage, however, lies in the newly created possibilites to combine existing or newly developed applications to a system with large functionality. These solutions were developed in our division. They make up the central components of a completely distributed system. They build the foundation on which the implemention of finer distribution concepts becomes meaningful.

Case Study: Mechanisms for Fault Tolerance

Problems connected with reliability and availability of a distributed system were analyzed in the project "security compound" using the Siemens internal SINEC H1 network. This resulted in a prototype of a transaction oriented and fault tolerant security mechanism, which can be integrated into any kind of server. This mechanism is user transparent. It provides consistent data and structured re-integration if a component of a network breaks down.

Local networks usually consist of different kinds of computers performing different tasks. Economic and performance oriented demands require reliability and availability in these heterogeneous networks. The network, considered as an entity, has to be able to react to malfunctioning of its components. This should be transparent to the user.

A major problem occurs in central services: their malfunctioning may have an effect on the entire system. Consider the central file server breaking down, it does not matter whether this is a result of a software or hardware failure. This can be avoided by using a redundant file storage system and redundant server systems. The aim is to provide standard components (in the following called SC-layer — security compound layer) which can be built into a service. Using the SC-layer the service can be extended to a redundant and fault tolerant service. Again, this redundancy should be transparent to the user.

The security compound is based on the following idea: any job started by a user (client) is processed by a server, which hands the job to a shadow server, which in turn executes the same job in parallel. The SC-layers task is:

- to guarantee consistent data
- to enable the user to work with the surviving devices if part of the system goes down
- to re-integrate a damaged component, after the fault has been dealt with.

Economic as well as performance reasons forced the implementation of reliability and availability mechanisms in heterogenous networks. The entire system should be able to detect failures and to make correction of the failure transparent to the user. One aspect in the field of fault tolerant systems are failures in central services. Redundancy in data storage and redundant server systems are two possibilities to increase reliability and availibility. The implemented mechanism is based on the idea of intercepting a user query and executing this query twice: one time on the server and one time on a shadow server. The layered architecture of the fault tolerance mechanism is typical for software components based on layered communication mechanism. The fault tolerance service can be added to every client/server system and therefore is an example for a reusable design and reusable programs.

7. Cooperating Systems

New Requirements

Todays CAD/CAM-systems resemble isolated islands. The integration of these system parts in CIM is expensive. The huge costs of complete CIM-systems allow reaching the goal CIM only step by step in an evolutionary process. The necessary features of systems on which CIM can be built are:

- evolution of systems
- integration of existing and future applications
- re-configurability
- performance
- fault tolerance on different layers of the complete system.

In the future, there will be the integration of CIM and the administrative office, which is sometimes called computer integrated business (CIB) or computer assisted industry (CAI). Therefore the future requirements will be more complex than they are for CIM.

Transparency

The new functionality of distributed systems, in contrast to central systems, can be expressed by the user hidden functionality. This is described by the various transparency forms, which are named below. Transparency names, on one hand, the realisation of system properties, and on the other hand, the abstraction of realisation details. Following transparency forms can be distinguished [Her89a]:

- access transparency, hiding the use of explicit communication
- location transparency, hiding the topology in a system
- concurrency transparency, hiding the effect of parallel execution potential
- replication transparency, hiding the replication of state in a system
- failure transparency, hiding the occurence of errors in system components and communications
- migration transparency, hiding the heterogenity of system components to enable the migration of system functions between components and communication
- performance transparency, hiding the performance penalties and the advantages of distribution
- scaling transparency, hiding the effect of changes in system size.

Enlarging the system functionality through implementation of transparency aspects is necessary to fulfil the above mentioned requirements. Scaling transparency, for example, is the basis for the evolution of systems and replication transparency is one functionality which is needed for performance and fault tolerance requirements.

In the introduction we mention a trend from the interconnection to the interworking in distributed systems. In light of this trend the implementation of transparency can be seen equivalent to the implementation of interworking.

Two types of distributed systems are distinguished:

- full transparent systems like distributed operating systems [Tan85a], and
- systems with selective transparency which are systems that offer only few of the transparency forms (see figure 5).

In the future, factory automation of both types will be used. At the cell computer level some kind of distributed operating system might be suitable, but at the other levels selective transparency is adequate. For example, location transparency between the process level and the company computer supports systems with a location transparent transport of statistical information. In this application, full transparency makes no sense.

Case Studies: COMANDOS and ISA

Mainly, we develop distributed systems for factory automation and process control systems. Therefore, we are involved in the development of systems that belong to both mentioned types of distributed systems.

The aim of the ESPRIT project COMANDOS (Construction and Management of Distributed Operational Systems) [Hor87a] is to build an infrastructure for distributed applications which manipulate long-lived data. COMANDOS integrates concepts out of the programming language domain, the data base domain, and the distributed operating system domain. Additional features are the enlargement of the UNIX kernel by the COMANDOS infrastructure, the integration of LAN and WAN within the operating system, and the implementation of administrative facilities. COMANDOS offers full transparency so that the user does not see the distributed system.

The implementation of full transparency increase the complexitiy of a system and might decrease the performance. For applications which take special emphasis on the performance — like real time applications — the construction of distributed systems with minimal overhead is needed [Kop89a]. Furthermore, application designers may want to use the advantage of distribution, e.g. to implement explicit parallel execution. To fulfil this requirements the ESPRIT-project ISA (Integrated Systems Architecture) combines a building block approach, one block for each transparency, with the concept of selective transparency (figure 5). The general goal of ISA, a successor of the ANSA project [Her89a], is the development of concepts for interworking in divers application domains and for multi-vendor-environments. These concepts should be used to consolidate and support the development of international standards especially the standardization process for Open Distributed Processing.

ISA and COMANDOS offer an object oriented infrastructure to the application programmer. Both projects want to integrate the quasi standard UNIX. Therefore, the process oriented view of UNIX is combined with object oriented view by adding a library to each UNIX process. COMANDOS and ISA specific libraries include routines for message passing, and for the import and export of local process specific functions, and so on.

In ISA two kinds of sofware re-use can be distinguished: re-use of software by an application implementor and re-use of the ISA architecture itself. While the first case of re-use is given by the common design principles developed in ISA (reusable design), which are based on the object oriented approach, the second case is given by the well defined kernel interface and the application platforms build from transparency building blocks. The kernel interface (nucleus) offers the capability of reusable programs. The concept of transparency building blocks are on one hand a reusable architecture concept and on the other hand an example of reusable programs.

8. Conclusion

We would like to summarize this article in three aspects:

- a general system development aspect
- an architectural aspect and
- a tool aspect.

The indicated trend from communication to cooperation does not mean that all future automation systems will be cooperating systems. Each mentioned step will be found. We think that the architecture of distributed systems of the future will be based on a building block approach. The system designer should be able to realize a pure communication system as well as a full transparent distributed operating system with the same methods.

It has been common that the system designers took a multi-level approach to structuring the architecture of operating and communication systems. Examples are the structuring of operating systems by virtual machines [Tan76a] and the ISO OSI reference model. The idea is to hide special features of the level below, yielding an abstraction of that level. The development goes bottom up from the hardware to the application. Today the object oriented design is very popular. The object oriented method takes a view that looks from the application or from the system software on the hardware. This is a top down view. New developments like ISA show that both design principles belong together. A layered architecture is chosen to hide special features and to offer a basic functionality. In ISA this approach is taken for the communication system (one possibility is TCP/IP) and for the kernel interface. In COMANDOS only the communication system has a layered achitecture. The other parts of the architecture take an object oriented approach. For some application areas a merge of both approaches is developed [Cor89a].

The tool aspect can be devided in three basic strategies that are necessary for creating a common platform for the program development of distributed systems in a heterogenous environment (different networks, devices and applications):

- integration of knowledge based concepts
- adaptability for easy system evolution by generator techniques

- modelling of system behaviour.

Most technical processes are planned, monitored, and controlled without using formal models or procedures. Usually, more or less subjective strategies are developed. An operator of a process control system may draw some conclusions from a multitude of information and modify the system according to these conclusions. By using knowledge based systems this application specific knowledege could be made available to others. The applied know-how will be more objective and thus more reliable. The availability of a problem solving capacity, which is problem specific and knowledge based, will faciliate not only dealing with complex systems but also with the evolution and development of these systems. The project CONSENS goes one step in this research direction.

Distributed systems are determined by dynamic changes to the environment. In flexible production units optimal performance of a whole production line depends on quick reaction to changes. These may be as different as changes to the crude technical basis (the adding of new machines or substituting of new machines for old ones) or suddenly occurring breakdowns (breakdowns of parts of machines or the entire machine). It has to be possible to substitute single components in a controllable manner. No software specialists, like the person in charge of a production cell, must be able to deal with these changes without destroying dependencies of the entire system. A distributed system has to supply software which recognizes these dependencies and which will change the system from one consistent state into another. If done without software support, this is not only very difficult but also very expensive. Generator techniques like those taken by our ODA/ODIF project should be used to solve this problems.

Dynamic properties are not just an aspect of system evolution but also of actual system behaviour. A broad overview can be provided by the human ability to view and abstract complex things. Graphics are used for abstraction, they allow the quick understanding of complex issues. Used to support software technology, one can describe several interconnected bearings: as it is done in the airbus industry, which provides pictures and graphics to describe actual flight conditions of the airplane. One step further goes the simulation of the distributed system behaviour which is a method to manage the complexity of future automation systems and to recognize design failures before any system part will be realized.

References

[Kop89a] H. Kopetz et al, "Distributed Fault-Tolerant Realt-Time Systems, The MARS Approach," *IEEE Micro*, February 1989.

[Cor89a] R. Cordes et al, "Layered Object-oriented Techniques Supporting Hypermedia and Multimedia Applications," *Proc. WOODMAN 89*, Rennes, France, 1989.

[Big84a] T. J. Biggerstaff and A. J. Perlis, *Foreword, IEE Transactions on Software Engineering*, SE-10, September 1984.

[Dil86a] R. Dillmann, "Computing Aids to Plan and Control Manufactoring," *Computer-Aided Design and Manufactoring*, pp. Springer-Verlag, Berlin, 1986.

[Her89a] A. J. Herbert, *ANSA Reference Manual*, Cambridge, UK, 1989.

[Her89b] H. Herzog and B. Stork, "Effects of the Standardization Process on the Reusability of Software," *Proc. ITG/GI/GMA Conf. on Software Technique in Automation and Communication*, 1989. German

[Hil89a] F. Hildebrandt and H. Herzog, "COMMA — A Hybrid-Coupled Multiprocessor System Supporting Parallel Activities in Database Systems," *Proc. Communication in Distributed Systems, Stuttgart, IFB 205*, pp. Springer-Verlag, Berlin, 1989.

[Hor87a] C. Horn and S. Krakowiak, "Object Oriented Architecture for Distributed Office Systems," *Proc. ESPRIT Technical Conference, Bruxelle*, July 1987.

[Jon84a] T. C. Jones, "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering*, vol. SE-10, no. No. 5, September 1984.

[Leh88a] D. Lehmann and V. Teuber, *An Expertsystem Shell for the Construction of Networks &J Proc. WiMPEL'88: Knowledgebased Methodes for Production, Engineering and Transport* , pp. Teubner-Verlag, 1988. German

[Sto87a] B. Stork and R. Zellner, "Architecture of user interfaces on graphical Workstations under UNIX," *Proc. Jahrestagung German UNIX User Group (GUUG)*, 1987. German

[Tan76a] A. S. Tanenbaum, "Structured Computer Organisation," *Prentice-Hall*, Englewood Cliffs, 1976.

[Tan85a] A. S. Tanenbaum and R. van Renesse, "Distributed Operating Systems," *ACM Computing Surveys*, vol. 17, no. 4, December 1985.

RApp: A Generic Routing Application in C++

Sanjiv Gossain

Bruce Anderson

Department of Electronic Systems Engineering
University of Essex,
Colchester, UK

goss@essex.ac.uk
bruce@essex.ac.uk

ABSTRACT

A new approach to application development within domains, based upon software reuse, is presented — the generic application. A generic routing application, RApp, is used as a vehicle to outline the features of the generic application. RApp's constituent classes are described and the method by which applications are created is outlined. Using object-oriented techniques, we have provided flexibility, maintainability and reusability, in the form of a class hierarchy achieving up to 76% reuse of code. We have also reused designs from previous implementations. The significance of these and other results are presented.

1. Introduction

In this paper we present an approach to software reusability that is an extension of the notion of software reuse through components. The components are specific to their application domain, and are part of an inheritance hierarchy that forms the basis for further development of applications within that domain, by elegant and natural extension or modification, of existing components.

The concept of a generic application for a domain is based on existing software framework packages [Sch86a, Wei88a] which implement a standard user interface. Unlike these packages however, there exists a high degree of domain knowledge represented within the class hierarchy. These classes, along with generic classes, common to all applications within the domain, provide all the essential building blocks needed to create domain solutions rapidly. This is only made possible by the features that object-oriented languages can offer in terms of data abstraction, encapsulation, and inheritance; thus allowing developers to augment or modify existing classes to suit the needs of their particular application.

We describe where, in the spectrum of research into reusability, the generic application framework lies, and identify how the generic application differs, in the issues it aims to solve, from interface frameworks such as MacApp, and ET++.

We have developed a generic application for the VLSI Routing domain, RApp [Gos89b], which captures much of the functionality and commonality that exists between various routing algorithms. RApp's constituent classes are described in section 3, and we also outline how one would construct an application with the framework.

The various forms of reuse experienced when using RApp are highlighted, along with the possible impact such a framework may have on domain-specific application development.

Part of section 3 of this paper was presented at an IEE Colloquium on Applications Of Object-Oriented Programming in London, November 1989.

2. Generic Applications

With the wide variety of research currently being done in the investigation of reuse through software components, there are naturally many differing approaches being taken. Biggerstaff and Perlis [Big84a] suggest there are two main approaches:

Reusable building blocks

Reusable patterns

The approach described herein falls within the first category, that of reuse of software by composition. It is our belief however that it also can be seen as a pattern approach to reuse. We have been able to reuse existing design structures, existing design diagrams and existing design by-products in subsequent designs, which can be considered as a form of patterns. Figure 1 shows where in the spectrum of reusable technologies, generic application frameworks can be found. We see that the generic application incorporates features from both application component libraries, and organisational approaches to software reuse.

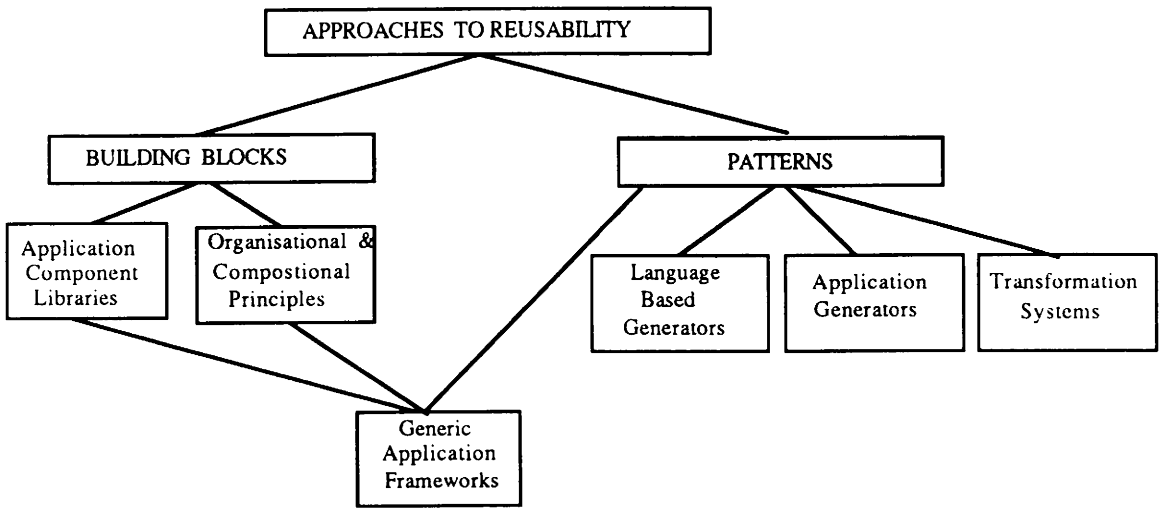

APPROACHES TO REUSABILITY
BUILDING BLOCKS
PATTERNS
Application Component Libraries
Organisational & Compostional Principles
Language Based Generators
Application Generators
Transformation Systems
Generic Application Frameworks

Figure 1: *Position of Generic Applications In The Spectrum Of Reusability Adapted from figure in* [Big84a]

Design of applications through use of classes has two aspects — extension of the hierarchy through subclassing; and creating objects of classes that are appropriate to the application in hand. A class hierarchy can thus be viewed as an improved program library, particularly as the internal representation of each class (component) need not be known to the user [And88a].

A generic application however, is an extension of this approach, in that it provides all the generic features needed for any application in that domain, with only the specific functionality for an individual application missing†. The system-builder provides this missing functionality by extending and augmenting existing classes where necessary. Figure 2 illustrates how the implementation of an application uses existing, refined and new code.

For our classes to be truly reusable, we must ensure that they are truly representative of the domain and that they model the behaviour accurately and without ambiguity. Each abstraction must be meaningful to domain experts in name and in behaviour. The initial, domain related stages of the design, are therefore critical if we are to create a framework that will be an effective software engineering environment.

The generic application described herein, is in contrast to those currently available, such as MacApp and ET++, that equip the user with a standard user-interface for any application in any domain. These frameworks do not set out to have any knowledge of the application domain‡ and thus emphasise software development after the application has been coded, by providing a front-end for it. RApp, however places its emphasis on the design and development of applications within one domain, providing generic, domain-specific components. We can say therefore that they address the task of software development in

† This programming paradigm could be seen as "programming by filling in the blanks" [Pro89a].

‡ They do however have knowledge of the interface domain.

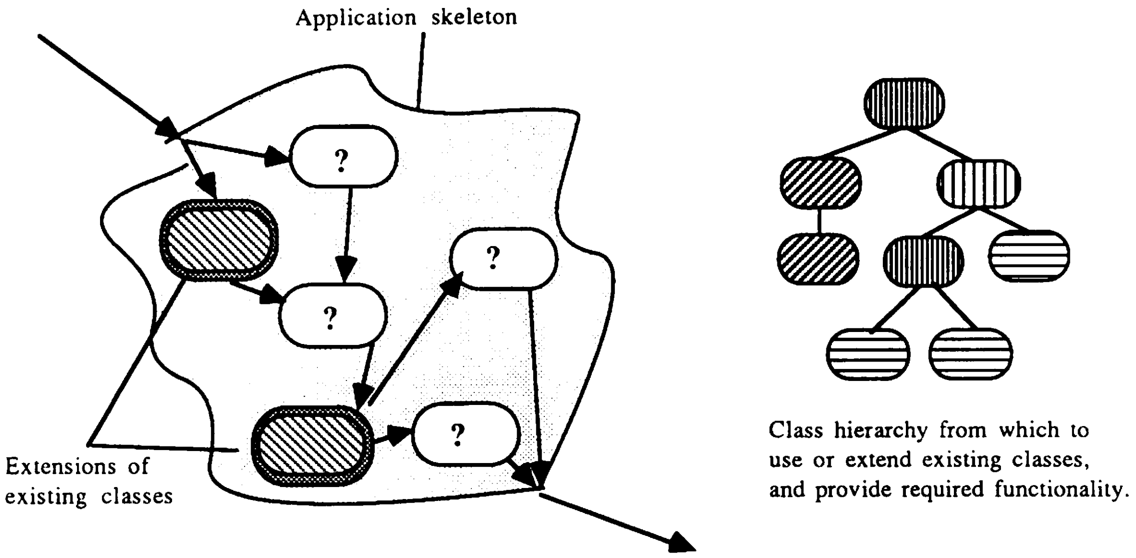

Application skeleton
?
?
?
?
?
Extensions of existing classes
Class hierarchy from which to use or extend existing classes, and provide required functionality.

Figure 2: *Reuse of existing classes using object-oriented technology*

different domains. MacApp type frameworks, address the **post-development** phase of applications, whereas RApp is used throughout the **development** of applications within the domain. It is important that this difference be clearly defined in order that we understand the reasons behind investigating such a paradigm. Although they have differing aims, the two forms of framework can be compared and contrasted with respect to their structure, and their approach to software development.

The generic application benefits from the advantages to be gained by both reuse of components [Mat84a], and by object-oriented programming [Mey87a]. In this way, the use of a generic application can provide a design approach that lays emphasis on both use of relevant components (reuse through composition), and on the organisation and composition of these components into modular solutions. There is an opportunity here for more than just bottom-up reuse.

3. The Architecture of RApp

RApp, a generic **R**outing **App**lication framework, for the VLSI Routing domain currently contains some 90 classes and 480 methods. Object-oriented concepts, expressed in were used to capture the common features of area routing algorithms and encapsulate them into one inheritance hierarchy.

The design process was an iterative one [Gos89c] and required ongoing consultation with domain experts, and a review of existing routing software in order to achieve a set of clear, well-behaved abstractions meaningful to routing experts.

The RApp hierarchy consists of four types of classes: Application Classes, Domain Classes, Factor Classes and Bridging Classes. We now describe the four types of classes, emphasising the underlying architectural philosophy behind RApp: that of maintaining generality at higher levels and specificity at lower ones [Gos89a].

3.1. Application Classes

The application classes are high level classes that extract the commonalities of area routing applications, providing a control structure for any application within the domain. Examples are *RoutingSurface*, *RoutingTask* and *ApplicationObject*.

```
class RoutingSurface {
      RoutingSurface();
public:
      virtual int size();
      virtual point * topLeft();
      virtual point * bottomRight();
      virtual int
            calcManhattanDist(point*, point*);
      virtual void print(OutputDevice*);
};
```

Figure 3: *Definition of Application class RoutingSurface*

The *RoutingSurface* class is an abstract class that encapsulates the general features of any surface that may be routed upon, be it grid based or grid less. It is a representation of a real domain entity and its subclasses provide the specialised behaviour required of particular types of routing surfaces. This specialised behaviour is provided by implementing virtual methods of its base class.

Therefore, for the class defined in figure 3, a sub-class would determine the value returned by "calcManhattanDist", and "size". It would also be responsible for the rest of these operations and any others, defined in its own declaration. It acts as a template for its derivatives in that it has unimplemented methods which its subclasses are expected to implement; these methods we have termed "pluggable methods".

The *RoutingTask* however, represents a packaging of information for routing an area. This is again an abstract class, but is not modelling a real entity. Once again, this class provides the outline behaviour for all its future descendants, in this case particular types of routing tasks. Each routing application can have a limitless number of routing tasks, each describing one individual routing problem. It provides a good example of how information can be encapsulated in order to provide capabilities for reuse and data-hiding.

The *ApplicationObject* controls the high level management of the application. The user customises this class to create the application object for their type of routing algorithm. We can distinguish a generic application from a kit of parts by the existence of an ApplicationObject class. Whereas a kit merely provides the components with which applications are constructed, a generic application provides the skeleton of all applications within the domain, through the ApplicationObject and its derivatives. The developer customises the ApplicationObject class for the application in-hand; only one exists in any application. See figure 4.

```
class ApplicationObject {
      Router* aRouter;
      TaskList* setOfTasks;
public:
      ApplicationObject(istream&);
      ApplicationObject(Router*, isream&);
      ~ApplicationObject();

      virtual int run();
};

class MyLeeRouterApplication {
      RouterType typeOfRouter;
      OutputDevice* myOutputDev;
public:
      MyLeeRouterApplication(Router*, istream&);
      ~MyLeeRouterApplicationObject();

      int run();
};
```

Figure 4: *Definition of ApplicationObject and MyLeeRouterApplication*

All applications within this domain require an instance of each of the Application Classes, or their immediate sub-classes. They are therefore made as abstract as possible, users can subclass as required to implement specific functionality. It is important to note that they model both real world entities (e.g. RoutingSurface) as well as packages of information (e.g. RoutingTask). The modelling of both real-world concepts and processes equally, as classes, allows the designer and developer the freedom with which to manipulate such classes in order to experiment, and prototype solutions rapidly. This is one of object-oriented programming's principal strengths.

3.2. Domain Classes

The class hierarchy aims to provide an accurate representation of the problem space through a set of domain relevant classes that are meaningful in name and behaviour [Gos89a]. This collection of classes includes both abstract and specialized representations of area routing concepts.

An example is the class *grid* which is a high level class of a routing grid providing general operations such as calculation of manhattan distance between two points or calculating grid area. Its subclasses, *cellBasedGrid* and *segmentedGrid* store the data representations, override and refine general behaviour if necessary, and also provide the more specialized operations one would associate with routing grids. Other examples include *obstacle, wavefront, line, polyline and via.*

The underlying requirement of a domain class is that it represent a concept of the real world, in this case the world of VLSI area routing. Each class should be used as an autonomous components, and this should be ever-present in the mind of the designer during the conceptualization of a component. In attempting to match the class with a domain concept, expert assistance is necessary, as the ultimate aim is to maximise the legitimacy of a particular component.

3.3. Factor Classes

Factor classes provide the potential for reuse of algorithms through the mapping of processes common to various routing algorithms. For example, the Lee algorithm [Lee61a] and its derivatives [Rub74a, Kor82a], can be factored out into distinct processes represented by abstracting these processes into classes for the high-level common features modelled by the general classes, with more and more details handled as we descend the class hierarchy.

The Lee algorithm operates on a cellular grid expanding outwardly in circular waves, from source to target, numbering each cell as it progresses. Once the target has been reached, the shortest route is determined by retracing back to the source using decreasing cell numbers. Its derivatives, by Rubin and Korn, introduce cost determination as a means of directing the expansion towards the target. These costs are calculated using distances. These derivations allow for sharing in the expansion stage, which is modelled by the high-level class Expander and its sub-classes. Figure 5 shows the definition of class Expander and one of its sub-classes.

We can see that only the operation "expandOneStage" need be overridden, as the behaviour for expansion only differed in the expansion of each stage. Additional information required by CostExpander is provided by the private data member *endCell,* with cost determining functionality contained within "findLowestCostCell" and "calcCost". Overall control was still with Expander. This example shows the power of the object-oriented paradigm, through its expressivity and its extensibility.

The encapsulation of such processes into classes with associated operations affords the designer many advantages. The flexibility and power of such design is reflected in the numerous customisations and variations possible. Johnson and Foote outline some of the advantages to be had in modelling proceses as classes and also provide some design guidelines [Joh88a].

```
class Expander {
      int noOfSteps;
      int count;
      int searchFlag;
public:
      // constructors etc.
      ..
      virtual int expandAllStages();
      virtual int expandOneStage();
      virtual StaggeredExpand();
      void setSearchFlag(int s);
      int getSearchFlag();
      int incrementCount();

      // ..more functions here
};

class CostExpander {
      Cell* endCell;
public:
      // constructors etc.
      Cell* getEndCell();
      int expandOneStage();
      virtual Cell* findLowestCostCell(Wavefront*, Grid*, Cell*);
      virtual float calcCost(Grid*, Cell*, Cell*);
};
```

Figure 5: *Definition of class Expander and a sub-class CostExpander*

3.4. Bridging Classes

A simple set of classes that are used for the internal representation of many of the other types of classes; this would include lists of vias, arrays of cells etc. They are called Bridging classes as they provide the link between the computer science domain (e.g. lists, arrays etc.), and the VLSI Routing domain (e.g. cells, vias etc.). They are not very "spectacular" classes, but are nonetheless as important as all of the other categories of classes.

4. Application Construction

To develop a RApp application, the user must design their own subclasses of ApplicationObject, RoutingTask, RoutingSurface and *Router*. This last abstract class, Router, controls the overall structure of the algorithm part of any application; once again by providing function templates and overall behaviour. Its subclasses would therefore be created for the different algorithms. A small sub-hierarchy descended from Router exists, with generality being dealt with at high levels and specificity at lower ones, thus increasing the potential for reuse.

The definition of the abstract class Router and a sub-class, for the implementation of the original Lee routing algorithm is shown in figure 6. The class LeeRouter is created by the developer to provide the overall algorithm control for the Lee routing algorithm. It overrides methods from Router and also provides many of its own, not all of which are shown here.

We can see from figure 6 that there exists a member function "expand", for LeeRouter. This operation creates an object of the class Expander (see figure 5) with various parameters, which is then "told" to expand according to some strategy. Once the expansion is complete, details are taken, and the instance of Expander is destroyed. in a subsequent application, we may require a variation of the Lee algorithm, thus requiring us to modify the existing strategy. A sub-class for LeeRouter would then override "expand", pass some additional information to a sub-class of Expander, and the expansion would thus be executed. This sub-class of Expander would probably only override those member functions that contained the functionality no longer required, as in figure 5.

In this way, the developer only need concentrate on the functionality required of the new application. Overall control of the application, would be as before. Such a methodology of development provides for a powerful paradigm.

```
class Router {
      Router();
public:
      virtual int routeIt();
      virtual void printResults(OutputDevice*);
      virtual int successOrFail();
};

class LeeRouter {
      int noOfSteps;
        routingTask* theTask;
pubic:
      // constructors and destructors
      // ..
      void routeIt();          // overrides that of Router
      void printResults(OutputDevice*);

            // virtuals for subsequent derivations
            // of Lee algorithm.  e.g.  Rubin derivation

      virtual int expand(LeeTask*);
      virtual void retrace(LeeTask*);
      virtual LeeTask* getNextTask();
      // ..  few other methods
      // ..
};
```

Figure 6: *Definition of Router and LeeRouter, a user defined class*

If the new algorithm to be implemented is derived from an existing one, then the design process is made easier by the existence of relevant classes, a ready-made design structure and design artefacts, and in all probability, much of the functionality has been implemented by existing algorithms. All that is required is the code specific to the new algorithm.

If the algorithm is a new one however, then the user must introduce new classes into the framework in all of the four types of classes. Some of the design activities would be:

(1) Identify commonalities with existing algorithms

(2) Identify new entities

(3) Use or extend existing classes to create new ones

(4) Integrate classes into the hierarchy

(5) Code application as for previous algorithm

This new algorithm would require sub-classing for various Application classes to provide the type of Router required, the kind of RoutingSurface and the particular features of the RoutingTask would need to be catered for. In addition to this new factor classes would no doubt be needed, along with some domain classes to introduce new concepts. The overall structure of the application would be the same and the developer should investigate the possibility of sharing commonalities with existing classes.

The existence of design documentation from previous designs can only be beneficial in understanding the structure of the class hierarchy and the interactions between classes. Refer to [Gos89a, Gos90a], and [Gos89c] for our experiences in designing applications within the generic application paradigm.

We have found that the best way to approach an object-oriented design for an application domain is to design iteratively, with representation and reusability as our two main goals, reusing as much of the existing design structure as possible. This implies reuse of documentation from previous designs as well as reuse of code.

5. Reusability

5.1. Code Reuse

This is the form of reuse that most people associate with "reusability". Object-oriented technology encourages code reuse through extension and modification of existing classes using inheritance, hence not changing any existing code. Such reuse is expected, and certainly achievable; yet there is a different aspect to reuse that highlights the advantages of both object-oriented technology and the generic application paradigm.

The code reuse possible from extending classes that model processes allows us to manipulate processes as if they were entities, we can thus extend and modify without altering the initial code, such is the advantage afforded by object-oriented programmimg. The generic application allows us to concentrate on the application in hand, and the differences from existing algorithms. Only the additional functionality need be added, as all other functionality is provided by the generic application. This extra dimension to application programming is only possible using object-oriented techniques.

To illustrate both the above points, we attempt to show how much algorithm-specific† code was required in order to implement two routing algorithms, derived from an existing one. Such reuse, being algorithm-specific, was within the various groups of Factor Classes.

The graph in figure 7 shows how much code was reused from the original implementation of the Lee algorithm. The graph has been divided into three columns to show the amount of reuse within those stages of the algorithms and to highlight the point that changes have been isolated. The graphs shows, by appropriate shading, the source of the code for each stage of the algorithm as a percentage of the total algorithm-specific code required for the current application.

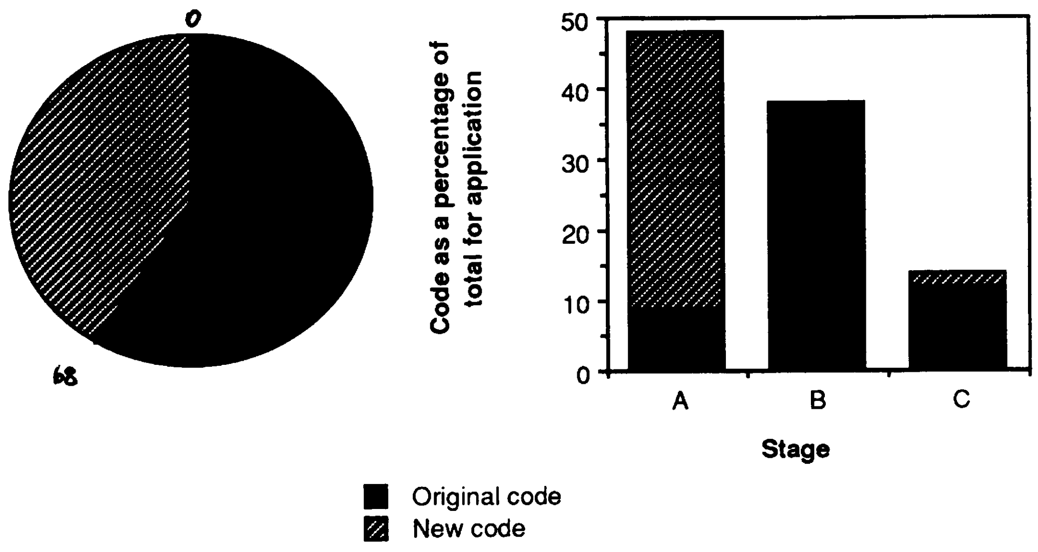


Figure 7: *Amount of code reuse in implementation of Rubin's variation of Lee algorithm as percentage of total code for new application*

We can see that in the above implementation, the majority of work was involved in the design and coding of stage A as this is where the refinements were made to the initial algorithm. The mechanisms for stages B and C is still handled by the original construction. The pie chart beside the graph of figure 7 expresses the amount of code reused as part of the whole code for the new algorithm. That is, of the total code for the new algorithm, some 59 percent originated from existing code.

The implementation of the Korn algorithm reused code from both its predecessors, the original algorithm and its derivative, described above. Figure 8 shows how the functionality has been distributed throughout instances of the original, and subsequent classes.

An examination of the graph shows that, as before, the majority of code written was for stage A of the algorithm as this is where it differed from previous algorithms. Therefore, only code that affected stage A of the algorithm needed to be examined. Had the changes been in stage B, then only code relevant to that stage would have needed to be examined. Such advantages can be attributed to the inherent modularity of the object-oriented paradigm.

The pie chart once again shows the code reused as part of the whole code for the new algorithm. Of the total code, some 50 percent came from the original implementation and 26 percent from the previous implementation. Such reuse meant that only 24 percent of the total code for that application had to be written.

† Algorithm-specific code refers to all code concerning the implementation of the algorithm, including code for the creation, implementation, and destruction of any objects involved.

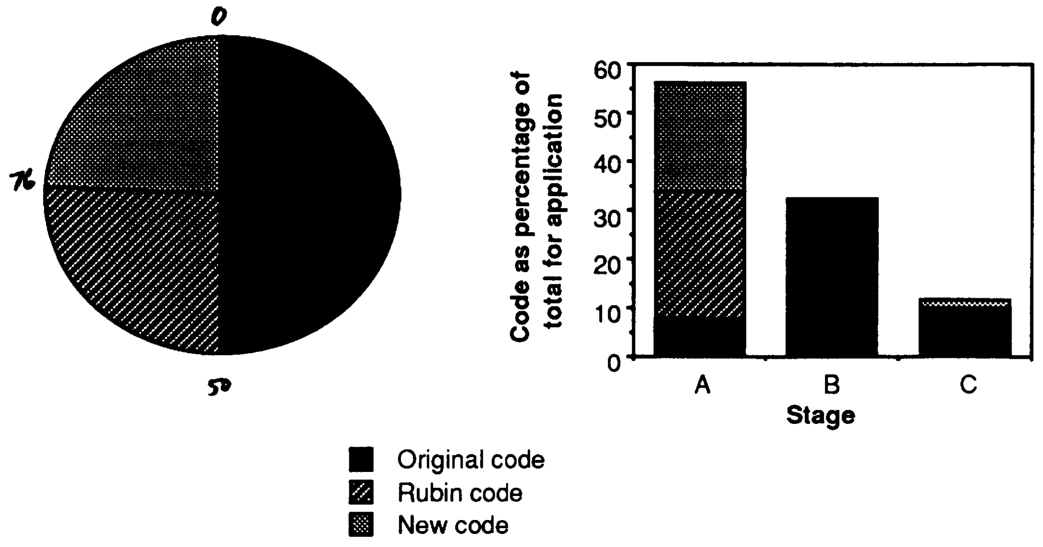


Figure 8: *Amount of code reuse in implementation of Korn's variation of Lee algorithm as percentage of total code for new application*

The two figures accurately reflect how the object-oriented paradigm can reuse code from its ancestor classes, and also illustrates the power of the generic application whereby the developer need only write code for those parts of the software that are algorithm specific. It is important to point out that *no other code was required* other than that which directly affects the algorithm being implemented.

One naturally begins to draw comparisons with traditional approaches to programming. Had the original implementation been in C, would we have been able to achieve similar measures for code reuse? The answer is not a simple one, although we can offer some ideas which may lead us in the right direction.

Reuse with procedural programming is possible through careful, well designed software, that is designed for reuse.† Object-oriented programming however, makes reuse a natural part of the development process. It is facilitated, and even encouraged, through inheritance and encapsulation, but with procedural programming such attitudes to reuse must be worked on and are harder to achieve with such ease. Such reuse is therefore possible, but the effort and time required to achieve such figures makes it unfeasible and also very difficult.

5.2. Design Reuse

Unlike code reuse, the reuse of designs is at a much higher level, and as it is implementation independent, has the potential for being reusable over a wider variety of applications. The conceptual base upon which our beliefs are founded aims at exploiting the commonalities that exist between applications within the same design, and more importantly, from a design perspective, the commonalities that exist between their designs.

It is difficult to prove the reuse of designs from existing applications as many facets of a design can influence a new software design. We do however recommend, in another paper [Gos90a], various design by-products that we have found useful in subsequent designs within the generic application paradigm. Such by-products include design diagrams, decision histories, and the design rationale behind object creation. Our reuse of designs has provided the motivation for us to now approach the task of software design from a reuse perspective, and as such has persuaded us to continually look for reuse and refinement of existing classes.

† One can design for reuse through decomposing code sensibly using information hiding techniques, and reusing various modules. See [Par72a].

6. Discussion and Conclusions

In this paper we have outlined an approach to software development within domains, using object-oriented programming — the generic application. Using object-oriented technqiues, we have provided flexibility, maintainability and reusability, in the form of a class hierarchy. Through a general design philosophy, we have attempted to provide domain relevant, reusable classes.

The essential feature of all applications within a domain are represented by the generic application, with the developer needing only to add the functionality specific to any application. As more and more applcations are coded using the generic application, applications can be built on top of others, requiring only the functionality that differentiates them from existing applications. In this way, the class hierarchy can be seen as an evolving, growing structure, that could be regarded as a model of the problem domain.

An application framework allows an application programmer to concentrate on the algorithm at hand, in this case providing up to 76 percent actual software reuse. The less tangible improvements such as quality and productivity have later pay-offs through extensibility and maintainability, but the beginnings of these are made apparent through our measurements of the amount of code having to implement an algorithm.

The generic application paradigm encourages reuse of classes within a compositional approach to software reuse. It is also an approach that utilises the reuse of patterns. We have been able to reuse existing design structures and existing design by-products in subsequent designs, which can be considered as a form of patterns. This "structural reuse" [Ste88a] allows the designer to concentrate on ensuring compatibility of protocols, and conformity of sub-classes, rather than having to design many new protocols, interfaces etc.

References

[And88a] B. Anderson, "Object Oriented Programming," *Microprocessors and Microsystems*, vol. 12 (8), pp. 433-442, October 1988.

[Big84a] T.J. Biggerstaff and A.J. Perlis, "Foreword for special issues on Software Reusability," *IEEE Transactions On Software Engineering*, vol. SE-10 (5), pp. 474-477, Sept. 1984.

[Gos89a] S. Gossain and D.B. Anderson, "Designing A Class Hierarchy For Domain Representataion And Reusability," *Technology of Object-Oriented Languages and Systems 1989 Proceedings*, pp. 201 - 210, Paris, November 1989.

[Gos89b] S. Gossain and D.B. Anderson, "An Object-Oriented Aproach To VLSI Routing," *IEE Colloquium on Applications of Object Oriented Programming*, IEE, London, November 1989.

[Gos89c] S. Gossain and D.B. Anderson, "An Iterative-Design Model For Reusable Object-Oriented Software," *Submitted for publication*, University of Essex, Colchester, UK, July 1989.

[Gos90a] S. Gossain and D.B. Anderson, "Towards The Reuse of Design Artefacts In Object-Oriented Software Development," *Submitted for publication*, University Of Essex, UK, January 1990.

[Joh88a] R.E. Johnson and B. Foote, "Designing Reusable Classes," *Journal Of Object Oriented Programming*, pp. 22 - 35, June/July 1988.

[Kor82a] R.K. Korn, "An Efficient Variable-Cost Maze Router," *Proc. of the 19th Design Automation Conference*, pp. 425 - 431, 1982.

[Lee61a] C.Y. Lee, "An Algorithm For Path Connection and Its Applications," *IRE Transactions on Electronic Computing*, pp. 846 - 865, Sept. 1961.

[Mat84a] Y. Matsumoto, "Some Experiences In Promoting Reusable Software : Presentation in Higher Abstract Levels," *IEEE Transactions On Software Engineering*, vol. SE-10(5), pp. 502-513, Sept. 1984.

[Mey87a] B. Meyer, "Reusability : The Case For Object Oriented Design," *IEEE Software*, vol. 4(2), pp. 50 - 64, March 1987.

[Par72a] D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications Of The ACM*, vol. 15(12), pp. 1053 - 1058, December 1972.

[Pro89a] A. Proefrock, D. Tsichritzis, G. Muller, and M. Ader, *ITHACA : An Integrated Toolkit for Highly Advanced Computer Applications*, Centre Universitaire d'Informatique, Universite de Geneve Switzerland, 1989.

[Rub74a] F. Rubin, "The Lee Path Connection Algorithm," *IEEE Transactions On Computers*, vol. C-23(9), pp. 907 - 914, September 1974.

[Sch86a] K. J. Schmucker, "MacApp : An Application Framework," *Byte*, pp. 189 - 193, August 1986.

[Ste88a] L. Stein and H.H. Porter, "Summary of Discussions from OOPSLA-87's Methodologies & OOP Workshop," *OOPSLA 87 Addendum to the Proceedings*, in SIGPLAN Notices, Vol 23, No 5, pp9-16., May 1988.

[Wei88a] A. Weinand, E. Gamma, and R. Marty, "ET++ - An Object Oriented Application Framework in C++," *OOPSLA 88, SIGPLAN Notices*, vol. 23 (11), pp. 46 - 57, November 1988.

PatMat — a C++ Pattern Matching Class

Peter Polkinghorne

GEC Hirst Research Centre
East Lane
Wembley
Middlesex HA9 7PP.
UK

pjmp@gec-rl-hrc.co.uk

ABSTRACT

The paper describes the process of using an Object Oriented language to design and implement a pattern matching system for operating on text. The pattern matching system is implemented as a C++ *class* called *patmat.* The design is based on *Snobol4.* The object oriented facilities provided by both provide a good user interface and a mechanism for the implementation. Details of the development statistics are given along with the tools used. The performance of a *patmat* based *grep* program is compared with a variety of others. Finally a detailed description of the class interface is given.

1. Introduction

This paper describes the process of using an Object Oriented language to design and implement a pattern matching system for operating on text. The pattern matching system is implemented as a C++ .I class called *patmat.* A *class* is the C++ term for the type of an Object. The paper includes statistics of the development process along with performance comparisons with other pattern matching systems. The appendix provides a detailed description of the *patmat* class.

C++ was chosen as the development language, because it is an efficient Object Oriented language for the UNIX environment and because of previous success in using the language. The *Snobol4* language's model of pattern matching was chosen for its flexibility and power.

This work was initiated as part of an Alvey Project, CARDS [Fos88a] (Conceptual And Relational Database Server) which sought to extend the capabilities of the traditional Relational Database systems. The pattern matching facilities were required by the application developers, ICRF (Imperial Cancer Research Fund), for their biological database.

2. Pattern Matching Requirements

The initial decision we had to make was: what type of pattern matching did we want? Examination of the literature indicated five main types of system:

i) Based on Boyer-Moore [Boy77a], there are systems for matching fixed strings or sets of fixed strings rapidly.

ii) Deterministic and fast methods exist for matching regular expressions, for example Liu's work [Liu81a]. These are often used in editors and for example *grep.*

iii) The compiler technology derived matching or parsing, is another type of system. This uses limited context free grammars, such as LR(1) [Aho77a], to enable the construction of deterministic parsers.

iv) There is the *Snobol4* [Gri71a] approach, which while less theoretically elegant, offers a wider variety of facilities, derived from the needs of string processing. It uses a non-deterministic algorithm, incorporating context free grammars.

v) Finally there is the approach used to compare how similar two strings are. This is exemplified in Sankoff & Kruskal's work [San83a]. This style of matching is *fuzzy*.

We decided to offer a wide variety of facilities and follow the *Snobol4* approach. The theoretical basis for the work, was a paper by Gimpel [Gim73a] giving a retrospective formal framework for *Snobol4*.

3. Formal Framework

Patterns are made of *elements*, *builders* and *wrappers*. A pattern is matched against a *subject*. This matching can be used to either yield the first match or all possible matches. The *cursor* is used to indicate where the match currently is or the region matched. The cursor positions are numbered between characters in the subject thus:

$$_0e\,_1x\,_2a\,_3m\,_4p\,_5l\,_6e\,_7$$

The *elements* perform a simple matching action based on cursor position or subject. The elements all have the following property: for a given subject and position in it, they either match a part of the subject, possibly zero length, or they do not match. In Gimpel's terminology they are *monic*.

The elements are joined together to form *path expressions*. This means each element can have a subsequent element and an alternative element. The alternative and subsequent elements are established by the *alternation* and *concatenation builders*. How the path expressions are formed is described below.

To match a path expression, one keeps on matching elementary components and proceeding to the next elementary component in the case of success, until there is either no next component in which case the pattern has matched, or in the case of failure, back up the path until there is an alternative, in which case that is tried. If there is no alternative left, then the pattern match fails.

The pattern *builders* which join together the *elements* and *wrappers* to build more complex patterns. These consist of *Concatenation*, *Alternation*, *Reference* and *Repetition*. *Concatenation* and *Alternation* are the infix operators & and | respectively from the user view, linking elements. & having a higher precedence than | as in C. For example:

"Bat" & ("man" | "girl") | "Robin"

would match any of:

Batman Batgirl Robin

Reference allows a pattern to reference another pattern, including itself. This allows for Context Free Grammars [Hop79a] to be defined.

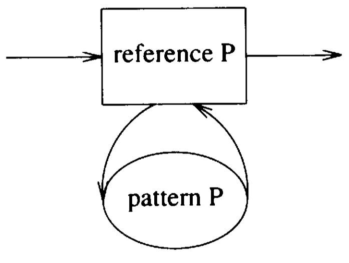


Figure 1: *Pattern Reference*

The pattern reference, as in figure 1, allows recursive patterns such as: *P = ("a" & ref(P) & "c") | "b"* which matches the language

$$a^n bc^n$$

Repetition (implemented by the *Arb* element) takes a given pattern and repeatedly matches it some number of times. The required number is in a range, with the initial and final possibilities given. At first the initial number is tried and on failures or backups, subsequent numbers are used until success or no more possibilities are left. An infinite and zero number of matches are allowed possibilities. The ordering of the initial and final numbers determines whether the initial match will be *greedy* or *parsimonious* that is match as many times as possible or a few as possible to ensure overall pattern match.

The *wrappers* are used to wrap a section of the pattern and carry out an action. The section of the pattern can be made up of more than one element, making a path expression, however it has one entry point and one exit point.

There are two times to carry out the action, the first is *unconditionally* when the wrapped subsection matches; the second is *conditionally* that is when the whole pattern has matched. This means that *unconditional* actions can take place many times owing to backup and retry. The actions available are string assignment and function execution. For function execution, the unconditional function returns a truth value which is taken as a sign of success or failure for that section of the pattern.

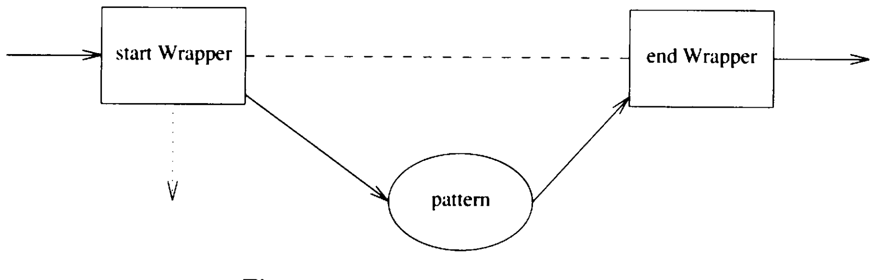


Figure 2: *Pattern Wrapper*

The dotted line represents an alternative. The dashed line represents a structure linkage.

4. User View

One of the traditional problems for users of pattern matching facilities, has been unpleasant interfaces and lack of flexibility, when grafted on to existing languages. Indeed one of the most widely used *Regexp*(3) in UNIX, comments *"The interface to this file is unpleasantly complex." [(nu88a]*. The other approach used, has been to build little languages such as *awk* [Aho88a] or *Perl*, which provide fine software tools, but are not of much help within a program except for large pieces of work, when it is worth running a process.

One of the advantages of using is the ability for a class to define its own meaning for operators, thus forming an abstract data type. This leads to a more natural style of pattern specification. Another important aspect is the ability to dynamicly build up patterns, as opposed to having predefined strings. Finally it is possible to extend the class to encompass further matching operations. The whole pattern matching system is available in a library that extends the C++ language. See the appendix for a description of *patmat*(3).

Regexp	**PatMat**

`"\(.*\)\1"`	`char *repeat=0;`
	`p = uas( &repeat, arb( -1, 0, len(1) ) & rs( &repeat );`

`"[a-z][a-z0-9]* *( *[a-z][a-z0-9]*[^)]*)"`	`id = any("abcdefghijklmnopqrstuvwxyz" ) &`
	` arb( -1, -1, any("abcdefghijklmnopqrstuvwxyz0123456789") );`
	`w = nil | span( " " );`
	`p = id & w & "(" & w & id & until( ")" ) & ")";`

Table 1: *User Interface Comparison*

As examples of the difference between *regexp*(3) and *patmat*(3), the table 1 is given. As can be seen *patmat* is more verbose and less cryptic. This is particularly valuable for complex patterns and for ease of definition. In particular if in the second example, upper case letters were now permitted in ids, only id would have to be change, whereas with *regexp* all the explicit occurrences of id would have to be changed.

5. Object Oriented Design

This pattern matching scheme is implemented in C++ with a user visible class *patmat* which implements the pattern abstract data type. This represents the facade for the user to see and operate on.

However the work and theory is implemented by a base element class *pat*, used to produce a basic null element, which has the matching operations, the path expression structure and other operations encoded. Classes are derived from *pat* by inheritance, to implement other types of elements. This meant the elements had only to implement those features that differed from the null element. This usually consisted of the match action, construction and output. We only had a two level inheritance hierarchy, whereas it would have been possible to move to a three or more level hierarchy to exploit commonality between the various pattern elements.

There are supporting classes - *patchar* (pattern characteristics used for optimisation), *stack, sstack* (system stack) and *node*. At the time of implementation, these were not available from existing libraries, so we used our own implementations. However it would have been better to derive them from existing stack and tree classes, thus promoting code reuse.

5.1. Pattern Expressions

A pattern expression consists of a number of elementary items joined by concatenation or alternation operators thus:

(e1 & e2 | e3) & (e4 | e5 | e6)

By the operators defined for *patmat* they become an expression tree linked by *node* elements.

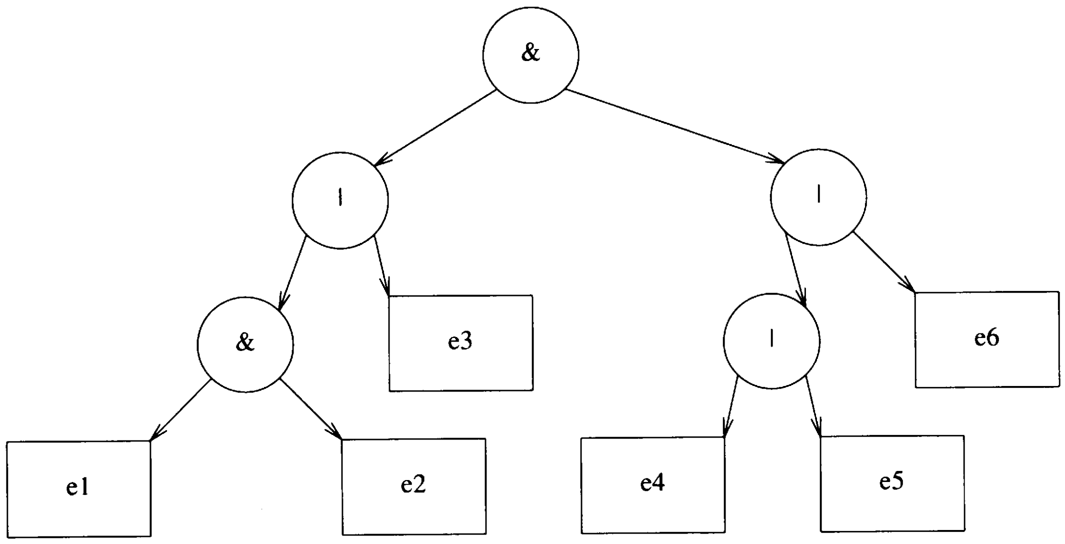


Figure 3: *Expression Tree*

These trees can be converted [Gim73a] into path expressions. This leads to a more efficient representation. So the preceding diagram changes to the following with dotted lines representing alternate paths and solid lines representing the next component. This conversion is carried out when a *node* type is assigned to a *patmat* type. This demonstrates the utility of the detailed control given by C++ on type conversion.

5.2. Optimisation

It is possible to optimise pattern matching performance by a number of methods. The first is to collapse subsections of the path into a new element. At the extreme this could include building finite state machines or building Boyer-Moore type data structures. The second is to apply heuristics about the pattern matching process. The heuristics used are derived from *Snobol4*, but are not identical. In particular there is no difference in matched results whether heuristics are applied or not. The only difference being in the unconditional actions carried out.

The approach taken was to apply heuristics. This was done at two levels. Firstly to derive the characteristics of the individual elements, that is the local characteristics. This means seeing if the element defined position either from beginning or end, minimum or maximum length, or start character set. Then for the path expression from that element on, define the global characteristics. These can be combined by simple rules. See figure 5.

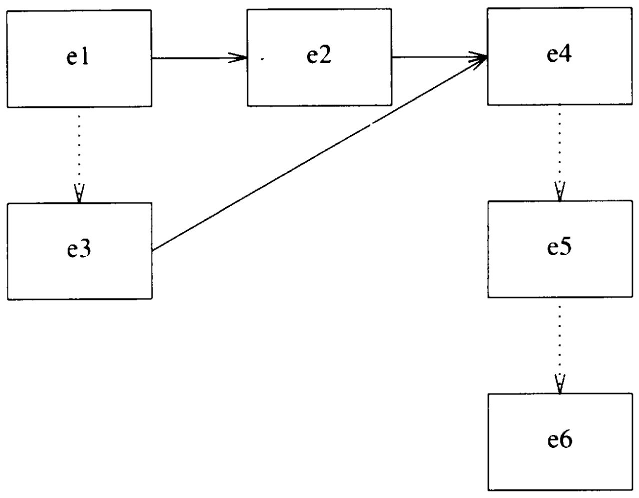


Figure 4: *Path Expression*

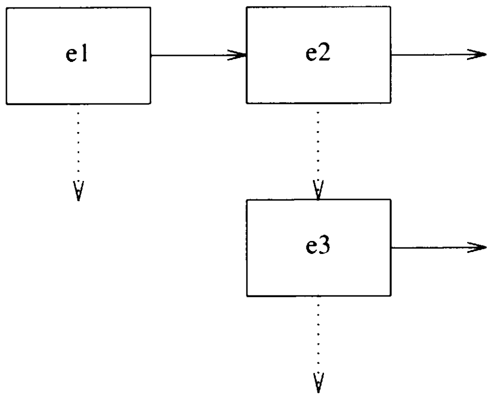


Figure 5: *Path Expression*

The global characteristics for *e1* are derived by considering the global characteristics for *e2* and *e3* along with the local characteristics for *e1*. So if *e1* is of zero length, the initial character set is derived by combining the global sets from *e2* and *e3*. Otherwise the set from *e1* is used. These rules are all implemented in the *patchar* class.

This design strategy of implementing the pattern characteristics in the *patchar* class has many advantages. Firstly it assisted simplifying the *pat* class. Secondly it allowed for incremental implementation of the rules for combining the pattern characteristics. Finally it allowed for the hard cases of *arb* and *ref* to be handled after the easier cases had been dealt with.

This is applied using the global characteristics to determine whether matching is worthwhile at the current point, or whether one should give up. The global is also used to determine where to start matching.

6. Implementation

Given the above theory, design and the C++ language, an implementation was made. The author of the paper was the sole person working on the implementation, which took roughly 12 weeks to design, code and test, although some extensions were made (in particular the *nompat* class was added – see section 6.3) and bugs were fixed in the subsequent months.

The figures in table 2 were derived as follows. The RCS logs recording changes to the various files were used to derive: *major bugs:* that is bugs recorded as being fixed; *enhancements:* that is changes internal to a class; and *class changes:* that is changes to the class interface. It should be born in mind that these statistics are rough. Classes defined early on such as *str_p* are likely to have undergone more changes because of changes in *pat* than classes such as *rs_p* defined later on.

File	lines	major bugs	enhan-ments	class changes	description
any_p.c	111	0	3	1	any pat subclass
arb_p.c	318	3	8	1	arb pat subclass
cas_p.c	127	1	2	4	cas pat subclass
cexf_p.c	130	0	1	1	cexf pat subclass
cur_p.c	72	1	3	1	cur pat subclass
len_p.c	78	0	3	1	len pat subclass
nany_p.c	108	0	3	1	nany pat subclass
node.c	315	7	2	2	node class
nompat.c	907	2	0	0	nompat class
pat.c	401	6	14	3	pat class
pat_ar.c	402	2	3	0	dump & restore part of pat class
pat_io.c	283	0	2	1	IO support routines
patchar.c	327	1	3	0	patchar class
pattern.c	885	4	16	14	patmat class
pos_p.c	70	0	4	1	pos pat subclass
ref_p.c	90	2	6	1	ref pat subclass
rpos_p.c	71	0	4	1	rpos pat subclass
rs_p.c	90	0	2	1	rs pat subclass
rtab_p.c	76	0	4	1	rtab pat subclass
span_p.c	119	1	3	1	span pat subclass
sstack.c	209	1	4	1	system stack class
stack.c	324	0	5	4	stack subclass
str_p.c	111	0	5	1	str pat subclass
tab_p.c	77	0	4	1	tab pat subclass
uas_p.c	141	1	6	1	uas pat subclass
uexf_p.c	128	0	1	1	uexf pat subclass
until_p.c	116	1	3	1	until pat subclass
i_pat.h	821	0	0	30	internal patmat definitions
nompat.h	124	0	0	3	nompat definitions
pat.h	140	0	0	22	patmat definitions
Total	7171				
regexp	1337				Henry Spencer Regexp(3)

Table 2: *Development Statistics*

However what it does show is that most of the pattern elements derived from *pat* were implemented in 70 to 130 lines and were remarkably trouble free. The disadvantage of having a facade class *patmat* is that each element added required a new friend builder and change to *pat.h*. While the code size is bigger than *regexp*, if one removes extra functionality provided by *nompat.c* and *pat_ar.c* code, and allows for the overhead by having many little files and large C++ class declarations, it is not that much larger.

Performance tests were carried out on a number of grep variants – *grep* – the SunOS supplied grep; *egrep* – the SunOS supplied egrep; *GNUgrep* – the FSF version of grep; *pgrep* – the *patmat* based version of grep, run both with heuristics on and off. Interestingly none was consistently best. Both *GNUgrep* and *pgrep* were compiled using the SunOS *cc*, with optimisation. *pgrep* was converted to C using AT&T Cfront 1.1. The target was a 4096 line file with 266,290 characters. It consisted of the following lines with the last being repeated 4095 times.

This is a text file for PATMAT performance tests.
This a boring standard line that appears a lot - i.e. ad nauseam

The table 3 is the result of running tests with various versions of grep on a Sun 3/50 running SunOS3.4. The tests were run 3 times with no more than 10% variation in the overall cpu time used.

Pattern	*egrep*	*GNUgrep*	*grep*	*pgrep(heuristics)*	*pgrep*
^	*3.2u 0.7s*	*4.1u 0.6s*	*1.8u 0.5s*	*9.4u 1.2s*	*9.8u 0.8s*
$	*3.3u 0.5s*	*8.0u 0.6s*	*5.9u 0.7s*	*9.6u 1.1s*	*30.5u 1.3s*
len(80)	*26.2u 0.4s*	*2.9u 0.4s*	*36.0u 0.3s*	*5.2u 0.7s*	*26.9u 0.6s*
ars	*3.7u 0.4s*	*5.7u 0.6s*	*2.8u 0.5s*	*16.0u 1.0s*	*25.4u 1.2s*
[a-z]{3,}	*3.1u 0.7s(+)*	*4.0u 0.8s(+)*	*2.5u 0.6s(V)*	*16.5u 1.0s*	*16.1u 1.1s*
"arn"|"arb"|"ars"	*3.5u 0.7s*	*5.6u 0.7s*	*N/A*	*27.3u 1.1s*	*78.5u 1.9s*

(+) means the pattern *[a-z]+* was used.
(V) means the System V version of *grep* was used.
len(80) was '..' for the greps.

Table 3: *Performance Statistics*

An important aspect of grep programs is their i/o performance [Hum88a]. So as an experiment, a version of pgrep was built called *sio* which did everything as pgrep did apart from matching. Depending on the arguments, it just read the input or read and output. The results indicate the slowness of the streams library used. Another comparison is the speeds of *cat* and *wc -c* both run on the same input.

sio - input	sio - input & output	wc -c	cat
4.5u 0.9s	8.5u 1.0s	2.1u 0.3s	0.0u 0.4s

Table 4: *Streams I/O Performance*

6.1. Portability

The system was also tried out on a GEC Series 63 super-mini running UNIX System V release 2. The code ran virtually straight off after a bug concerning false assumptions about the order of static & global variables' construction had been fixed.

However attempts to use the GNU G++ 1.35 compiler at the time, hit quite a few problems, and was abandoned owing to insufficient time. This was coupled with problems with the available debuggers.

6.2. Debugging

When using Cfront on the Suns, the debugger of choice was *gdb*, mainly because it seemed much less fragile than *dbx* and it had a good set of features. The one thing that would have been nice for an O-O style language, but I could not easily see how to do with the existing debuggers, would be the ability to step from function call to function call. This is because when using a class in especially with derived classes, many small functions will be invoked, differing upon the type of the object. The "source line at a time" model of both *gdb* and *dbx* is inappropriate for this. Particularly when there are constructor and destructor functions being invisibly invoked.

The other main tool for debugging was *snow*, a simple interpreted language. This was developed using *lex* and *yacc*. This allowed for both interactive testing of the library and predefined tests to be run for regression testing. For many bugs, the interactive mode of *snow* was most useful being more convenient than a debugger and quicker and more flexible than writing a oneoff test program to compile and run.

One of the biggest problems with C++ were largely connected with getting the constructors and destructors right. Other difficulties included deciding when to use a function and when to simply access a member. However one great advantage was the ability to implement features incrementally, both in terms of *pat* derived classes and superimposed classes, *nompat*.

6.3. Persistence

A difficult area was persistence. This is the idea of extending the life time of an object beyond the execution of a program. This was desirable from the users point of view so patterns could be stored in the database. Since C++ at present has no standard for storing objects on stable storage, we decided on a simple human readable ASCII dump format.

The problem of references to other objects, was solved by devising a *named* pattern class, *nompat*. This meant each object of the *nompat* class had a name, and a list of other names both referred to and referenced by. Thus storing and recovering patterns was a class wide operation. The problem of making other object classes persistent remained, but the scheme worked reasonably well.

7. Summary and Conclusions

C++ is a good software development language, in particular when compared to C, the obvious alternative. The ability to perform incremental and modular development by use of classes, coupled with the increased type safety, made the development of the complex data structures and algorithms more straightforward than they would have been in C. This is reflected in the time for the development of the system. C++ also has the advantage of quite good run-time efficiency. In comparison with the C based *grep* programs, the *patmat* implementation performed creditably.

C++ however does suffer from some problems. Firstly, being a comparatively new language, C++ support tools were not available, although this situation is changing. To fill this gap, the development of an interpreted language *snow*, with interactive and programmed operation, proved invaluable. A further problem is that the implementation of the *constructors* and related operators for building objects is error prone and source of much initial frustration. The relationship between classes in terms of *friend*, *private*, and *public* access is hard to get right.

In order to exploit the advantages of object oriented systems, a good design for the objects is vital. The *Snobol4* based design proved invaluable with the virtues of being extensible and quite efficient, with enormous flexibility and power. It is quite easy for someone with the library source code to *patmat* to add further pattern matching elements or incorporate it into another class, as was done with *nompat*. It is not as safe or as fast as some of the more restricted forms of pattern expression, but the performance is quite creditable especially with the aid of heuristics.

References

[(nu88a] *Regexp(3)*, SunOS Reference Manual, p. 1, Sun Microsystems, Inc, January 1988.

[Aho88a] Alfred Aho, Brian Kernighan, and Peter Weinberger, *The AWK Programming Language*, Addison Wesley, Reading, Massachusetts, 1988.

[Aho77a] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, Massachusetts, 1977.

[Boy77a] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *CACM*, vol. 20, pp. 762-772, October 1977.

[Fos88a] A. Foster, "Conceptual and Relational Database Server for Knowledge Based Applications," *IEE Colloquium on Knowledge Manipulation Engines*, IEE, February 1988.

[Gim73a] J.F. Gimpel, "A Theory of Discrete Patterns and Their Implementation in SNOBOL4," *Communications of the ACM*, vol. 16, pp. 91-100, February 1973.

[Gri71a] R.E. Griswold, J.F. Poage, and I.P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, New Jersey, 1971.

[Hop79a] John E. Hopcroft, Jeffrey D. Ullman, and I.P. Polonsky, *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.

[Hum88a] Andrew Hume, "A Tale of Two Greps," *Software - Practise and Experience*, vol. 18, pp. 1063-1072, John Wiley & Sons, Ltd, November 1988.

[Liu81a] Ken-Chih Liu, "On String Pattern Matching: A new model with a polynomial time algorithm," *SIAM Journal of Computing*, vol. 10, pp. 118-140, February 1981.

[San83a] David Sankoff and Joseph B. Kruskal, *The Theory and Practice of Sequence Comparison: Timewarps, string edits and macro molecules*, Addison-Wesley, 1983.

Network Management Gateways

Gabriele Cressman-Hirl

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
(415) 336-1085

gabi@eng.sun.com

ABSTRACT

The effectiveness of managing large and complex multi-vendor networks depends on the availability of network management platforms that specifically address the diversity of heterogeneous networks. This paper discusses design requirements, architecture models and implementation issues for managing different types of networks simultaneously. The theoretical discussion is illustrated by examples of a case-study conducted to integrate two different network management protocols and network architectures.

1. Network Management — An Introduction

With the increasing complexity and diversity of existing networks, effective automation of network management has become one of the most difficult and complex problems for the current network community. Growing competition among vendors and the aggregate of multi-vendor, multi-product systems make modern communication systems more and more difficult to administer. In the past the users of these services have been responsible for managing their networks with only architecture-specific tools. Today's network management applications have to address multiproduct network environments to ensure effective use of network resources.

The growing integration of networks has increased the load and traffic on communication lines creating bottlenecks. The necessity to control rapidly growing distributed systems introduces new challenges to solve efficiency and performance problems in networks. According to P.J. Brusil and D.P. Stokesberry,

> "the key to managing networks is determining what management decisions are needed. This, in turn, requires an understanding of what network resource parameters need to be controlled, what sequence of control actions is necessary to effect the management decisions, what metrics are required to make decisions, and what measurements are required upon network resources to estimate these parameters and metrics" [Bru89a].

This paper suggests possible solutions to these management requirements in heterogeneous networks, and discusses implementation issues in the context of integrating different network management models.

1.1. Typical Functions of Network Management Applications

To explain the complexity and diverse aspects involved in managing network resources, it is useful to describe network management features with the facilities defined in the OSI management framework [ISOa].

- **Fault Management** is the set of facilities that enables the detection, isolation, and correction of abnormal operation in a network. It involves functions to recognize error situations, maintain and examine error logs, trace and possibly correct faults.

- **Accounting Management** is the set of facilities that establishes charges for the use of managed objects, and determines the cost for the use of those managed objects.
- **Configuration, State and Name Management** provides the facilities to change system parameters and configurations, initialize and close down managed objects, recognize significant changes of state, and associate names with a managed object or sets of managed objects.
- **Performance Management** analyzes the behavior of managed objects and the effectiveness of communication activities. It involves gathering of statistical data, defining threshold values, interpreting traps or events, maintaining and examining logs of system state histories for purposes such as planning and analysis.
- **Security Management** addresses those aspects of security essential to operate a network correctly and to protect managed objects. It provides facilities to support authentication, control and maintain access, maintain and examine security logs.

Existing architecture-specific applications include many of the above functions. They tend to emphasize one or more aspects of the above management functions with respect to the special requirements of their customer base.

1.2. Network Management Implementation Concepts

Network management applications are usually implemented on top of a *manager-agent* protocol (see figure 1). The agent functions as a server for a device being managed and provides information that is requested by its client, the manager. The protocol used between manager and agent defines the communication facility through which management applications retrieve and change the data that they process for their specific purposes, such as performance statistics or alarm mechanisms. The management applications interface with the manager side of the protocol, and run on a host called, the *management station.*

Managed objects are devices or groups of devices such as hosts, gateways, bridges, terminal servers, printers, scanners, fax machines, modems or any type of network device. Protocol layers and other software services are also managed objects. These objects may be described in various ways. For example, to name a few, an application managing configurations views a host object in terms of operating system type, driver configuration, capacity. At the same time, a daily performance statistic manager views this host object in terms of throughput counters. Two aspects of managed object data are critical:

- **Name and Address to locate managed objects.** Usually the mechanisms of the native network architecture are used to name and address managed objects. Managing objects in heterogeneous networks raises additional issues related to naming and addressing that are discussed later.
- **Syntax and Semantics to describe managed objects.** Structures must be defined to describe the characteristics or attributes of each managed object — parameters such as statistical counters, operating system parameters, network layer data structures, control information, or events signaling any kind of change.

The conceptual repository of all this information is usually called the Management Information Base (MIB). The Management Information Base resides on the manager side and has a similar data representation on the agent side. To allow a protocol to communicate information about managed objects, manager and agent have to agree on the *syntax* of the Management Information Base. The actual content and interpretation of the Management Information Base depends on the type of objects being managed.

2. Architecture Models for Managing Heterogeneous Networks

This section defines design strategies and architecture models for heterogeneous network management.

2.1. Design Strategies for Integrating Heterogeneous Networks

Internets contain different network architectures or protocol stacks, like TCP/IP, OSI, DECnet and SNA. The combination of these different protocol stacks in a heterogeneous network demand the network management application to be able to solve translation problems between different network architectures and protocol implementations. To address the heterogeneous aspects of network management, the following design issues have to be considered:

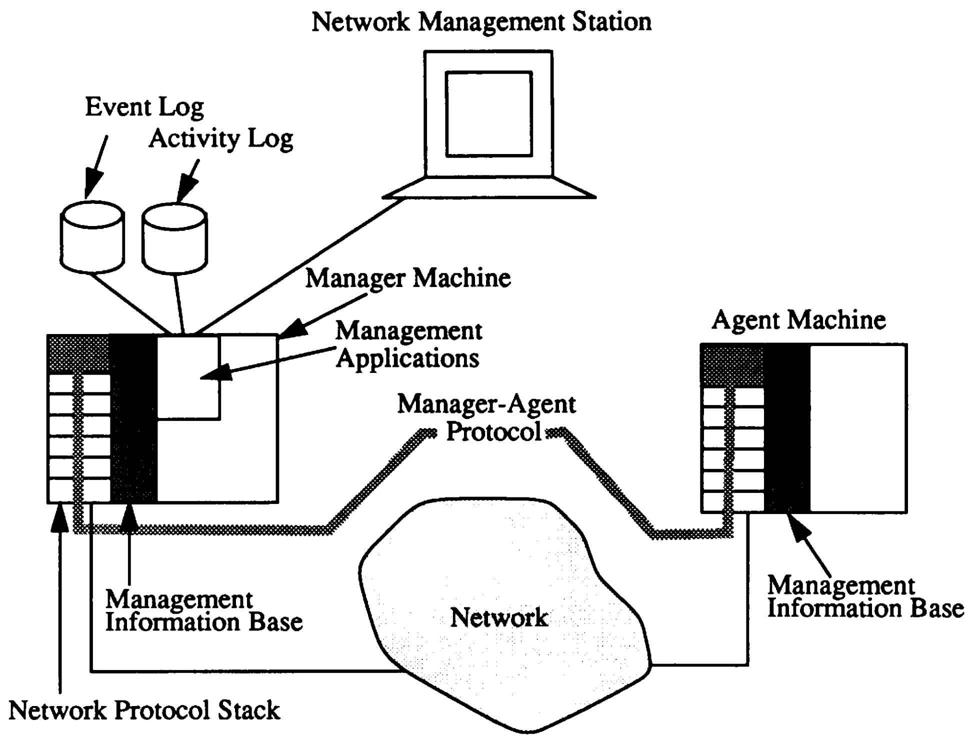


Figure 1

- **Integration of existing tools.** In a well-established internet a variety of management tools from different vendors are already available, designed for proprietary use and offer different network management functionality. The coordination and integration ("value-added network management") require special consideration for integration of external and internal data representation and consideration for the usage of these tools in a unified application.
- **Unified and consistent user interface.** Integration of network management tools is only helpful to the user if the network management system can unify these applications through a flexible and consistent user interface. This implies that the application tools have to provide means to dynamically add new object representations and operations to manage them.
- **Coexistence of different network management protocols.** As long as vendors offer different network management protocols, it is necessary to provide means for interoperability between these protocols. Involved in such an integration are issues like protocol translation, filtering of requests, maintaining state information, translation of syntax and semantics of available management information, and expanding the existing Management Information Base.
- **Interoperability between different protocol stacks.** The coordination of different protocol stacks requires a high degree of flexibility within the network management system. Different conventions and methods of addressing network objects and integration of protocol layers with different or even overlapping functionality have to be resolved. For example, if two protocol stacks reside on the same host and use an Ethernet, alterations of the Ethernet counters by one protocol stack will present a problem.
- **Flexible management of network objects with different capabilities.** Objects of similar functionality do not necessarily have identical characteristics. For example, some objects have additional vendor specific data structures. The network management system has to be able to adapt to those differences. Managing objects from other protocol stacks dramatically increases the variety of properties and data representations.
- **Notification of unresolved problems to the user.** The management station must report any mismatches or inconsistencies resulting from integrating heterogeneous networks into one network management application. Therefore event and error reporting tools have to be general and extensible.

- **Easy-to-use agent development tools and generic interface libraries.** Development tools should be easy to use. Then the developer can spend the implementation time on the issues specific to enhancements.
- **Well structured and flexible information base for data collection, resolution of statistics.** An information base that can be modified dynamically makes it possible to integrate different managed objects in a heterogeneous environment without interrupting the network service.

Rigid "turn-key" solutions for network management are not capable of achieving all of the above issues. Only the architectural concept of a network management platform that addresses generic behavior of services and provides a high level of extensibility will meet the needs required to administer heterogeneous networks.

2.2. Choosing an Architecture Model

As noted above, a heterogeneous network runs different protocol stacks, uses different network management protocols and supports different Management Information Bases. Two design alternatives [War89a] for integrating different protocol stacks exist. They are:

2.2.1. Centralized Manager

The management functions are centralized at one single management station, meaning that the manager station will run *multiple protocol stacks* in one physical host with several logical managers. This approach opens up the possibility to control interactions of different stacks in an effective way, since all the necessary translations and mappings happen on one designated host. Drawbacks of this design are:

- a centralized management station has to be able to run a significant number of protocol stacks. Even if it is possible to install all different stacks on one host, the management station becomes a bottleneck and time critical fault reporting and other events may be lost or delayed.
- licensing strategies of certain protocol stacks limit the number of possible installations; and therefore, makes it more expensive in a production environment to install additional protocol stacks on one host dedicated to management only.
- devices with no direct network address cannot be queried by the central manager (a local agent is required).
- in departmental size networks it cannot be assumed that one host can be fully dedicated to a network management station in the way the centralized manager approach requires.

2.2.2. Network Management Gateways

A management station implements a *primary* protocol stack. Management of other (*secondary*) stacks is delegated to other hosts running those stacks. These hosts — usually already gateways — function as network management gateways. An agent as the actual implementation of a network management gateway is called proxy agent. The manager communicates to the secondary protocol stacks through this designated gateway running one or more proxy agents (see figure 2). Drawbacks of this design are:

- compared to the solution of the centralized manger station, a request addressed to a device in another protocol stack has to be sent to the gateway first and will be dispatched from there. That can result in additional network traffic and response time delays.
- interdependencies between protocol stacks are more complicated to manage than in the centralized solution.

However, this approach has the following advantages:

- functionality can be delegated dynamically to other machines. For example, if the network management gateway itself has problems or if a reconfiguration is necessary, the proxy agent can be loaded on another host.
- new protocol stacks can be added or reconfigured transparently without affecting the availability of the management station.

Sun's network management system uses the second approach, since it fits most naturally into Sun's view of a network as a decentralized and frequently changing environment of distributed systems with autonomous administration needs.

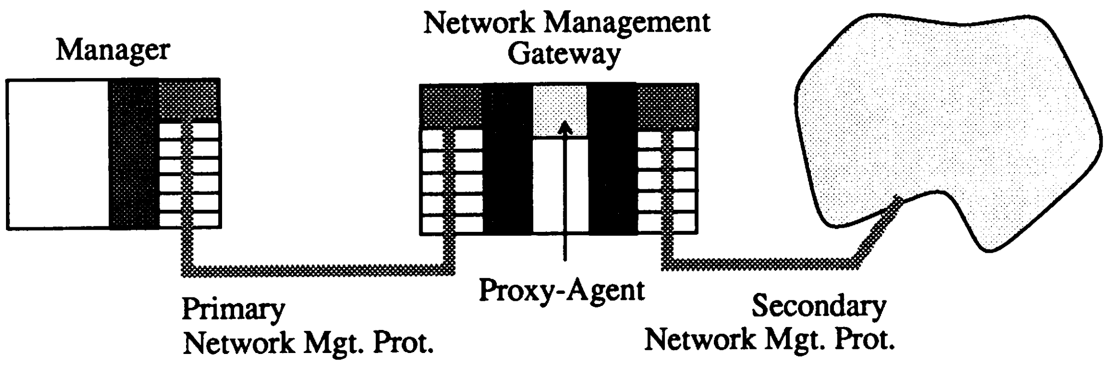

Manager
Network Management Gateway
Proxy-Agent
Primary Network Mgt. Prot.
Secondary Network Mgt. Prot.

Figure 2

3. Different Network Management Solutions

One aspect of integrating heterogeneous networks in a network management application is the integration of the different management protocols running on the different architectures. To illustrate the full range of interoperability issues, three network management designs are discussed. Simple Network Management Protocol (SNMP) is considered to be the management protocol standard for TCP/IP networks. It is a protocol that implements management through manipulating attributes in a Management Information Base. Network Information and Control Exchange (NICE), is a vendor-specific protocol used to manage DECnet networks where the protocol primitives support the architecture of a command language interface.

Sun's SunNet Manager (SNM) is designed as a protocol-independent network management platform to integrate existing applications and protocols, like NICE and SNMP.

Aspects, such as design goals, protocol services or interface primitives, form and scope of exchanged management data will be discussed. The different designs are introduced in the context of integration and translation issues that have to be resolved when implementing a network management gateway.

3.1. The Network Management Protocols SNMP and NICE

3.1.1. Simple Network Management Protocol (SNMP)

In March 1988 the Internet Activities Board (IAB) adopted SNMP as a standard to manage TCP/IP networks. Three RFCs describe the protocol and the Management Information Base [Casa, 1066a] and [1065a]. The philosophy and design goals of SNMP [Cas89a] state:

- the protocol design is kept simple to minimize cost and time of implementation.
- the number of agents is significantly higher than the number of management stations. Most of the data processing should occur on the management station.
- the performance impact and memory usage of the agent is minimal.
- the protocol design is extensible to allow additional variables, vendor specific variables, expansion for unilateral agreements and unanticipated needs.

All actions involve reading or writing of variables/attributes and there are no imperative command services. The datagram protocol UDP is used as the transport to avoid uncontrolled retransmissions so that it will keep working in broken networks. For example, in networks where acknowledgments of data packets are getting lost. The management station remains in control of network management traffic overhead. The number of unsolicited message types (traps or events, such as a state change of a host) is limited. Traps may be used to guide the timing and flow of polling. Service primitives have an associated request identifier and can encapsulate multiple requests. The SNMP service primitives are:

- *getRequest* and *getNextRequest* — request one or a list of data items and are responded to by a *getResponse*
- *getResponse* — responds to a request
- *setRequest* — changes a variable and the response is a *getResponse*
- *trap* — reports events

The requests are interpreted atomically. This means, if an error occurs on the responding side, the complete request will be terminated as an error request.

The syntax of the Management Information Base is a well defined subset of the syntax described by ASN.1 (Abstract Syntax Notation). It describes objects and their attributes which are ordered in a numbered tree. Objects and attributes can be added to the tree dynamically.

3.1.2. Network Information and Control Exchange (NICE)

The DECnet network management protocol NICE defined in DECnet Phase IV [DECa] can be characterized as a command-oriented protocol for managing DECnet networks only. It supports the following functionality:

- configuration of resources
- setting of parameters and variables for objects like hosts and communication lines
- examining of parameters, status of network and performance measurement
- initiating and terminating network functions (such as turning on/off DECnet on a host)

NICE is a command/response protocol using a reliable transport (NSP). Service primitives are:

• "request down-line load"	— loads software to a target host
• "request up-line dump"	— allows a host to dump its memory into a file on another host
• "trigger bootstrap"	— forces a boot on a remote host
• "test"	— tests local and remote availability of network elements
• "change parameters"	— modifies local or remote variables
• "read information"	— reads local or remote variables
• "zero counters"	— initializes variables
• "system-specific functions"	— are operating system specific messages
• "event reporting"	— reports events

The responses return status information of success or failure and any associated data if requested in the original command. Self-describing, vendor-specific encoding is used for counters and parameters of hosts, communication lines, and events. The Management Information Base is well defined, has a flat structure, and is not intended to change. Devices do not have to support all data requests.

3.2. SunNet Manager — A Network Management Platform

SunNet Manager (SNM) is a *network management* platform designed to run on Sun workstations managing TCP/IP networks and supports the integration of different network management protocols. One of the major design goals is to allow the user to customize available operations. SunNet Manager offers a set of integrated tools; some of which the end user might already be familiar with as stand-alone applications, like *traffic* or *netstat*. The user can also add functionality in two ways:

- an API for both manager and agent services allows developers to add new features. The agent service library implements a Remote Procedure Call (RPC)-based protocol over UDP and TCP.
- the Management Information Base is open for dynamic modifications. The network administrator can create data structures for new network objects or modify existing structures to adapt the tools to specific views of the network.

In the types of networks where Sun systems are usually installed, it cannot be assumed that a host will be solely designated to network management. Since SunNet Manager uses remote procedure calls, features like event, activity or general data logging can be distributed to other hosts within the network. In addition, some of the network management functionality is distributed to hosts running agents. The agent has to be able to respond to the following message types:

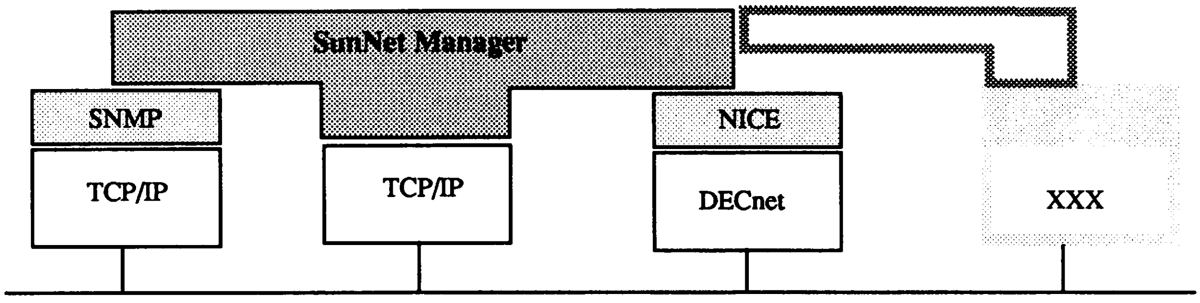


Figure 3

- *start_event_reporting* message is used for setting of thresholds on the agent and can be used to define some conditions for fault notification, and configuration changes. No responses are generated until the threshold conditions are met.
- *start_data_reporting* message from the manager may cause many responses. For example, to compute a statistic over a period of time, continuous polling will be triggered with one request. The manager starts the request, specifying the interval and duration of the polling. From that point on, the agent gains control and is expected to send the responses back to a synchronization point specified by the original request. This design cuts the network traffic back considerably, compared to a polling process implemented with SNMP or NICE.

Developing an agent for SNM does not mean implementing a protocol. SNM provides a library that defines an agent Application Programming Interface (API), and it assists the developer to write agents. To respond to the above message types, the agent calls service primitives from the agent library.

The Management Information Base has three interfaces:

- *schema and component files* on the management station are used to define the topology of the managed network, to build requests, and map incoming data responses to the outstanding requests.
- the *agent specific* internal representation of the Management Information Base.
- the agent response mechanism uses *quadruples* of name, type, size, and value as the encoding of the attributes describing managed objects.

SunNet Manager is not command/response oriented. The manager issues requests that manipulate the equivalent representation of the Management Information Base at the agent. In this respect, an integration of networks managed with SNMP is relatively straightforward. The main tasks involve mapping the protocol service primitives to library functions and interpreting the SNMP related Management Information Base. On the other hand, the integration of NICE, a command oriented protocol, involves more complex issues in mapping protocol service primitives and translating management information.

4. Implementing Network Management Gateways

Each vendor supplies their own management tools, their own components and component structures, syntax, semantics, protocols and user interfaces. The goal is to integrate these different network management approaches and to unify network management facilities in a generic platform. The concept of a network management platform offers extensibility to third parties through well-known, publicly available interfaces and protocols. It allows the system administrator to install or customize views of the networks that suit organizational requirements and preferences. In the following chapter, implementation issues for network management gateways will be discussed and illustrated in terms of how SNM integrates SNMP and NICE.

4.1. Creating the Management Information Base

The Management Information Base has two main functions:

- providing the data structures that describe the managed object in terms of the attributes of interest (such as counters, events, etc.).

- providing the information needed for representing managed objects at the management station. That includes all the data structures needed to create the topological view of the network to be administered, like addresses, names, owners of machines and responsible administrators. It also includes the information needed to formulate requests for a managed object.

The usage of this information is often overlapping. For example data from the topology information is used as data in requests for management information, or events may be used to change the state information in the topology display.

4.1.1. The Management Information Base as a Means to Communicate with the Proxy Agent

In order to allow the Management Information Base to handle a newly integrated network architecture on the management station, new objects and their attributes must be added. An example is the integration of NICE specific managed objects like that of a DEC host. The NICE request for host information is designed as a generic request for all known attributes of that host defined by NICE, like "Identification", "Maximum Links" and "Routing Version," etc. The request asks for the whole group of attributes, not for each single attribute. The response is a message containing the names, data types and values of *all available* attributes. In the SNM architecture, a new schema file contains all NICE managed objects. The attributes are grouped so that a request for the DECnet proxy agent can be built and so that the responses can be interpreted. Here is an excerpt of the schema file for the DECnet proxy agent:

```
proxy proxydni
        description "proxy dni agent"
        map "_targetsystem=DECnet_Hostname"
(
       group host_chars
           description "Host Characteristics"
        (  string[12]    Ident
                 description "Identification"
           unsigned int  Max_Links
                 description "Maximum Links"
           ...
           string[20]    Rout_Vers
                 description "Routing Version"
        )
        group host_cnts
           description "Host Counters"
           ... and a set of attributes like "User Bytes Received", "User Bytes Sent"
        ...
)
```

The schema file is used for mapping the Management Information Base of NICE into a format that the SNM management applications can interpret. In order to match both Management Information Bases it is necessary to structure the data of the DECnet network architecture according to a format useful for the Management Information Base of SNM.

A proxy agent in general has to translate differences in the syntax of the primary Management Information Base and the data structures used in the secondary network management protocol. This involves:

- providing translations between different data types. A special issue is the handling of tables, like the array of all installed network interfaces on one machine. SNMP, for example, uses the special operation getNextRequest to walk through the table entries, NICE sends a message per table entry, whereas SNM expects the full table in one response message
- mapping the request syntax of the primary and secondary protocol
- recognizing whether the managed object has the requested attributes
- informing the manager station of any mismatch between requested and available attributes

4.1.2. Supporting a New Network Architecture on the Management Station

Managed Objects from a different network architecture are added to the existing view of the network. They have different characteristics, and therefore they must be specified in the Management Information Base on the management station. With SNM one can define new components for networks and devices. Consider the following two DECnet specific structures:

A VAX running DECnet:

```
record component.vax (
        string[32]      Host_Name
        string[40]      Host_Owner
        string[40]      DECnet_Hostname
        string[32]      Administrator
        string[80]      Notes_Comments
)
```

A DECnet network component can be declared as:

```
record view.decnet(
        string[32]      Network_Name
        string[40]      DECnet_Area_Number
        string[40]      DECnet_Hostname
        string[16]      Phone_Contact
        string[80]      Notes_Comments
)
```

The DECnet host name DECnet_Hostname has to be included in the DECnet network component definition to specify the network management gateway that is responsible for this network. All management requests regarding a network or a host use the address specified in the string DECnet_Hostname. The proxy agent does not have any knowledge about the above described data structures. These component structures are necessary for the management applications to be able use the correct addressing domain.

4.1.3. Representation of a Gateway on the Management Station

The network management gateway is usually installed where there is already a gateway between the two architectures. Possibly several other gateways of the same type are on the network. Therefore, the system administrator wants to distinguish these machines in a special way in the topology of the network. It is necessary to show what protocol stacks are installed on each gateway. Since the number of protocol stacks on one machine can change over time, it must be easy to reconfigure a gateway component.

SNM offers two alternatives representing a gateway. First, the gateway can be represented as one component. If another protocol stack is installed on that machine, the component can be renamed and attributes can be added without impacting the availability of the already defined protocol stack. Consider the following example of a Sun host that has DECnet installed:

```
record component.dnisun (
        string[32]      Host_Name
        string[40]      Host_Owner
        string[40]      IP_Address
        string[40]      DECnet_Hostname
        string[10]      Version
        string[80]      Notes
)
```

Or conceptually the component may be broken up in a set of smaller subcomponents in this manner:

```
record component.sun (
        string[32]      Host_Name
        string[40]      Host_Owner
        string[40]      IP_Address
        string[80]      Notes
)
```

and

```
record component.dni (
        string[32]      Host_Name
        string[40]      DECnet_Hostname
        string[10]      Version
)
```

4.1.4. Version Control of the Management Information Base

Many Management Information Bases are extensible; that is, new objects and attributes can be added to them. The mechanisms of changing a Management Information Base varies. For example, the supplier of enhancements to the Internet Management Information Base related to SNMP formally registers with a committee that has to approve modifications. SNM allows dynamic changes anytime, as long as the proxy agent can dynamically adjust. Changes of the Management Information Base of the secondary protocol have to be synchronized with the Management Information Base of the management station. Usually conversion utilities can be provided for integrating and updating the Management Information Base on the management station. If version control mechanisms for an information base are available, they are usually implemented very differently. Some protocols integrate the version number into the object name, SNM uses the RPC version control mechanism. The proxy agent is responsible for mapping the different version control mechanisms. Not only the Management Information Base versions have to be coordinated. With changes in the Management Information Base usually different versions of proxy agents have to be installed in order to provide the correctly matching information. For the first release of SNM the RPC program number is used to coordinate proxy agent and Management Information Base versions with the management station.

4.2. Using the Management Information Base to Request Information

The previous section illustrated the different aspects of a Management Information Base. This section discusses how the Management Information Base is used to actually form requests.

4.2.1. The System Administrator's View of a Managed Object in the "Foreign" Architecture

Data structures in the Management Information Base of the management station have to be provided to send the request to the right host — the network management gateway. SNM solves this problem in the following way. Each managed object is defined as an instance of a special component type (see sections 4.1.2. and 4.1.3.). The actual object contains both the component type and a list of other characteristics. All the agents that monitor this object are included in this list of characteristics. The following defines a VAX named "munich" after DECnet has been installed, and it is monitored by a network management gateway host called "DECnetGateway".

```
cluster(
    component.vax ( munich "Karl Valentin" 8.27 ... "DECnet installed")
    ...
    proxy ( proxydni DECnetGateway )
)
```

The host "munich" knows only about the DECnet proxy agent "proxydni". The proxy agent declaration contains one additional parameter to define where the proxy agent is actually running; in this example it resides on Sun host "DECnetGateway".

When a system administrator on the SNM management station directs a management query to a device in the DECnet subnet, it appears to the user that the request is handled by the addressed device. However the request is transparently redirected to the proxy agent for processing. The user will not realize that all received responses come actually from the proxy agent running at the Sun network management gateway, and not from host "munich".

4.2.2. Addressing Managed Objects at the Network Management Gateway

In the native network architecture, managed objects can be addressed in native addressing mode. For example, a monitoring request for a host in a TCP/IP network uses the internet (IP) address to communicate. A DECnet host cannot be accessed by specifying the IP address. Therefore the request will be sent first to the proxy agent on the network management gateway. The gateway itself is addressable in native mode, but it knows the managed object in the other addressing domain. The primary network management protocol has to provide a mechanism to specify an address in the format of the second addressing domain. Some protocols solve the problem by introducing special hierarchical structures in the Management Information Base. SNM provides a designated field of generic type in each request message to specify the address. This address is defined in the component definition, such as DECnet_Hostname (see section 4.1.2.). The proxy agent uses a valid DECnet address to dispatch the request to the right managed object.

4.2.3. Managing Subnets

There are requests in some network management protocols that are applicable to a network instead of a single device. An example is the NICE command primitive "show known hosts" in a DECnet network. Since the proxy agent implements the NICE protocol, the request can be answered by the network management gateway. However, it does not fit into the SNM concept requesting data on a per device basis. Therefore, SNM allows that the proxy agent can also be associated with a DECnet *network component* (see example in section 4.1.2.). A different schema file is used to define the attributes to build requests for a whole subnet.

4.3. Implementation of Proxy Agents

This chapter covers issues concerning protocol translation and other implementation issues. Each protocol has a set of service primitives that have to be translated. In some cases there will be a close match of functionality, in other cases a creative interpretation has to be devised.

4.3.1. Translation of Provided Service Primitives

A network management gateway receives many requests for different managed objects. The proxy agent has to administer the name space of objects, multiplex the requests, keep track of the outstanding responses and send the responses back in the way the manager station expects the responses, opposed to the way the secondary protocol provides the data. When a proxy agent receives a network management request, it may have the data to respond immediately or it may have to query the target system. In either event, the mechanism for responding is under the control of the proxy agent. If SNM is the primary protocol, the proxy agent has to translate one SNM request into a number of requests for NICE or SNMP.

SNM, SNMP and NICE have the functionality of retrieving data, "read counters" for NICE, "start_data_reporting" in SNM, and "getRequest" in SNMP. However, the provided parameters with each of those requests are different. SNMP allows a list of attributes to be requested in one message, but SNM and NICE only request groups of well defined attributes and possibly expect multiple responses containing the values for each attributes, but in slightly different senses.

The proxy agents has to match requests like:

- an SNM request:

```
request(type, host, group, key, count, interval, options)
  unsigned int type;            /* request type, like
                                   start_data_reporting or
                                   start_event_reporting*/
  char *host;                   /* target host name */
  char *group;                  /* object group name */
  char *key;                    /* object group key */
  unsigned int count;           /* number of times to sample*/
  struct timeval interval;      /* sampling interval */
  unsigned int options;         /* request options */
```

- a NICE request:

```
request(type, object)
  unsigned short type;          /* request type as in one of the service
                                   primitives defined in chapter 3.1.2, like
                                   "read counters", "down-line load" */
  struct DECnet_address object; /* name and address of object */
```

The mapping between SNM host and NICE object, SNM type and NICE type is straightforward, but the other parameters such as key, count, etc. have no equivalent. For example, the proxy agent will have to make multiple secondary requests as defined in the count and interval parameters in order to gather the data and interpret it to furnish the required SNM responses.

4.3.2. Creating Mappings for Unmatched Protocol Service Primitives

There may be service primitives that are not supported by the primary network management protocol. For example, NICE supports the command "load program", that is inappropriate for the SNM architecture. Therefore SNM provides remote login to the VAX, as an alternative, giving the users the opportunity to execute this command on the machine directly, if they wish. This is implemented through SNM's ability to associate remote login commands with objects. For the "vax" and "dnisun" components, as defined in section 4.1, a set of commands can be installed:

```
instance componentCommand (
   (component.vax   "DECnet Control" "windowtool dnilogin %DECnet_Hostname")
   (component.sun   "TCP/IP Control" "windowtool rlogin %Sun_Hostname" )
)
```

The first line allows automatic login to the DECnet node using the remote login facility provided by the DECnet implementation on a Sun (dnilogin). The second line shows the equivalent remote login for a Sun host (rlogin).

4.3.3. Overlapping Functionality Between Network Management Protocols

The lower protocol layers have common attributes. For example, Ethernet statistics may be viewed from a TCP/IP Management Information Base or viewed from the DECnet Management Information Base. It depends on the network management application whether to consolidate the data or to keep the data structures separate and leave it to the system administrator to interpret it. The advantage of consolidating such information is that the user has a coherent view of the network and can change values consistently if necessary. However, keeping the data separate could prove useful in debugging interoperability problems between protocol stacks. Usually the user interface of the management station defines, how these attributes have to be viewed. The SNM user interface keeps these attributes separated.

4.3.4. Coordination of flow control

The manager/proxy agent communication assumes a certain, well defined flow of control. The flow of control of the secondary network management protocol has to be integrated into the communication flow of the primary protocol. For example, SNM defines the agent process through the modules "agent initialization", "request verification", "request dispatching" and "request handling". Even if the secondary management protocol does not use these modules, the proxy agent has to fill in functionality like "request verification" (to assure the integrity of the request). Factors such as time-out values of a request have to be taken into consideration when integrating additional modules.

In this context it is also necessary to mention the provided quality of service. If the primary protocol uses reliable service, the secondary protocol should also try to ensure reliability. That might require the implementation of additional retransmission algorithms and confirmation procedures.

4.3.5. Mapping of Authentication Schemes

Since certain management operations have far-reaching effects on the network or device, it is often desirable to guarantee reasonable security of those operations. The proxy agent would then have the responsibility of providing the required management information without violating the security requirements of the network. Since the authentication schemes are usually not easy to integrate, this is a very problematic issue. A satisfactory compromise seems to be to at least ensure the usage of both native authentication schemes. For example, SNM uses the mechanisms of secure RPC, whereas NICE uses user-password identification. To guarantee the security of a request beyond the direct SNM environment, a general purpose field in the structure of an SNM request is used to encode the NICE user identification, necessary to execute certain NICE protocol services. The decoding and parsing of that field is the proxy agent's responsibility.

4.3.6. Error Handling

The error handling is different from protocol to protocol in the following respects:

- definition of what an error situation is
- data structures of errors
- handling of error situations or error notification

Roughly two groups of error situations can be observed:

- errors caused by availability problems of the network management gateway and/or the managed object
- errors due to problems that are request specific

Errors related to the first category have to be handled through event reporting mechanisms. Errors in the second category affect usually one or a sequence of requests. SNM has a verification phase that allows for a basic integrity check for each request. If the proxy agent discovers errors *after* the verification phase, an error message can be sent back to the management station. That message will be displayed on the management station. Additionally, an SNM request contains a flag indicating whether to handle the request as "atomic" or "best effort". If an error occurs in an atomic request, the whole message will be discarded. With "best effort", the management station expects that other data items may still be reported.

5. Conclusion

Today's network management applications must address the issues involved in the joining of heterogeneous networks. It is a complex problem that requires careful consideration in the early design phase of a network management product. The network platform approach that distributes heterogeneous network management among master stations and network management gateways provides a flexible and extensible architecture to implement network management functions in dynamically changing environments. The experience gathered with the SunNet Manager product proves the feasibility of this approach towards network management.

References

[ISOa] ISO DIS 7498-4, *Basic Reference Model: Management Framework'*.

[Bru89a] P. J. Brusil and D. P. Stokesberry, "Integrated Network Management," *Proceedings of the IFIP TC 6/WG 6.6 Symposium on Integrated Network Management*, May 1989.

[Cas89a] J. D. Case, "Network Management of TCP/IP-Based Internets," *Internetworking Tutorials, Advanced Computing Environments*, April 1989.

[Casa] J. D. Case, M. Fedor, M. Schoffstall, and J. Davin, "A Simple Network Management Protocol," *Internet Request for Comments RFC 1098.*

[DECa] Digital Equipment Corp, *DECnet Digital Network Architecture Phase IV — Network Management Functional Specification*, AA-X4374-TK.

[1066a] K. McCloghrie and M. T. Rose, "Management Information Base for Network Management of TCP/IP-based Internets," *Internet Request for Comments RFC 1066.*

[1065a] M. T. Rose and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-based Internets," *Internet Request for Comments RFC 1065.*

[War89a] U. S. Warrier and C. A. Sunshine, "A Platform For Heterogeneous Interconnection Network Management," *Proceedings of the IFIP TC 6/WG 6.6 Symposium on Integrated Network Management*, May 1989.

An introduction to OSI architecture, concepts and terminology

John Henshall

EUCS
University of Edinburgh
Edinburgh
Scotland
J.Henshall@ed.ac.uk

ABSTRACT

This paper will take the form of a tutorial slide presentation as an introduction to the application orientated layers of the ISO reference model. It is aimed at the level of "no knowledge" of OSI and goes from there to a point where the listener should have a reasonable grasp of the basics of design and terminology of OSI. It is not a "justifying" paper – neither knocking nor glorifying any particular communications technique – it is a simple tutorial. There are 30+ slides hopefully providing an ideal "starter" for OSI communications literacy both for those who wish to join the ranks of its supporters, or to have more buzz words to use in the fight against its rather "top heavy" design.

PCSERVE: An Attempt To Integrate PC Users Into The UNIX Community

Toshiharu Harada

Takehiko Nishiyama

Hidekazu Enjo

NTT Data Communications Systems Corporation
Kowa Kawasaki Nishi-guchi Bldg.,
66–2 Horikawa-cho, Saiwai-ku, Kawasaki-shi,
Kanagawa 210, Japan

harada@rd.nttdata.jp

ABSTRACT

PCSERVE is a system which integrates PC (hereafter, PC stands for a DOS based Personal Computer) users into the UNIX environment. Using PCSERVE, PC users are able to access the UNIX community without a knowledge of UNIX.

The system is based on the server-agent-client model, which is an enhanced version of the server-client model, by introducing an agent between a client and a server. The agent is placed on a UNIX host and provides methods to invoke and utilize anonymous stream type internet services in UNIX hosts according to the requests from sophisticated applications on PC's.

This paper describes the design issues and their consequences. With the introduction of the agent, PCSERVE achieved a higher level of flexibility and solved the existing problems of the server-client model between PC's and UNIX hosts.

1. Introduction

To integrate novice PC users into the UNIX environment, an elementary knowledge of UNIX should not be required. Moreover, it is desirable to facilitate user interface that is familiar with PC users. For example, most PC users will prefer editor programs of PC's than *vi* or *emacs*. These requirements have not been satisfactorily met, though there have been many attempts to integrate PC's into the UNIX environment.

Regarding electronic mail, the first attempt was to apply SMTP [Pos82a] (Simple Mail Transfer Protocol) between PC's and UNIX hosts. In that system, PC's are treated as independent hosts. This means that mailboxes for each user are placed in PC's. However, for a service, like electronic mail, it is desirable for data files to be stored in UNIX hosts, because it is hard to guarantee the security of mailboxes in PC's and PC users need to access their mailboxes from different PC's, such as PC's located in their homes or offices.

To solve the above problems the server-client model has been introduced (POP [Ros88a] and PCMAIL [Cla86a] are the examples). These attempts seem successful so far, but we think they are not good enough to integrate PC's into the UNIX environment for the following reasons. First of all, the flexibility of the system is not very high; in the system, applications on PC's are coupled directly with the services, so even slight changes in underlying protocols are rarely acceptable without modifying the applications on PC's. Secondly, the maintenance of the system is difficult, because some information about system administration (including the name of the service and the UNIX host) is still kept within the PC's. This is especially troublesome when the system has to deal with a huge number of PC's. In this case, who will be responsible for changing the applications on all these PC's and how can this change take place? If we introduce the agent program between applications on PC's and internet services in UNIX hosts, it acts as a mediator and is able to solve the above problems.

2. The PCSERVE System

2.1. System Organization

The PCSERVE system consists of the agent program called *pcserve* in UNIX hosts, client applications on PC's, server programs of anonymous internet services in UNIX hosts, and a pair of communication drivers for PC's and UNIX hosts. The organization of the system is shown in figure 1.

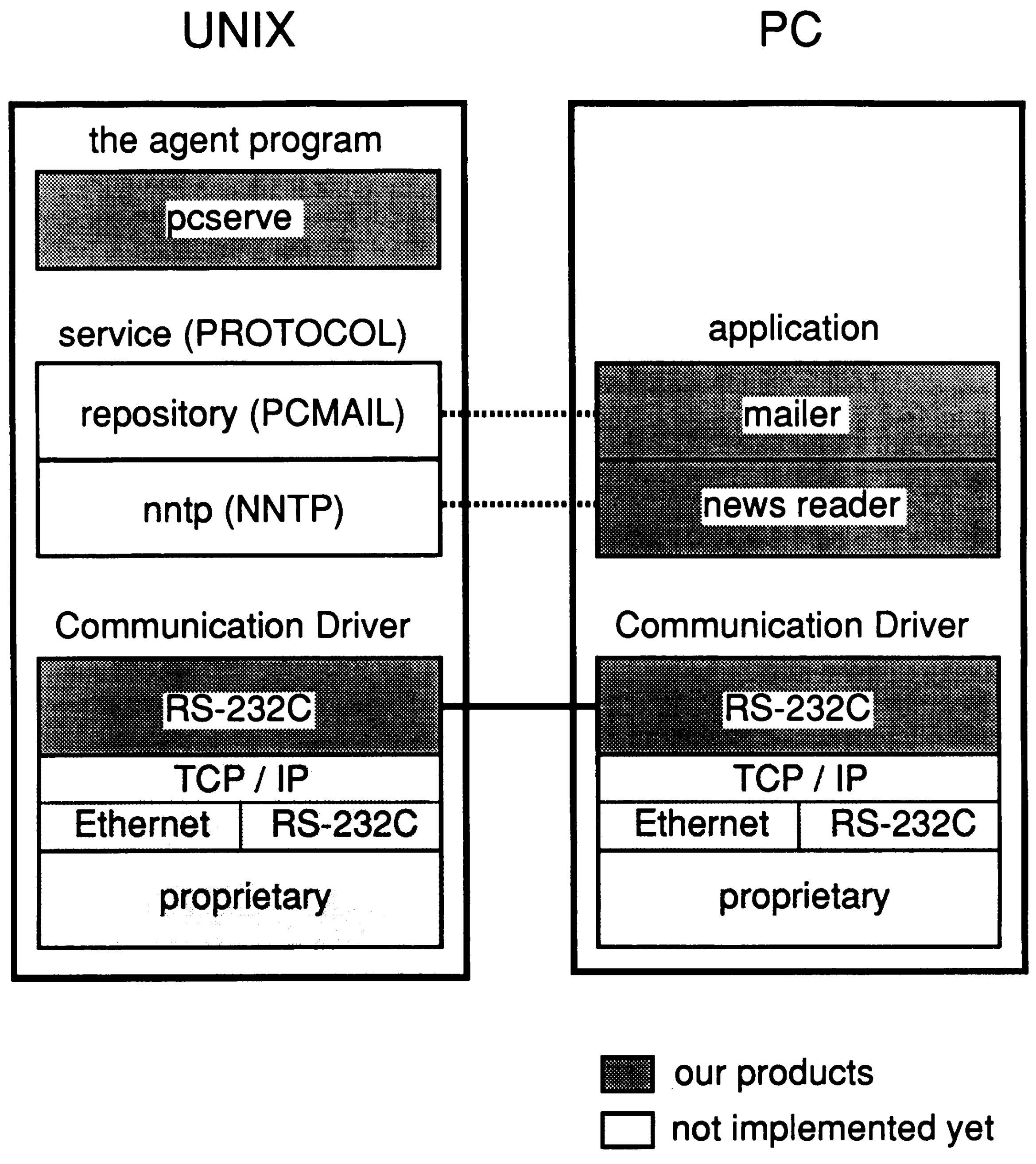


Figure 1: *System Organization*

In the PCSERVE system, client applications on PC's communicate with servers of internet services via the *pcserve* program; they do not talk directly with the servers in UNIX hosts. Physical connections are performed through a pair of communication drivers. The connection between client applications on PC's and the internet services in UNIX hosts is shown in figure 2.

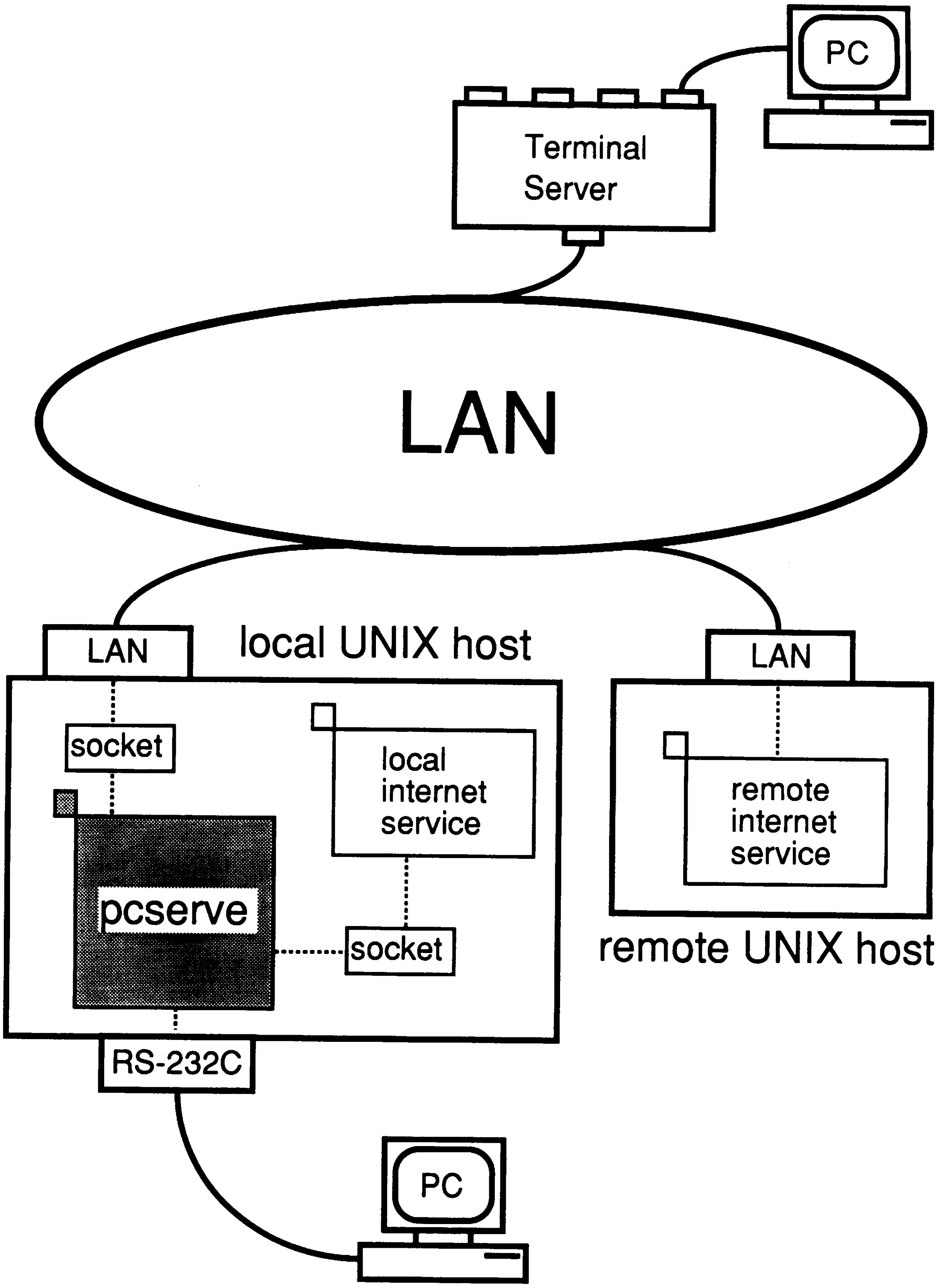


Figure 2: *Connection between PC's and UNIX Hosts*

2.2. Functions of PCSERVE

PCSERVE provides two major facilities: one is the service as a mediator and the other is fundamental networking procedures.

As a mediator, it provides the following jobs.

- Choosing an internet service
- Choosing a UNIX host of an internet service
- Invoking an internet service
- Passing a line of data to the internet service
- Reading a line of data from the internet service

For network procedures, it provides the following facilities.

- User authentication and logging
- File transfer between PC's and UNIX hosts
- Printing files in UNIX hosts

3. Features

By introducing the server-agent-client model, several advantages are achieved.

The system becomes more flexible. Anonymous stream type internet services in UNIX hosts can be invoked and utilized by talking with *pcserve* from applications on PC's. Since *pcserve* is independent of the specific network protocol, the PCSERVE system is applicable to various kinds of network protocols, such as "Cheaper-LAN," without any change in client applications on PC's.

Another advantage is the service extension. Since commands to the internet services are once interpreted and delivered by *pcserve*, the functions of the service could be extended by adding new functions to the *pcserve* program instead of modifying internet services. PCSERVE is also able to absorb slight changes of internet service protocols.

The third advantage is multiple services. It is possible to enhance *pcserve* to handle multiple services at the same time. However, it is now implemented only to handle one service during the session. This feature might be a great advantage for services such as a distributed database.

The last advantage is the virtual service. The simplest way to use *pcserve* is to let it work as a truthful mediator between applications on PC's and servers of internet services. However, it is possible for *pcserve* to alter commands from PC's by referring to a correspondent table in the *pcserve* program. In this way, client applications on PC's can adopt a (simple) virtual server protocol. *pcserve* converts it into a specific server protocol in UNIX hosts for PC's. (See figure 3).

4. Design Issues

4.1. Connection between PC's and *pcserve*

To connect PC's and *pcserve*, three combinations are possible: TCP/IP on Ethernet, SLIP [Rom88a] on RS-232C, and packet on RS-232C. In our current implementation, the last one was chosen, because it is the most common and requires the least costs. With this approach, a client application on a PC must login to a UNIX host like an ordinary terminal session and invoke the *pcserve* program instead of UNIX shells. In the future, we are planning to support other communication methods.

4.2. User Authentication and Logging

PCSERVE supports user authentication protocol. If a PC user passed a proper pair of a logname and a corresponding password, the *pcserve* process will take the place of the specified user's UNIX process and get his privilege and access rights. Without passing the right pair, the PC user can not use any of the facilities which are related to the internet services.

In the PCSERVE system, the *pcserve* program deals with all requests from PC users. So it knows all the information needed to manage client applications on PC's and logs can be taken easily.

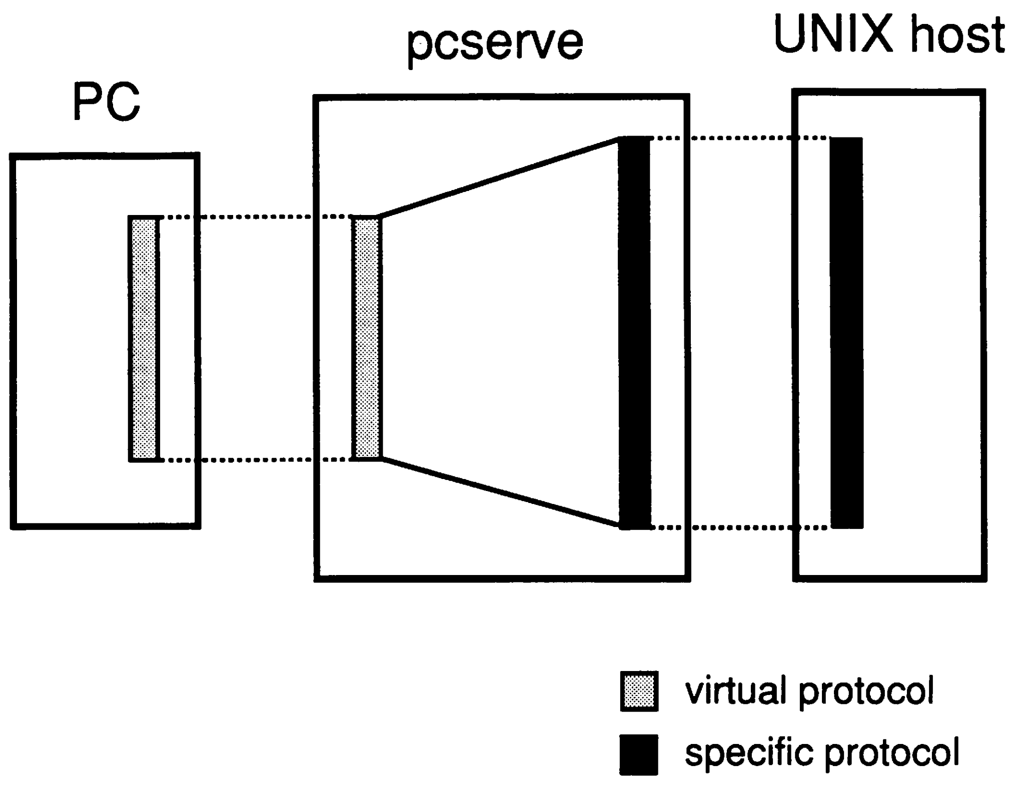


Figure 3: *Virtual Service*

4.3. The *pcserve* Program

4.3.1. Internal Variables

pcserve has three internal variables of *user*, *service*, and *server*. Values of these variables can be set and shown by passing a *set* command to *pcserve*. Defining *user* requires two arguments: a logname and the corresponding password. When they match to the password file entry of a UNIX host, the variable *user* will be defined to the specified logname. To invoke an internet service, all three variables must be defined beforehand.

4.3.2. Commands

Commands to *pcserve* are classified into two groups. That is, commands for the *pcserve* program itself and commands for the opened internet services. Commands for the *pcserve* program are shown in Table 1.

Client applications on PC's can tell *pcserve* that the command is for the internet service by putting a prefix to it. If *pcserve* receives such a command, it strips the prefix and passes the result to the internet service. The command prefixes are "*write*" and "*command*". The difference of those two prefixes is that the latter one includes one "*read*" command. It is designed so to save communications between PC's and UNIX hosts, because most internet services respond with a line for commands from their clients. The former prefix can be used for passing data to internet services. Commands for the internet services are shown in Table 2. Examples of command recognition are shown in figure 4.

A line is the unit for the commands, "*read*", "*write*", and "*command*". However, *pcserve* also supports the commands which manage multiple lines at a time for efficiency. For example, "*read text*" command would read lines until *pcserve* receives a line that includes only one period from the server of the internet service.

4.3.3. Command Response

pcserve responds to every command with a combination of three digits and a message. Client applications on PC's can know the status of its commands by examining the three digits. The meanings of the three digits are shown in Table 3.

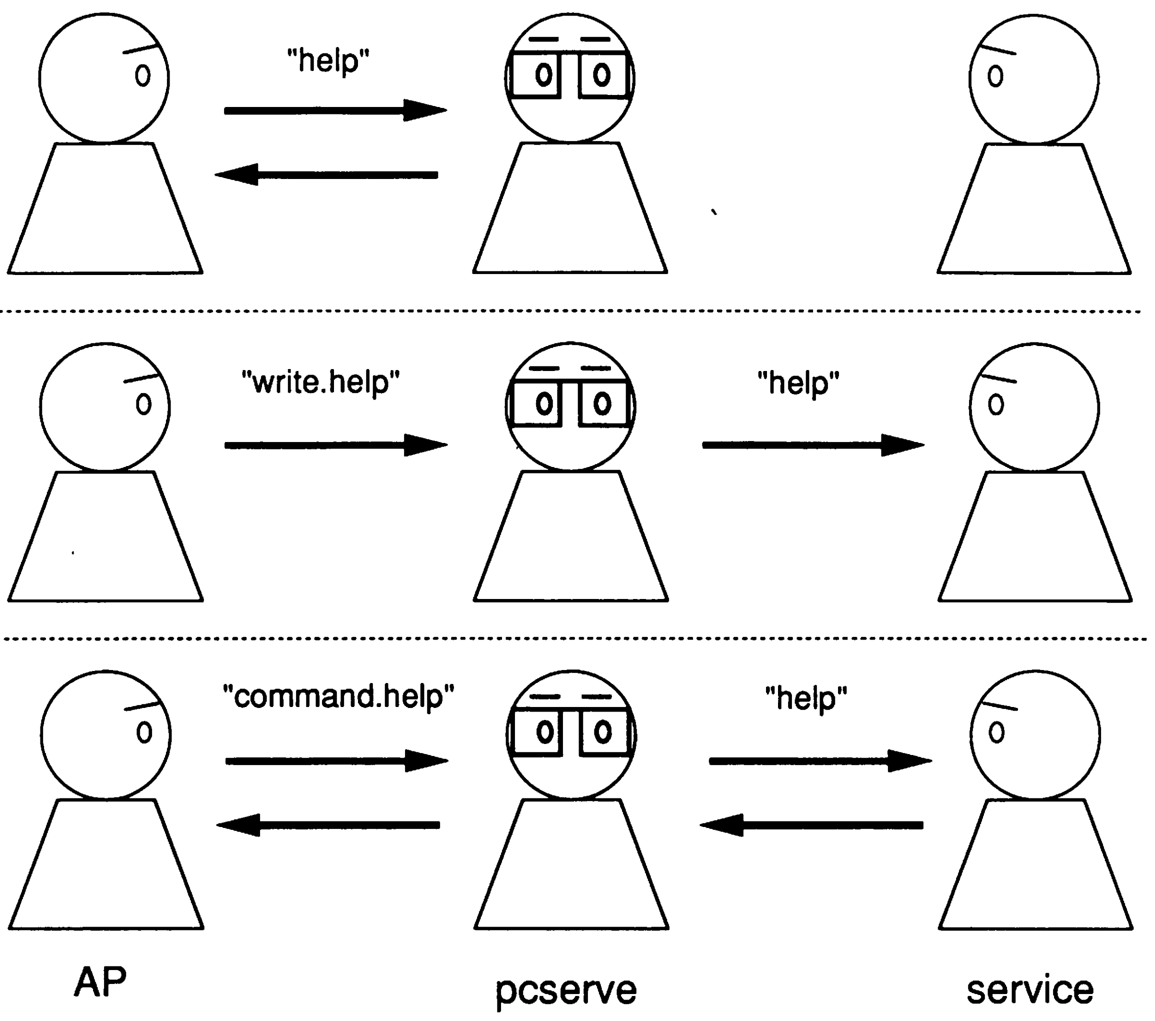


Figure 4: *Command Recognition*

Command [arg]	Function
set server [server name]	select a UNIX host which provides the internet service
set service [service name]	select an internet service
set user [logname] [password]	do user authentication
set	show current values of all variables
set [variable]	show current value of the variable
set [variable] [value]	set value to the variable
put [file]	transfer a file from PC's to UNIX hosts
get [file]	transfer a file from UNIX hosts to PC's
help	show help message
open	open the specified internet service

Table 1: *Commands for the pcserve Program*

Command	Function
read	read one line from the service
read text	read till a line with only one period is found from the service
write	pass one line to the service

Table 2: *Commands for Internet Services*

digits	meaning
700	command completed successfully
701	waiting to read
830	waiting input for the service
900	unknown commands
901	syntax error
910	variable *user* is not defined ("open" error)
911	variable *service* is not defined ("open" error)
912	variable *server* is not defined ("open" error)
921	unknown *service* ("open" error)
922	unknown *server* ("open" error)
930	file does not exist

Table 3: *Command Response*

5. Current Status and Future Plan

The PCSERVE system has been designed and implemented. The SUN3 workstation was used to act as a UNIX host, and Japanese PC's were used as regular PC's.

For the internet services, the repository (PCMAIL [Cla86a] and *nntp* (Network News Transfer Protocol [Kan86a]) were chosen, because electronic mail and the news system are the most fundamental networking facilities and are in great demand by many PC users.

The *pcserve* program was written on the SUN3 workstation. A mail reader and a news reader programs were used as client applications on PC's and are products of the PCSERVE system.

Performance of the PCSERVE system is being investigated. It is easy to adapt our system to another internet service such as network-wide nameserver. Our future plans for the PCSERVE system are enhancement of *pcserve* facilities, introducing other internet services, adaptation for distributed database services, and introducing a procedure control language.

6. Conclusion

Some systems based on the server-client model have been introduced to integrate PC's into the UNIX environment. They are successful, but some problems in flexibility still remain.

To solve the problems, we introduced the server-agent-client model which was enhanced by introducing an agent program between a server and a client. Thanks to the agent, the system became more flexible and easy to manage. Furthermore, it shows high possibility of extension. In this paper, the design of our system and its features were described. A prototype system was produced and is being investigated for basic networking services of electronic mail and the news system.

7. Acknowledgements

Current implementation of the PCSERVE system uses pcmail [Cla86a] repository and *nntp* [Kan86a] as servers of the UNIX services. We would like to express our gratitude to the authors of them.

Special thanks to Mr. Kouichi Mochigase, Mr. Shigefumi Takahashi, and Miss. Yasuko Kumai for writing a mailer and a newsreader program on PC's. Also many thanks to Mr. Takeshi Tanaka for his assistance in writing the *pcserve* program on the SUN3 workstation.

Thanks are also due to Mr. Yoshiaki Tanaka, Mr. Takashi Katsukawa, Mr. Koji Abe, Mr. Hirotaka Fuchizawa, and Mr. Shin-ichi Yamada for their encouragement.

References

[Cla86a] David D. Clark and Mark L. Lambert, "PCMAIL: A Distributed Mail System for Personal Computers," *RFC993*, Massachusetts Institute of Technology, December 1986.

[Kan86a] Brian Kantor and Phil Lapsley, "Network News Transfer Protocol," *RFC977*, U.C. San Diego and U.C Berkeley, February 1986.

[Pos82a] J.B. Postel, "Simple Mail Transfer Protocol," *RFC821*, August 1982.

[Rom88a] J. Romkey, "A Nonstandard For Transmission Of IP Datagrams Over Serial Lines: SLIP," *RFC1055*, June 1988.

[Ros88a] M. Rose, "Post Office Protocol — Version 3," *RFC1081*, TWG, November 1988.

Appendix: Example of a Session

The following is an example of a session that used *pcserve*. The whole lines were given as the result of the UNIX command *script*. Please note that a string "input>" was added to make the user's input distinguishable from its response by *pcserve*. Some comments were also added for readability.

```
$ pcserve
700 Pcserve service version 1.3 12 26 89 ready at Mon Jan  8 11:36:35 1990
input> help                                 # "pcserve" answered as follows:
701 extended commands list follow:
set:    set or show variables
open:   open a service
help:   show this message
get:    get a file from the server
put:    put a file to the server
quit:   exit pcserve
.
input> open
910 user undefined                          # "user" must be defined before open!
input> set user harada stormbringer         # "stormbringer" was my password
700 user set to harada
input> set                                  # see values of variables
701 current value of variables
user: harada
server:
service:
.
input> open                                 # "service" is not defined yet!
911 service undefined
input> set service nntp
700 service set to nntp
input> set server nttose                    # "nttose" is the name of the UNIX host
700 server set to nttose
input> set
701 current value of variables
user: harada
server: nttose
service: nntp
.
input> open                                 # now, we are ready to invoke the service
700 open nntp completed!
input> command.help                         # send "help" to "nntp" at "nttose"
201 nttose NNTP server version 1.5 (26 Feb 88) ready at Mon Jan  8 11:39:50 1990 (no posting).
input> read text                            # read till a line with only one period is found
100 This server accepts the following commands:
ARTICLE     BODY         GROUP
HEAD        LAST         LIST
NEXT        POST         QUIT
STAT        NEWGROUPS    HELP
IHAVE       NEWNEWS      SLAVE

Additionally, the following extension is supported:

XHDR        Retrieve a single header line from a range of articles.

Bugs to Phil Lapsley (Internet: phil@berkeley.edu; UUCP: ...!ucbvax!phil)
.
input> command.group nttdata.test
211 6 85 90 nttdata.test
input> command.body
222 85 <28109@nttdpe.RD.NTTDATA.JP> Article retrieved; body follows.
input> read text
Sorry. Just a test.

---
    //v--\
   /   /\___\       Toshiharu Harada
   \--[=][@]/
     |    > |       NTT DATA COMMUNICATIONS SYSTEMS CORPORATION
     \  - /


.
input> command.quit
205 nttose closing connection.  Goodbye.    # from "nntp"
input> quit
700 bye                                     # from "pcserve"
$
```

Architecture of a Collaborative System Using Smalltalk and Unix

C Vieville
A Derycke
P Vilers

Trigone Laboratory
University of Lille
Vieville@frcitl71.bitnet

ABSTRACT

Our team of Trigone laboratory works in the field of "New Technologies of Education". The aim of this work is to develop tools which can help tutors and students. The main features of these tools include an user-friendly interface as well as important communication facilities between the users' workstations. The paper will show the various problems that we have had to work out so as to fit Smalltalk to UNIX in order to set up a rapid prototyping platform.

After an explanation about the choice of Smalltalk, we will further specify our communication needs before analysing Smalltalk's handling facilities of external events. We shall end with a description of our communication architecture upon the sockets.

1. Smalltalk: A Rapid Prototyping Language and Environment

The platform for software development that we wish to set up is intended for incremental prototyping of applications needing sophisticated user interfaces. We have chosen Smalltalk-80. This language is now available on numerous machines and in various environments. We can buy a version for Mac II, IBM PC/AT 386 (with 4M ram), Tektronix, Sun, etc.

1.1. A bottom-up and top-down development

Smalltalk is one of the most popular object-oriented language and it also provides programmers with a rich programming environment for software engineering. The two important concepts of this language are encapsulation and hierarchy [Mil87a].

Encapsulation is useful for hiding the internal structure of objects only accessible by an interface (the access protocol). Objects are organized in a class and subclass hierarchy. The class of an object defines the messages that it can respond to, this class can inherit many messages from all its super-classes.

The set-up of an evolving prototype requires consideration about its general architecture. We must use a top-down approach. Yet, during this phase, we are often confronted with the need to experiment with alternative approaches in order to solve a part of the problem. So, it would be of interest to use a bottom-up development too. In Smalltalk, these two methods exist. The class creation is top-down and the fast procedure (i.e. a method in Smalltalk) creation can be viewed as bottom-up. Thus we can experiment and modify our important design decisions straightforwardly.

Smalltalk encourages experimentation. A variable is not typed. Its type is checked during execution of a method that uses it. This late binding between an object and a variable allows us to test an object that is not fully defined. The reaction of the object helps us to define it more precisely.

Like all object-oriented languages, Smalltalk allows reusability of software components [Mey88a]. It transforms the programmer into a reader; with this kind of language, one understands the benefit that can be obtained by reading how a new unknown object is used in an example program provided with the new class. New objects are re-used by imitation.

1.2. The programming environment

The multiple overlapping windows of Smalltalk give the programmer a rich environment. Numerous classes are provided in order to handle graphic and interactive user interfaces. Underlying inheritance allows a very fast construction of flexible interface based on the M.V.C. model.

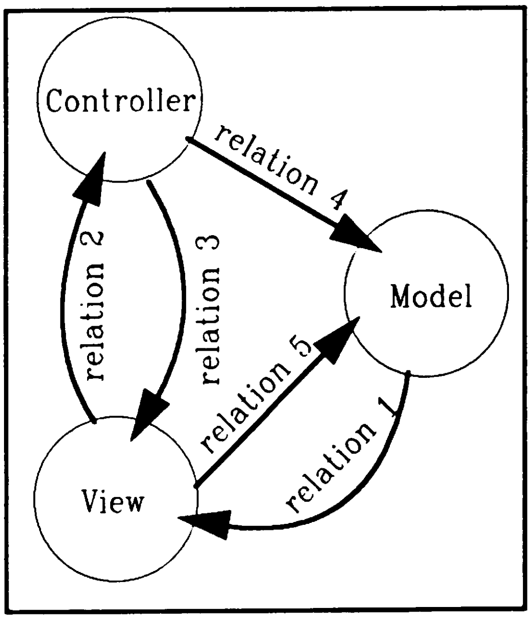


Figure 1: *The MVC model of Smalltalk-80*

The MVC model (figure 1) is a framework given by the system to develop interactive applications. We have to design our application in three parts. The model stores the state of the application object, and it uses a view to display its state (**relation 1**). This view reads some internal data of the model to display on the screen (**relation 5**). Each view has an associated controller to manage the user's interactions (mouse move, keyboard press, etc.). Many interactions of the user do not modify the state of the model. In this case, only **relation 2** and **3** are activated. For example, a scroll for a Model of text does not change the content of the text, but only the part of text the user sees. When he clicks to have a pop-up menu, he can change the state of the object with the **relation 4**; the model executes the change and uses the **relation 1** to indicate this modification. The views are refreshed. Many classes of views and of controllers are provided with Smalltalk and a programmer can rapidly build up an interactive application.

In addition to this multi-windowing, Smalltalk gives important tools to help the programmer. A browser authorizes paperless programming. It is a system management of all the documentation of programs. The browser helps us to show on screen the class hierarchy, to find an object that can interact with another by means of a particular method. The system can be entirely "explored" with very powerful tools. We may use an inspector that shows the internal structure of an object. This inspector connected to a debugger allows a very good perfection of methods ([Gol84a] and [Tes81a]).

If it is true that learning Smalltalk is a very long process compared to the one of a conventional procedural programming language, the long-term productivity gains may be worth the initial investment [Mil87a].

2. UNIX Communications and Smalltalk

2.1. Our communication requirements

The applications that we develop belong to the field of tools to help students and tutors during a distant learning-teaching process. We want to build a computer-based real time multimedia conferencing system where a small group of learners (4 to 5) can work with a tutor. It appears that we need to exchange a lot of information between the stations. Each participant should see on his screen, in realtime, the changes of the application that is the "subject" of the conference. Our prototype, built in Smalltalk on a SUN under UNIX, must allow us to validate the control algorithms for this conference; it must also allow a rapid modification of algorithms. We have chosen UNIX for all its communication facilities. Network Communication and IPC (messages, semaphores, and shared memory) are very useful in our development [Gua88a].

2.2. The object and message exchange

Smalltalk consists of a Virtual Image (VI) and a Virtual Machine (VM). The Virtual Image is the set of objects in the system and the Virtual Machine is the storage manager and interpreter. In our case, several Smalltalk Images have to communicate. The aim of our work is not to realize a distributed Smalltalk and above all, we do not want to modify the Virtual Machine as [Cul87a] and [Ben87a]. Approaches with modifications of the Virtual Machine have been made by [Dec86a] and by [Ble88a].

Our main goal is to send a message to a remote object and to retrieve the result object back. This work is very similar to a Remote Procedure Call. First, we have to deal with the external representation of our structured objects. This corresponds with the XDR when we use RPC and this kind of work has already been made by Vegdahl [Veg86a].

Smalltalk does not provide a structured object linear representation tool. So we have developed a Loader class to linearize objects on a stream. With these "goodies", we can exchange structures between Smalltalk Images without an activation of the compiler.

In the version 2.5 of Smalltalk, we can use the BinaryStorage class which allows us to linearize an Object. The external representation of an object is realized starting also from the works of Vegdahl. We can use this tool because it presents the same particularity as our Loader class. An object is serialized in a ByteArray with:

```
        ba := BinaryStorage put: anObject.
```

It is created in an Image with the following message:

```
        object := BinaryStorage from: aReceivedByteArray readStream.
```

This kind of persistence is used for the exchange of objects between workstations.

2.3. Smalltalk and the external events under UNIX

This study has two phases. The first one has been conducted with release 2.3 of Smalltalk-80 on a SUN workstation. A first report has been written; it pointed out some deficiencies but an architecture for collaborative applications had been clearly defined.

The new version of Smalltalk, named ObjectsWorks, which has been available since August 1989, has compelled us to restart the study again. In this second phase, we have found out that the previous deficiencies have disappeared and that Smalltalk makes use of all the functionalities of UNIX. Its level of integration is much higher than in the previous release. Some parts of our architecture have been modified and the new classes provided by Parc-Place have been used instead of those we had developed. In order to understand how Smalltalk is well integrated in UNIX, we shall rapidly introduce the two phases of our study which consists in synchronizing Smalltalk activities with a UNIX Process.

2.3.1. The first phase of the study

The communication between two Images

In a second step, we have to give a transparent forwarding means for the programmer. We have chosen the sockets of BSD as IPC mechanism. We must state that the Socket class provided in the standard Image had been an important factor of decision for Smalltalk as our development system.

Unfortunately, this class does not offer all the primitives that are useful to generate a server process on a socket. We had to make them using the C language. The following functions were not implemented: *bind()*, *listen()* and *accept()*.

Development of primitives

Smalltalk gives the possibility of extending the Virtual Machine with primitives written by the user (User Defined Primitives). The file *userprim.h* gives the programmer a mean of linking his compiled code with the object file of the Virtual Machine (VM). Many functions allow the conversion of a C structure into a Smalltalk Object and conversely.

```
#include "userprim.h"
#include <sys/types.h>
#include <sys/socket.h>

/* function called by Smalltalk 2.3
   to listen on a socket : nSocket
   wait for 1 client at time
*/

upInt smListen(recv,nSocket)
  upHandle recv;
  upInt nSocket;
{
UPbegin(recv,1);                   /* method with 2 arguments          */
UPreturnHandle(                    /* return to Smalltalk              */
  UPCtoSTint(                      /* convert a C int to a SmallInteger */
    listen(                        /* call the function                */
      UPSTtoCint(nSocket),1)));    /* convert nSocket to a C int       */
}

char *
  UPinstall()      /* called by VM at beginning to install the C function */
{
UPaddPrimitive(-1,smListen,1);
/* the primitive has the number -1(negative value) */
/* name of C function and number of arguments      */
return("the listen function is installed");
}
```

Figure 2: *The User Defined Primitives for the function listen()*

In figure 2, we can see a C source of the primitive for the function *listen()*. The words beginning by UP are macros or functions of the file *userprim.h*.

This extension of the Virtual Machine is not entirely satisfactory because we have to respect rules in the C source writing. An important rule is the one that forbids us to make a system call like a *read()* which might cause Smalltalk to block until the call is completed. During this time all the other Smalltalk activities cannot run.

In order to manage multiple independent activities, Smalltalk proposes the creation of process with 3 classes: Process, ProcessScheduler, and Semaphore. Processor, the single instance of the ProcessScheduler class chooses the process through which the code should be executed by the Virtual Machine. This is done according to the state and the priority of the process. Semaphores are used to synchronize the processes.

But a new problem arises: How to synchronize a UNIX Process with a Smalltalk activity?

The Synchronization between Smalltalk and UNIX

The management of the activities by process class is made by the Virtal Machine. The mechanism is a semaphore linked to an external event. If the VM knows this event we can find somewhere in the Smalltalk Image a primitive to link our semaphore to it. When this event occurs, it is captured by the VM which signals the linked semaphore. Then, before fetching a byte code, the VM scans the set of semaphores and wakes up all the waiting processes on the signaled semaphores. According to its priority, a process can immediately use the VM to run its code, otherwise the VM puts it in a Ready to Run Queue. The synchronization by this means can be done by a well-known set of events. Nevertheless, we can realize the synchronization if we develop a User Defined Primitive in C which captures a signal attached to an external user event (figure 3).

```
#include "userprim.h"
#include <signal.h>

TYPE8=32768;                /* user event for smalltalk */
SIGN=32;                    /* the signal to catch      */

sigCatch()                  /* handler                  */
{
UPpostEvent(UPCtoSTint(TYPE8));     /* to interrupt the VM      */
signal(SIGN,sigCatch);              /* to handle the signal again*/
}

upHandle smCatch(recv,SIGN)   /* call from Smalltalk */
  upHandle recv;
{
UPbegin(recv,0) ;           /* unary message                       */
UPreturnHandle(             /* return result to Smalltalk          */
  UPCtoSTint(               /* convert a C int into a SmallInteger */
  signal(                   /* calls the function                  */
  UPSTtoCint(SIGN),         /* convert aSmallInteger into a C int  */
  sigCatch)));              /* the handler                         */
}

char *
  UPinstall()               /* installation at the beginning       */
{
UPaddPrimitive(-3,smCatch,0); /* this primitive has number -3      */
return("installation of the catch function");
}
```

Figure 3: *Catching an external event*

In the defined handler of the *signal()* function we have to send a message to the VM which is able to understand it as a user event (type=8). After this step, we can write a Smalltalk process waiting for this type of event in the input loop. When an external UNIX process sends a signal, the Smalltalk process is awoken.

Instead of polling the external activities, such as a "no waiting" read on a pipe or a file, we can use this technique that gives CPU time to all the other activities inside Smalltalk.

The modification of the polling of inputs in MVC

In the multiwindowing system proposed by the MVC model, the methods are written with a very heavy use of a polling strategy for the management of the interactive inputs (mouse or keyboard). So, Smalltalk uses the CPU when "no useful work" is to be done. Cooperation with all the other UNIX processes is not very good. Schiffman [Sch88a] proposes a modification. It consists in giving the CPU to another UNIX process after an amount of time without interaction from the user. This small modification seems to be essential when a Smalltalk program interacts with another UNIX application.

2.3.2. The second phase of the study

The most important deficiency of Smalltalk V2.3

The Synchronization with a UNIX process seems to be the principal deficiency of Smalltalk. This mechanism has been written with C primitives. Of course, the use of the C primitives do not favour the flexibility of our platform. Fortunately, the present version of Smalltalk resolves all the previous problems.

The input/output system

In the version 2.5, we find an important abstract class to manage I/O access with file descriptors. According to the platform (MS-DOS, UNIX, MAC) we use a subclass which offers various facilities like the ones we have when developing in C language. In particular, the management of the synchronization on the reading/writing operations is very easy because the VM accept this primitive (for the UNIX platform in the UnixIOAccessor class):

```
SetSem: aSemaphore forWrite: aBoolean.
```

This allows us to link a semaphore to a reading/writing operation on a file descriptor. Built from this primitive, some other features like read or write can now suspend only the invoking process and not all the Smalltalk activities.

The Socket class

This class has been entirely modified. It is now a subclass of UnixIOAccessor and the previous difficulties do not exist any more. A set of operations, provided by Parc Place allows us to use a socket as a stream and the input/output are now harmonized with all the other stream protocol.

Figure 4 shows an example of a server and of a client process.

```
!Connection class methodsFor: 'client-creation'!

clientOn: aService
    "generate a client on aService"
      skt
    " the server must be ready to accept the connection "
    "get a socket on a server workstation"
skt := UnixSocketAccessor newTCPclientToHost: aService hostName
                                              port: aService port.
    " start a process on the slave socket"
^ self new generate: skt !

!Connection class methodsFor: 'server-creation'!

serverOn: aService
    "start up a server "
    | skt d ns |
    "take a TCPIP address and a port number from aService"
skt := UnixSocketAccessor newTCPserverAtPort: aService port.
    "one client at time"
skt listenFor: 1.
d := Delay forSeconds: 5.
ns _ nil.
    " start a server process for aService "
[ [ ns isNil ] whileTrue:[
    d wait.
    " every 5 seconds test if a client is connected on the port"
    ns := skt acceptNonBlock].
       self new generate: ns ]
    "start a process on the slave socket and loop again"
          forkAt: Processor userInterruptPriority!
```

Figure 4: *A client and a server process*

3. Architecture of the Communication System for Prototyping

3.1. Our requirements

We want to build up a system which can easily support collaborative applications. With the previous tools (BinaryStorage and Socket) we are able to construct it. Our problem is not to realize a pure distributed system but:

- We only want to make as transparent as possible the calls to a remote object.
- In our case the classes are permanent during the execution time of the application.
- We have only to deal with a few objects.
- The creation of these objects need not be transparent for the programmer. He will declare that he wants to access a remote object known by a Unique Name. We simplify the localisation problem.
- The objects do not move across machine (no "call by move").
- Actually, we prototype on a local area network (Ethernet) which allows direct communication between all the stations. But in the final version, we plan to use a public network (ISDN) and all the direct communication links will not be possible. So, we decide that all the communication links are centralized on a central station and that there is no connection between the other stations.

In the following part, we describe our basic objects and we finish by a presentation of the modified MVC model where some users (4 to 5) can cooperate in real time within an application.

3.2. The basic objects

3.2.1. The Policy class and its subclasses

An instance of this class can easily encapsulate any other Smalltalk object. It captures all the messages which are aimed to this object and then process it according to some rules.
The major features of this capture are:

- to redirect the message over the network to a remote object, if the object is not on the same machine.
- to give protection for concurrent access to the object if it is a local one.
- the Policy class is a protection and indirection mechanism. But this class is an abstract one and we have developed four subclasses.

The Proxy class

This class is used when the object is not local. We need only the indirection mechanism. Using a link to the remote object, it sends the message over the network and waits for a result object.

The SharedObject class

This class allows any object of the Smalltalk Image to be used by multiple sending process. The sending process may be a local process or a remote one. An instance of this class always returns an object. A proxy is associated with a SharedObject.

The UpDateObject class

In Smalltalk Image, some objects, have dependencies with other ones. So, when this kind of object is modified, it broadcasts a message to all its dependencies. The result object has no meaning in this situation. The UpDateObject broadcasts the captured message to all its remote dependencies and does not wait for a result back. The remote object is also an UpDateObject.

The CooperativeObject class

By now, we have developed only the conference manager through this means. The programmer inherits the protection and the indirection mechanism to build his own methods. The remote object is always an instance of this class.

3.2.2. The network objects

The Connection class

There is only a single connection between two machines. The connection manages a socket as an IOstream, the serialisation of the message or the result object, and all the input data from the socket. A thread is defined to manage the data coming from the socket. In order to dispatch this information to the corresponding policy, it analyses the incoming packet which is described below.

The identification of the object

All the instances of subclasses of Policy have a Global Identity (GID) which is unique for all the connected machines. This Global Identity is given by a special object which is the nameServer. It will be presented below.

The format of an exchange packet

In an exchange packet, we have to put the message or the result object and some information which are use by the connection. All the data in this packet are in conformity with the external representation we have defined in part 2. These constraints correspond to the eXternal Data Representation of the Remote Procedure Call. The first field of the packet is a byte that represents the Global Identity of the object. The connection can find the right object represented by the Global Identity by using a table named LinkTable. The second field of the packet is the type of the packet:

- open packet to indicate the creation of a new policy,
- close packet to indicate that a policy is never used,
- execute packet to send a message (calling mechanism),
- reply packet to send a result object after the execution of a message (replying mechanism).

The third field is the length of the fourth field. The fourth and last field transports the byte array with the linearized object. The connection is responsible to create this object (message or result object) in the Smalltalk Image and to generate a process that sends it to the right policy.

The link

The link stores, on each machine, the indirection between the policy and its connection. Several Policies may use the same connection if their remote object stays on the same machine, but a policy can be connected to several connections. So, a link is defined as a pair (a Policy, a set of connections).

The LinkTable

This global variable is defined as an Array of links indexed by GID (Global Identity). When we know this GID, we can find the local policy where to execute a message, and we also know where to send a broadcast message.

The Naming Service

To achieve the system, we have to deal with the initialisation problem. We create a NameServer as an instance of the SharedObject class. This object is located on the central machine and its GID is equal to 0. At the first creation of connection on the server, this object is started and a proxy is also started on the other station at its first connection to the central station. This NameServer can:

- add a Name of Object and return the GID,
- return the GID of an existing object,
- remove an existing GID.

3.3. A first use of our architecture: the Conference Manager.

The conference manager is an application that allows us to share a standard Smalltalk Application written according the MVC rules.

3.3.1. The rendez-vous

The Conference Manager on the central station opens a socket which waits for the first connection. When a station connects itself to this socket the first connection is started and the NameServer is started on the two machines (figure 5).

Just after, the conference Manager (a CooperativeObject subclass) is also linked to the connection. The Conference Manager of the connected station exchanges messages to know who is connected. After a while, the users can decide to begin the cooperative work.

3.3.2. The shared application

An application written according the MVC rules can run on the central Station as a SharedObject and an associated RemoteMVC is run on the other stations. A RemoteMVC consists of a controller and a view belonging to a class which is consistent with the central station's one, but in this case the model becomes a proxy.

All the views of the MVC model are dependent on the top view. To realize this mechanism, we have to introduce on each station a viewManager which is an instance of the UpDateObject class. When there is a modification on the model, the latter changes all the views attached to it, and the viewManager broadcasts the change to all the other views. All the users of the other stations can see, in real time, the modification of the view of the active one.

In this typical case of use of our architecture, we only need 4 GID. The cooperative application is written without knowing the existence of our underlying architecture. We have developed the conferenceManager as a subclass of CooperativeObject and the viewManager using an instance of UpDateObject class. The programmer of these classes has to pay attention only for the creation of our policy objects, but after he can send messages in a transparent way.

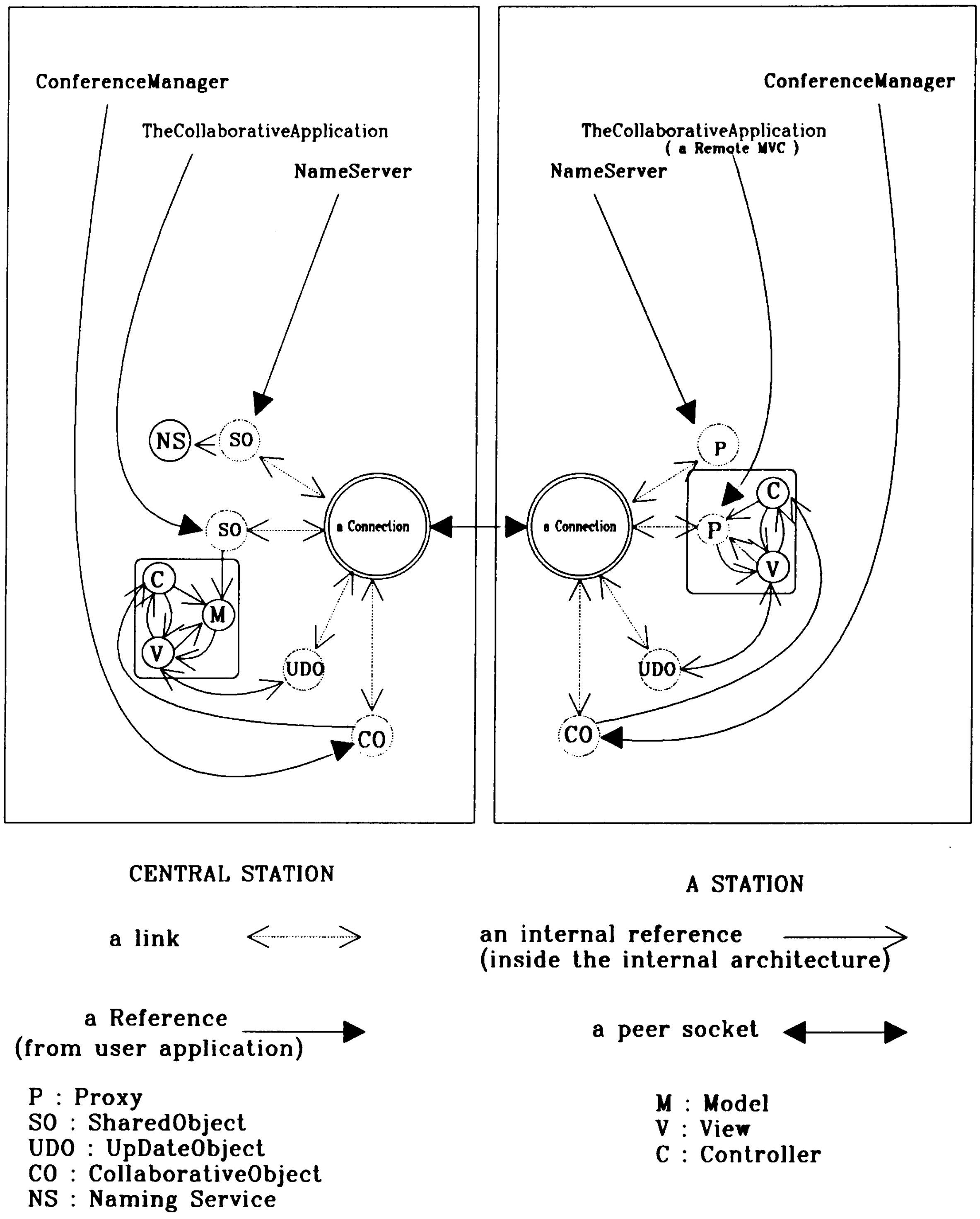


Figure 5: *The Conference Manager and the shared Application*

4. Conclusion

The new version of Smalltalk allows us to easily use the communication facilities of UNIX. If an access to a function provided by UNIX is not possible with the Smalltalk language, we can develop a primitive in C. But this development should be as short as possible to give flexibility for the next changes of the prototype or of the Smalltalk version. By using only Smalltalk code we can develop a platform where the objects can send messages to a remote one. The level of integration of Smalltalk with UNIX seems to be very high.

References

[Ben87a] John K Bennet, "The Design and Implementation of Distributed Smalltalk," *OOPSLA '87 Proceedings*, pp. 318-330, 1987.

[Ble88a] Eddy Bledoey and Marcel Schelvis, "The implementation of a distributed Smalltalk," *OOPSLA '87 Proceedings*, pp. 212-232, 1988.

[Cul87a] Paul Mc Cullough, "Transparent forwarding: first steps," *OOPSLA '87 Proceedings*, pp. 331-341, 1987.

[Dec86a] A D Decouchant, "Design of a distributed Object Manager for the Smalltalk-80 System," *OOSPLA '86 Proceedings*, pp. 444-452, 1986.

[Gol84a] Adele Goldberg, *Smalltalk-80: The interactive programming environment*, Addison-Wesley, 1984.

[Gua88a] Sheng Guan, Jay Nievergelt, and Aussein M Abdel-Wahab, "Shared workspaces for Group Collaborations: an Experiment Using Internet and UNIX Interprocess Communications," *IEEE Communications Magazine*, pp. 10-16, November 1988.

[Mey88a] Bertrand Meyer, *Object Oriented Software Construction*, Prentice-Hall, 1988.

[Mil87a] Jack Milton and Jim Diederich, "Experimental prototyping in Smalltalk," *IEEE software*, 1987.

[Sch88a] Allan M Schiffman, "Time Sharing Citizenry for Smalltalk-80 under UNIX," *Parc-Place NewsLetter*, vol. 1, no. 2, pp. 9-10, winter 1988.

[Tes81a] Lary Tesler, "The Smalltalk environment," *BYTE Magazine*, pp. 90-147, August 1981.

[Veg86a] Steven R Vegdahl, "Moving Structures between Smalltalk Images," *OOPSLA '86 Proceedings*, pp. 466-471, September 1986.

ITHACA: An Overview

Anna-Kristin Pröfrock†
Martin Ader
Gerhard Müller
Dennis Tsichritzis

Nixdorf Microprocessor Engineering GmbH, Berlin

ABSTRACT

This paper describes the Ithaca project (**I**ntegrated **T**oolkit for **H**ighly **A**dvanced **C**omputer **A**pplications) developed under ESPRIT contract 2121 as a technical integration project. Ithaca is an integrated application support system which is evaluated by demonstrator applications, even at an early stage of development. To a high degree, the Ithaca environment supports an object-oriented approach to application development. Thus, it consists of a kernel including a concurrent object-oriented programming language called CooL, with support of persistent objects through integration of a structurally object-oriented database system interface called NooDLE. Development of object-oriented applications is supported by tools which build an application development environment. Here, special emphasis is placed on a modified object-oriented life cycle in application development. The users of an Ithaca application will be supported by a user support system based on X-Toolkit. The Ithaca office workbench includes a multi-media environment and supports typical office organisations. Other demonstrators to evaluate Ithaca are a chemistry workbench, a financial workbench and a public administration workbench.

1. Introduction

The objective behind work on the Ithaca project is to create a platform for the development of information and production control systems. At present, applications of this kind have a long and cost-intensive life-cycle. We use a coherent object-oriented approach to develop applications in a way which is faster, more reliable and more productive in comparison to the "traditional" approach to software development. The idea of "application support systems" or "operating environment systems" are well acknowledged in the software engineering community. Up-and-coming products, such as Steve Jobs' NextStep, for example, show the relevance of this idea for future markets. Ithaca develops this idea further, the aim not being to achieve niche applications, but rather to reach the market for data-intensive applications. For this reason, Ithaca is designed as an open system, and consideration is given to recent and future standards or industrial *de facto* standards.

2. The Object-Oriented Kernel

It is meanwhile widely accepted that the object-oriented approach to software development provides valuable support for the construction of reusable, integratable and open software systems. However, its use is largely restricted to systems, tools and interface programming. In the field of information system development we do not know of any serious attempt to use an existing object-oriented programming language for this purpose.

We may argue that the reason for this neglect is due to the additional requirement of information system building to handle persistent data. In most production environments persistence of data is achieved by storing this data in a database system and by delegating information-sharing and retrieval, transaction control and recovery to the database system. The integration between database system and language is usually achieved through embedding the data management language into the programming language. The

† The material contained in this paper was produced by the respective working groups.

programmer is usually required to have at least some basic knowledge of two different (and sometimes even contradictory) concepts and mostly he or she is also required to use two different programming languages.

However, in object-oriented languages the situation is even worse. Due to their application domain object-oriented programming languages do not bother about persistent data and information management. At best, object-oriented languages support some mechanism for dumping objects to secondary storage.

The kernel of the Ithaca environment, called HooDS (Highly Object-Oriented Development System), combines those worlds and offers both an object-oriented programming language (called CooL) and an integrated structurally object-oriented database system interface (called NooDLE). The kernel is reinforced additionally by a number of low level tools.

CooL in itself is a newly developed language. Nevertheless most of its concepts are derived from existing languages. A different strategy would have been to adopt an existing language and extend it by object-oriented concepts and by a call interface to a structurally object-oriented database system. To do so and not to end up with a hybrid dinosaur seemed to us likely to fail (some C++ with Pascal/R).

Instead we chose Eiffel as a conceptual guideline for our design. Eiffel is an object-oriented programming language which is especially suited for system and tools programming. We extended this concept by persistent objects, external interfaces, secure parameter passing, processes, streams and other mechanisms especially useful for the development of large-scale information systems. However, for the sake of adequacy for today's programmers we also dropped a number of features related to inheritance (multiple and repeated inheritance), genericity (instead we re-introduced the concept of type constructors) and the mechanisms for specification (which should more likely be supported by the Ithaca tool environment).

NooDLE (New Object-Oriented Database System for advanced programming Language Environments) is a structurally object-oriented database interface implemented on top of a relational database system. The NooDLE data model NO2 (New Object-Oriented data model) is a newly developed data model. However, most of its concepts are derived from existing data models, such as NF2 (non-first normal-form), ENF2 (extended NF2) or O2 (object-oriented data model).

In addition to the usual basic types, NO2 supports the type constructors TUPLE, SET and TENSOR (corresponding to the CooL types). The use of type constructors is orthogonal. NO2 supports type referencing as well as inline expansion of types/objects. Referencing of types means reusing previously defined types. Inline expansion of a type means any improvement of its internal structure using a type constructor. 1NF and NF2 relation types may be modelled by nesting of tuple and set type constructors.

In contrast to established data models, the NO2 data model distinguishes between the definition of a (composite) type and the declaration of a type extension to keep instances of a type. The mechanism to distinguish between a type and the extension of a type is well-known from programming languages. An extension in NO2 may be compared with a variable of a programming language.

NO2 does not support either inheritance or object encapsulation. Inheritance hierarchies defined in CooL must be appropriately mapped to NO2 object types. Since object encapsulation is not provided at database level, all components of an object may be reached using generic operators. This renunciation permits access to NooDLE even by programming languages which do not support the object-oriented paradigm. To ease this use NooDLE offers an SQL-conformant query interface called NooQL (New Object-Oriented Query Language).

3. Application Development Environment

The goal of the Ithaca Application Development Environment (ADE) is to reduce the long-term costs of application development and evolution for standard applications from selected application domains. By "standard" applications, we mean classes of similar applications that share concepts, domain knowledge, functionality, software components, or which can otherwise be seen as variations of a generic application.

Although the Ithaca environment and tools will not be specific to any particular application domain, there will be an initial overhead in tailoring the ADE to a domain of interest. Thus, on the one hand, it is precisely those domains for which many similar applications are to be generated that would benefit most significantly from the ADE tools (and can therefore justify the short-term overhead), and on the other hand, it is the fact that the applications share many common characteristics that will enable the ADE to exploit reusability during the various phases of application development.

The Ithaca project contains several examples of suitable application domains, such as case processing, financial management, and other examples from office information systems, and information systems in general. These are included in the form of demonstrators.

The scope of the tools in the Ithaca environment spans two closely related activities: that of tailoring the software base to particular application domains, and that of using the tailored environment to construct a specific application. Ultimately, the goal is to reach a point where standard applications can be quickly built and easily modified to meet evolving requirements. The tools support the application engineer who must take the steps leading up to this goal.

The Ithaca ADE distinguishes between two closely related roles: that of the *application engineer* whose responsibility is to tailor an Ithaca environment to an application domain by packaging information and software pertaining to application development; and the *application developer* whose responsibility is to generate specific applications. The application engineer is concerned with all phases of software development (i.e. from requirements collection and analysis down to coding and validation), but does so for so-called *generic applications* rather than specific ones. The initial overhead of developing generic applications is expected to be higher than that of developing specific applications using traditional techniques, but the long-term pay-off can be significant. The application developer, starting with a generic application, collects application-specific requirements, selects from possible design choices, and configures the final application from the available software components. In exceptional circumstances, the generic application may need to be adapted to meet unexpected requirements.

The proposed Ithaca Application Development Environment will consist of an evolving *Software Information Base* (SIB) and a collection of tools to support the application engineer and the application developer in the construction of generic and specific applications. Loosely speaking, the application engineer is the producer of the contents of the SIB, and the application developer is its consumer. The SIB contains and organises all information pertaining to application development, i.e. the application domain model, the requirements model, functional specifications, software components and their specifications, documentation, etc. A bare ADE may contain an "empty" SIB, or one containing only the most generic software information (standard libraries, etc.). As the SIB grows and evolves, the ADE does a better job of supporting the application developer.

As the primary mechanism for organising the SIB, we introduce the concept of an *application frame* (or simply "frame"). A frame collects and organises all application development information pertaining to an application, whether generic or specific. The goal of the application engineer is to produce *generic application frames* (GAFs), which serve as skeletons or templates for the application developer to work on. Starting either with a more-or-less empty frame or with one or more existing frames, the application engineer develops a GAF which encapsulates all information concerning a class of standard applications, from the requirements model down to generic software components. The application developer selects a GAF, and fills in the missing information (requirements, design choices etc.) until a frame for a complete, specific application is produced.

The following capsule scenarios outline the steps in developing applications within Ithaca:

1. A bare ADE provides only a general-purpose SIB (i.e. containing basic software but no GAFs).
2. Precursor applications are developed using an object-oriented design methodology (i.e. factor out functionality into encapsulated object classes that promote reusability via inheritance and genericity). All information pertaining to application development is structured within an application frame. External precursors are also expected to contribute in this step.
3. Based on the experience of multiple applications, the application engineer constructs a GAF by extracting and possibly redesigning the common parts (i.e. requirements model, etc., as well as software components).
4. An application developer constructs specific applications by selecting and filling in GAFs. The SIB may evolve as GAFs are re-evaluated to improve their reusability. New software components are added to the SIB as necessary.

As a variation on this scenario, we also consider the possibility that the application precursors may have been developed outside of Ithaca. In this case, GAF development in step 2 would also entail object-oriented redesign of the applications and possible reverse engineering of software components.

The tools provided by the ADE, together with the SIB, support a frame-based, object-oriented application development methodology. They include a *graphical tool* offering a general-purpose graphical editing interface to the other tools, a *selection tool,* which functions as a filter/browser for querying and navigating through the SIB, a *requirements collection tool,* a *requirements specification tool* for acquiring design knowledge and application domain knowledge for specification of applications, a *design tool* to produce and manipulate application design information, and a *visual scripting tool* to aid in the construction of applications from parameterised software components (such as object classes) prepared by the application engineer. Tools for monitoring and debugging and configuration management will also be part of the complete ADE.

4. Application User Support

Human factors engineering results are available in the literature, but are difficult to access and use when actually designing user interfaces. With the aim of collecting and organising such data, our effort consisted of collecting human factors recommendations out of current human factors guides and literature reviews.

From this work, various conclusions can be offered:

- Recommendations are usually simply organised according to major interface elements, such as data entry, data display and sequence control.
- Recommendations are often too general to be usable directly by non-expert users.
- A major issue concerns the interacting dimensions of some guidelines (e.g., usage *versus* learning performance).

So far, we have attempted to collect and decipher a large amount of human factors recommendations. They have been translated into elementary rules (one premise/one conclusion). Each rule has been characterised by:

- the type of interface element concerned;
- the type of interface level concerned (using abstract layered models);
- one or more design criteria.

Sixteen tentative criteria have been selected empirically: compatibility, consistency, brevity, flexibility, immediate feed-back, mental load, minimal actions, explicit actions, significance of coding, user control, user guidance, smart features, user experience, data grouping, data distinguishing, task-related items.

All this preliminary work was conducted on a HyperCard self-tailored application and then translated into text files. About 500 individual rules have been collected.

Our first aim is to explore the various solutions for giving a programmer using an object-oriented language graphic tools providing the same look and feel as in the X-Toolkit. In an X-Toolkit, several levels of interaction are offered to the interface programmer:

- The lowest level is the X library (Xlib) which provides primitives for the basic graphic output and input.
- Layered on top of the Xlib, the X Intrinsics level provides generic functions to create and handle instances of interactive objects called *widgets.*
- Finally, the toolkit level usually offers customised primitives for each type of widget, and possibly a language for describing the interface initialisation, such as the UIL in the OSF Motif.

On the other hand, no specialised tools are offered to programmers who are widget developers, hence their task can easily become cumbersome. When studying scenarios offering a user interface environment, we will have to take into account which level(s) of interaction will be provided to toolkit users and whether we consider that widget developers should be able to perform their task within the object-oriented language.

Several approaches can be envisaged for providing the tools and the look and feel of an X-Toolkit in an object-oriented language. For instance, a "black box" solution can be adopted, which means that all the C primitives offered by the toolkit to the user will be directly interfaced in the object-oriented language. At the other end of the spectrum of solutions, the toolkit can be completely rewritten in the object-oriented language and provide an Application Programmer Interface (API) consistent with the object-oriented language features, only preserving the toolkit look and feel.

In order to choose the solution(s) which seem(s) realistic from the range of possible solutions, we first have to identify some guidelines on what can be considered a good user interface environment. For example, the graphical toolkit provided from the object-oriented language should offer response times close to those of the reference toolkit. And since we intend to rely on an environment (OSF Motif) which will be exposed to various modifications from its original developers, and to numerous (necessary and useful) extensions, it should be possible to offer these new tools inside the object-oriented language at short notice.

5. The Office Workbench

As set of advanced office applications will be implemented on top of the Ithaca environment. These applications will support cooperative work in a structured office environment with explicit representation of the office organisation and procedures.

These office applications will integrate advanced office tools like multi-media editors, filing servers and X.400 electronic mail systems. Extensions to the Ithaca environment will be built for that purpose in the form of a library of objects for office applications and office modelling, as well as a library of knowledge-based tools that can be used for other purposes than office applications. This office solution is based on five years of research in the office automation field in different ESPRIT-1 projects. The main building blocks of this office environment will be the following:

- An abstract office model that enables the representation of organisations, of documents, of facilities and of administrative procedures.
- An activity mangement system enabling the description of administrative procedures and their interpretation for user assistance in a cooperative way.
- A budget management system offering a constraints-based expert capability for budget preparation and optimisation.
- A multi-media environment made of filing, printing and electronic mail servers that can be used by workstations in a distributed way.
- A desktop application that provides each workstation with an object-oriented consistent view of the various facilities offered by the multi-media environment, the activity management system and the abstract office model.

For developing the above applications, various facilities are provided to the Ithaca environment:

- A Lisp environment will be provided as a prototyping tool for projected applications. Object-oriented interfaces to the Motif toolkit of OSF will be prototyped on top of that environment.
- A set of knowledge-based tools will be developed as basic tools for supporting the above applications and will be available for other applications as an extension of the CooL library.
- Advanced user interface tools for help and tutoring will be designed on top of the Ithaca user interface toolkits. An expert system for interface design will be provided.

6. The Chemistry Workbench

The process industry, in particular the chemicals, pharmaceuticals and foodstuffs industry, is investing increasingly in integrated manufacturing concepts (CIM). The special features of lot-oriented manufacturing processes place special requirements on production packages, such as production planning and control (PPC), quality assurance and process monitoring. Due to the fact that liquids, gases and granulates are processed, the production prerequisites for the process industry differ from those for industries involved in parts production. For this reason, an autonomous solution concept has had to be developed for this sector of industry.

In large companies from the chemicals industry, controlling of processes is developed on the basis of the demands placed by process engineering. Today, proven process management systems are available which cover all monitoring and control functions for a particular plant or for an entire setup. Aspects of production planning, integration with materials management and commercial applications have not been realised. On the other hand, the applications covering the company as a whole from the commercial or logistics sector show few signs of being able to attain continuous functional chains — in the sense of CIM — by enhancing existing software. The stumbling-block lies in the production setups in which the link between the process management level and the company management level must be created.

The demand for shorter delivery times, improved utilisation of capacities and a reduced amount of stock on hand is making it necessary to develop integrated production concepts which cover all sectors of production, from production planning right up to process control and process monitoring. The PPC systems available on the market are, without exception, geared to parts production of the type found in the field of mechanical engineering, for example.

In Ithaca, parts of the comprehensive solution for chemistry sectors will be realised. The topics of interest are integration of CIM concepts to office automation, rapid prototyping, graphical supported user interfaces and the development of a generic solution to be customised for special needs. For all those aspects the object-oriented approach seems to be the most promising.

7. The Public Administration Workbench

The needs of public administration in the coming years will evolve in the sense of information systems able to handle multi-media objects and to perform a wide range of tasks in a cooperative way. The ability to be adapted to an evolving task mix will be the main success factor, thus creating applications which are dedicated to a specific set of tasks and which are unable to evolve obsolete. Office work in public administration is often highly structured. Work-flow processors or case handling systems will play a prominent role in office systems for public administration. Case handling and procedure automation are well-adapted to the concept of object-oriented systems. In particular, most of the current case processing systems use a methodology based on objects in some way or other (i.e. reusable items of software which can be configured in different steps or processes and in different case types).

Ithaca seems particularly well suited for office work in public administration. Its object-oriented approach expands the domain of tasks candidates for automation. Applications in public administration are closely related to an activity coordination system and to the generic office model; these issues are covered in the Ithaca project. For public administration applications, activity coordination, the office model and even some of the tools may have to be customised or enhanced, even if most of the requirements seem to be considered.

Moreover there is a set of domain-specific objects, i.e. a means for signature recognition. Add-ons to general tools and domain-specific objects are the goal of the public administration workbench.

8. The Financial Workbench

The insurance scenario has been chosen as the target environment for the financial demonstrator. The insurance market is characterised by an high degree of competition, an aggressive sales strategy and a good disposal towards the use of innovative tools. In this scenario, one of the major problems is the management of a large amount of structured information, i.e. the products to be sold, strictly bounded together by relations and constraints. Therefore, there is a claim for a sale-support tool based on some kind of expert "products base" able to collect all the objects belonging to the applicative domain in order to increase the sales volume and the quality of insurance products.

The analysis of the applicative scenario - the major activity in the first six months - has been carried out together with NIKOLS, a brokerage company belonging to the Ferruzzi Group, the largest brokerage company in Italy. This cooperation has defined a particular applicative case study concerning the cross-sale of insurance and banking products. The framework for the demonstrator, identifying key issues and goals, has been specified and a general architecture for the application proposed.

The problem of integration of heterogeneous networks has been approached at application level, i.e. the focus is upon the integration of network-based services rather than upon low level communication protocols. This decision is based on the consideration that today a set of built-in services are available and, in general, it is more useful for application programmers to have access to these services directly within the programming languages than to able to work at lower network levels. The network applications considered are the services that can be accessed on-line on a local or wide area. In particular two main kinds of services are addressed: databases and electronic mailers. During the first half-year, the application scenario has been defined which points out the main problems to access and use services from the end-user point of view. Furthermore, the current state of the art has been viewed with specific reference to solution approaches adopted within EEC-sponsored projects and based on the object-oriented paradigm. A solution approach has been investigated and a general architectural proposal made.

9. The Partners, Organisation and Management of Ithaca

The Ithaca Consortium consists of the following partners:

Partners/Associated Partners	Role	CY
Nixdorf Computer AG	M	D
Universität Zürich	A	D
Trinity College Dublin	A	IRL
D-Tech	S	GR
Bull SA	P	F
Inria Rocquencourt	A	F
Delphi	A	I
CMSU	A	GR
Datamont SPA	P	I
Politecnico di Milano	S	I
Universita di Milano	S	I
Foundation of Research and Techn.	P	GR
TAO-Tecnics en Automatitzacio d'Of	P	E
University of Geneva	P	CH

The overall effort of the consortium is about 500 person years distributed over a five-year period. The main interest of the partners is the industrial dissemination of the results of Ithaca. Bull and Nixdorf were among the founding members of OSF. Dedicated components of the Ithaca environment will be submitted for standardisation in OSF.

The management of Ithaca consists of several boards. The Project Management Team (PMT) has the managerial and technical responsibility for the project as a whole. It consists of one managerial director from Nixdorf and three technical directors from Nixdorf, Bull and the University of Geneva. It reports to the Partner Coordination Committee (PCC) which represents the partners' interests in the project. The technical work is done in working groups which are led by coordinators.
The following working groups are established:

- the language group,
- the database group,
- the tools group,
- the user interface group,
- the activity coordination group,
- the office group,
- the chemistry workbench group,
- the public administration workbench group and
- the financial workbench group.

The working groups do not meet in isolation or on a general basis, but rather they exchange observers in order to discuss their respective requirements, to avoid double work and, in certain cases, to design their architectures in a joint effort.

With a view to integrating the work, the coordinators and directors formed a Technical Integration Board (TIB). The workbenches organised a separate user group to formulate their requirements and to evaluate the usefulness and the quality of the several Ithaca components at an early stage.

For the future, it is planned to set up a consultancy board in which major customer companies, well-known scientists and reputable management consultancy organisations will review the results of Ithaca. The respective viewpoints are the state-of-the-art nature, the market chances (European and non-European) and the adequacy of Ithaca as a product.

Clean Semantics of Multiple Inheritance

N Giambiasi
C Oussalah

Laboratoire d'Etude et Recherche en Informatique
Parc Scientifique Georges Besse
30000 Nmes - France

L Torres

ITECA
Centre ATRIA - 5, Bd de Prague
30000 Nmes - France

ABSTRACT

The ORL† elaboration led us to propose an inheritance model based on the definition of various graphs. Two versions of the language concern knowledge without conflict, called exact knowledge, and knowledge where conflicts may appear or inexact knowledge.

1. Introduction

Seldom formalized [Car84a, Dug86a], the handling of conflicts of multiple inheritance appears implicitly in a lot of works. It remains a crucial problem of object oriented programming. In some current works, the resolution of the conflicts is restricted to the definition of a priority path on the graph representing knowledge. Thus, the object inherits the characteristics of only one of its fathers. Other solutions consider that all the fathers intervene in the inheritance of the object, so defining an operation between the fathers. This operation, often called *fusion*, is particularly defined for the methods inheritance. Finally, some approaches prefer the conflict to be directly settled by the designer of the knowledge hierarchy.

The development of the language ORL [Orl88a], has led us to propose one of these latter approaches. ORL offers all the features of representation and programming with objects. Furthermore, the introduction of operators is going to allow the designer to define his own inheritance process.

After the definition of the various graphs useful to our approach, we will introduce the different operators the designer handles to make his choices when conflicts appear. First, we are going to discuss the main processes applied until now for resolving conflicts.

2. Previous works for resolving conflicts

The simplest and most commonly used techniques for resolving the conflicts of multiple inheritance are based on linearisation algorithms. They decide on a priority of a path of the knowledge graph, thereby defining the inheritance search process as a *class precedence list* [Bob86a]. They make inheritance dependent on the order in which the knowledg is defined. Among the most known processes, we can cite the (left most) depth-first search, adopted in some languages like Flavors [Min75a], LOOPS [Bob86a], and the (left most) width-first search, in the language MERING [Fer83a]. Some linearisation algorithms justified by a synthesis and a clear definition of the inheritance problems, are proposed [Duc89a]. Other techniques are based on mechanisms of viewpoint for removing the conflicts [Dug87a]. All these algorithmic solutions have the disadvantage of systematizing the processing of each conflict without taking into account the semantics of the involved properties. Thus it seems interesting to consider a technique which is a matter for the designer's expertise.

† ORL: Object Representation Language, object oriented language developped in LERI, inspired by frames [Min75a].

Indeed, languages exist where the inheritance process is defined by the designer. In the frame language KEE [Int90a], the designer specifies the resolution of the conflicts in a facet of the concerned slot. For this, he has at his disposal several operators: *override.values*, *union*, *runion*, etc. In CRL [Gro88a], the designer defines *inheritance specifications*, e.g. the inheritance semantics of the *relations*. The resolution of the conflicts may also be made by a *double filtering* to select the objects to inherit from and the inherited *fields*, through the *structural relation*. This makes the designer intervene only for the inheritance conflicts associated with this structure relation. This relation defines an object as an aggregation of objects. In this meaning, the composition relation, often called "is-a-part-of" or "component-of", is a particular structural relation.

The linearisation algorithms are not enough to resolve any conflict case in multiple inheritance [Duc89a]. Conflicts cannot be resolved without making the designer intervene. Our purpose is then to provide a general theoretic model, based on the definition of three graphs, which justifies the possibilities offered to the designer. This model concerns any inheritance type, without restriction: specialization inheritance or structure inheritance. In what follows, we use the term "attribute" to qualify an object property. The term refers to both a static property and a dynamic one (like a method).

3. Classification, Precedence and Execution Graphs.

Representing, by a graph, the hierarchy of objects describing knowledge, is usual. This is the *classification graph* or inheritance graph. It illustrates the inheritance relation existing between the objects. The inheritance semantics are no longer sufficient, as soon as there is multiple inheritance. Therefore, the conflict solving problem lies in defining operations associated with the classification graph. By analogy of the acquisition of an attribute with a solving task, a *precedence graph* of the attributes can be defined in the same way as a precedence graph of tasks. Any inheritance process is necessarily an *acceptable chronology* deduced from this precedence graph. Thus, we are led to define the *execution graphs* which describe the *chronologies* deduced from the precedence graph. We prove that the conflicts come from the fact that several chronologies are *syntactically acceptable*, and thus respect the precedence of the attributes. It is the designer who provides the semantics which allow the determination of the *semantically acceptable chronology*. This semantic aspect is introduced by the designer by means of operators. These considerations appear in Figure 1.

Afterwards, we adopt the notations given in Table 1.

a, b, c	attribute names
O_1, O_2, O_3, O_4	objects
a_{O_1}	attribute named "a" and defined in object "O_1"
$D_{O_1}(a)$	definition domain of the attribute "O_1"

Table 1: *Notations*

3.1. Classification graph.

The inheritance semantics affect two notions: the object from which the attribute is inherited, and the inherited attribute itself. For this reason, the inheritance model based on filtering [Dug86a] is composed of two filters, a filter of objects and a filter of properties. The conflict sources that we consider are induced by semantic ambiguities. The names of the objects representing knowledge are supposed to be unique. On the other hand, their *attributes* may be *homonymous*. Therefore, we are interested in the conflict associated with the attribute, and not with the object seen globally. These considerations immediately lead us to define four types of classification graph. Once we have isolated the case with no semantic ambiguity, noted case (α), we distinguish three other cases with semantic ambiguity. We are going to see how the definition domain of the attribute intervenes to distinguish these three cases, respectively called (β), (γ) and (δ). The simplistic classification subgraph given in figure 2, lets us define these four types. Only the crucial attributes are considered in order to simplify the representation: a and b represent attributes of the objects O_1 and O_2, respectively. O_1 and O_2 appear as the fathers of the object O_3.

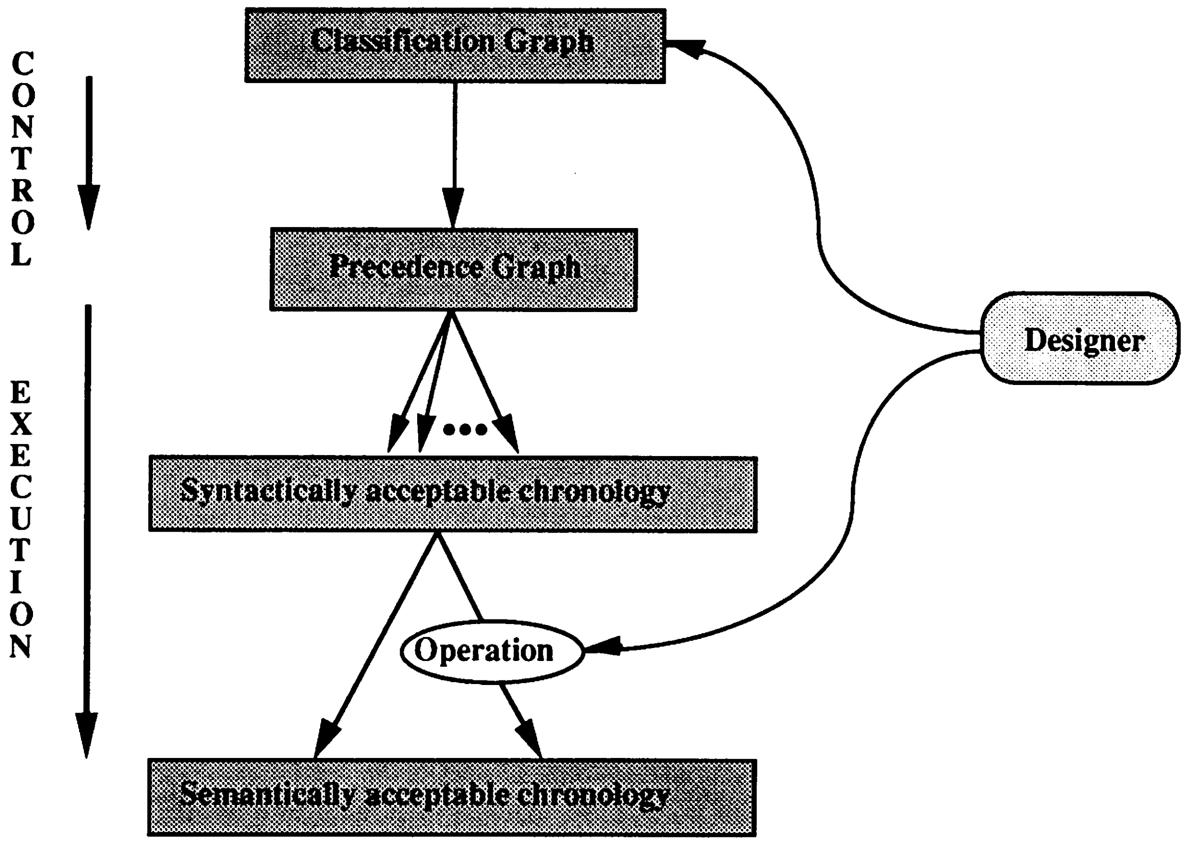

CONTROL
Classification Graph
Precedence Graph
Syntactically acceptable chronology
EXECUTION
Operation
Semantically acceptable chronology
Designer

Figure 1: *The stages of the approach*

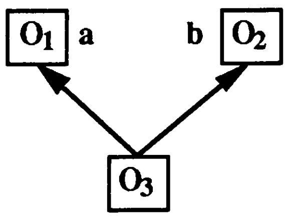


Figure 2: *A simple classification subgraph with multiple inheritance*

Four syntaxes follow from this classification graph. They are given in Table 2.

The semantics attached to these four syntaxes is immediate (see table 3).

• *the attributes are not conflicting* if they are heteronymous, or homonymous and compatible: cases (α), (β) and eventually (δ)
• *the attributes conflict* if they are homonymous and incompatible: cases (γ) and (δ).

Table 3: *The semantics of the classification graph*

Compatibility is defined from the definition domains of the attributes: domains, types, constraints, etc. The conflict is no longer limited to only the value of the attribute, but extended to the definition domain of this value. This notion is also encountered explicitly in some works, for instance, in defining the compatibility of the domains and the *cardinalities* by their non-zero intersection [Dug87a]. We can give another example with the LORE operator *herit* defined between two relations, in part by the intersection domains of these relations [Cas85a]. The last type (δ) has to be considered particularly for definition domains which are not syntactically comparable. For instance, this case concerns ORL for the validity constraints of the value of the attribute which are expressed in COMMON LISP terms.

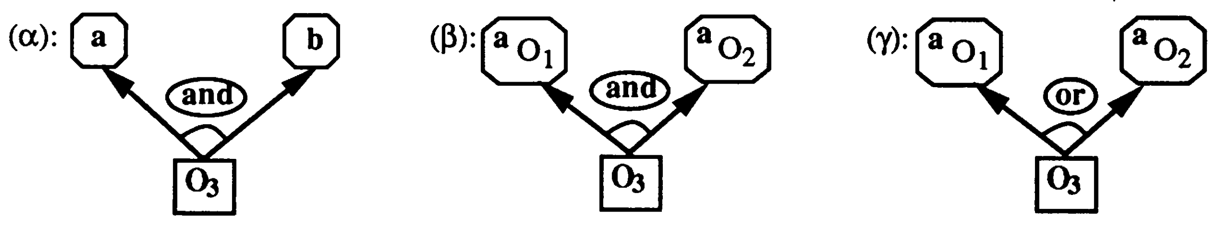
The classification graph of Figure 2 immediately conveys that the object O_3 is defined from the objects O_1 and O_2. The data constituting O_3 are easily conceivable from this graph: it is the reunion of the attributes of O_1 and O_2. On the other hand, the order and the way these attributes occur is imperceptible. We must build the precedence graph which will induce the various execution modes, from the classification graph.

(α)	a and b define to heteronymous attributes: $a \neq b$
(β)	a and be define two homonymous attributes and the definition domains of a in O_1 and of a in O_2, $D_{O_1}(a)$ and $D_{O_2}(a)$, are explicit, therefore comparable and compatible: $a = b$ and $D_{O_1}(a) \cap D_{O_2}(a) \neq \varnothing$
(γ)	a and be define two homonymous attributes and the definitions are explicit, therefore comparable and incompatible: $a = b$ and $D_{O_1}(a) \cap D_{O_2}(a) = \varnothing$
(δ)	a and be define two homonymous attributes and their definition domains are implicit, therefore incomparable: $a = b$ and $D_{O_1}(a) \cap D_{O_2}(a) = ?$

Table 2: *The four syntaxes of the classification graph*

3.2. Precedence graph.

We recall that the conflict only relates to the attribute. Thus, each attribute can be assimilated to a task which must be completed for the inheritance process of the object. The precedence graph renders the combination of these tasks. It expresses which attributes are useful to the inheritance process of an object. For the four previous cases, we give the corresponding precedence graph (see figure 3).



(δ): The precedence graphs (β) and (γ) are possible since it is not perforce a conflict situation. Information received from the outside has then to allow the choice to take place.

Figure 3: *The precedence graphs*

Semantically, the operator "and" which appears between non-conflicting attributes means that all the attributes are necessary for the inheritance process of O_3. On the other hand, all of the conflicting attributes may not be useful. Some of them will be selected in order to remove the conflict. This is expressed by the operator "or".

3.3. Execution graph.

The choice of a *sequential execution syntax* enables us to work out from the precedence graph *the chronologies* evoking the sequencement of the tasks. The possible chronologies with respect to the precedence graph are *the syntactically acceptable chronologies*. The execution graphs of Figure 4 represent the syntactically acceptable chronologies and follow on from the precedence cases.

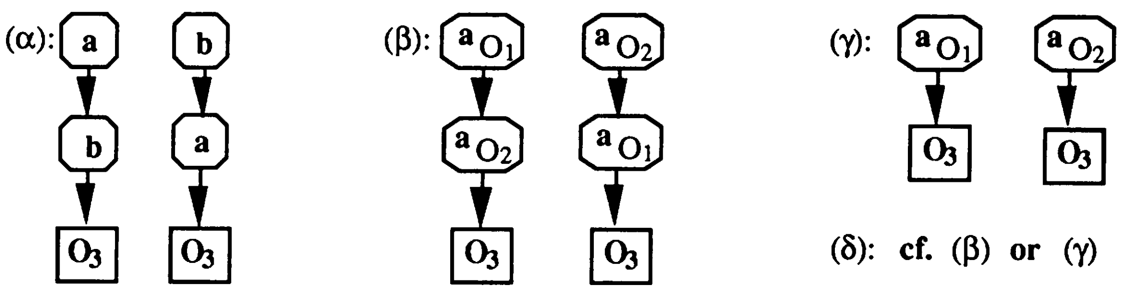


Figure 4: *The execution graphs*

The semantics for each case appear in Table 4.

	By language misuse, the same symbol designates here the name of the attribute and the attribute itself.
(α)	We are concerned here with *the union* of the attributes a and b. Since the union of sets is commutative, the two cronologies give the same result and are semantically acceptable: $O_3 = a \cup b = b \cup a$.
(β)	This time, the syntax induces *the intersection* of the definition domains of the homonymous attributes of O_1 and of O_2. The intersection of sets is commutative, therefore the two cronologies lead to the same result and are semantically acceptable: $O_3 = a / D_{O_1}(a) = D_{O_2}(a) = D_{O_2}(a) \cap D_{O_1}(a)$.
(γ)	The inheritance process of O_3 needs a *mutually exclusive*

Table 4: *The semantics of the execution graphs*

3.3.1. Operators

We arbitrarily choose the complete logic set of the operators $\wedge$ (conjunction) and $\neg$ (negation) to describe any execution mode previously mentioned (see table 5).

(α)	$a \wedge b$ or $b \wedge a$
(β)	$a_{O_1} \wedge a_{O_2}$ or $a_{O_2} \wedge a_{O_1}$
(γ)	$a_{O_1} \wedge \neg a_{O_2}$ or $a_{O_2} \wedge \neg a_{O_1}$
(δ)	This case concerns (β) or (γ)

Table 5: *The execution operations*

Every conceivable inheritance situation is in fact a combination of the different elementary cases we have just studied. Recognizing and applying these elementary solutions is enough to build the precedence graph from the classification graph, and then to deduce the chronologies.

By studying each case from the classification graph given in figure 2, we have built the precedence graphs, and then the execution graphs which have allowed us to define the operators useful to the designer. The progression we have just followed, is presented in figure 5. All the notions we have set out above are recapitulated there.

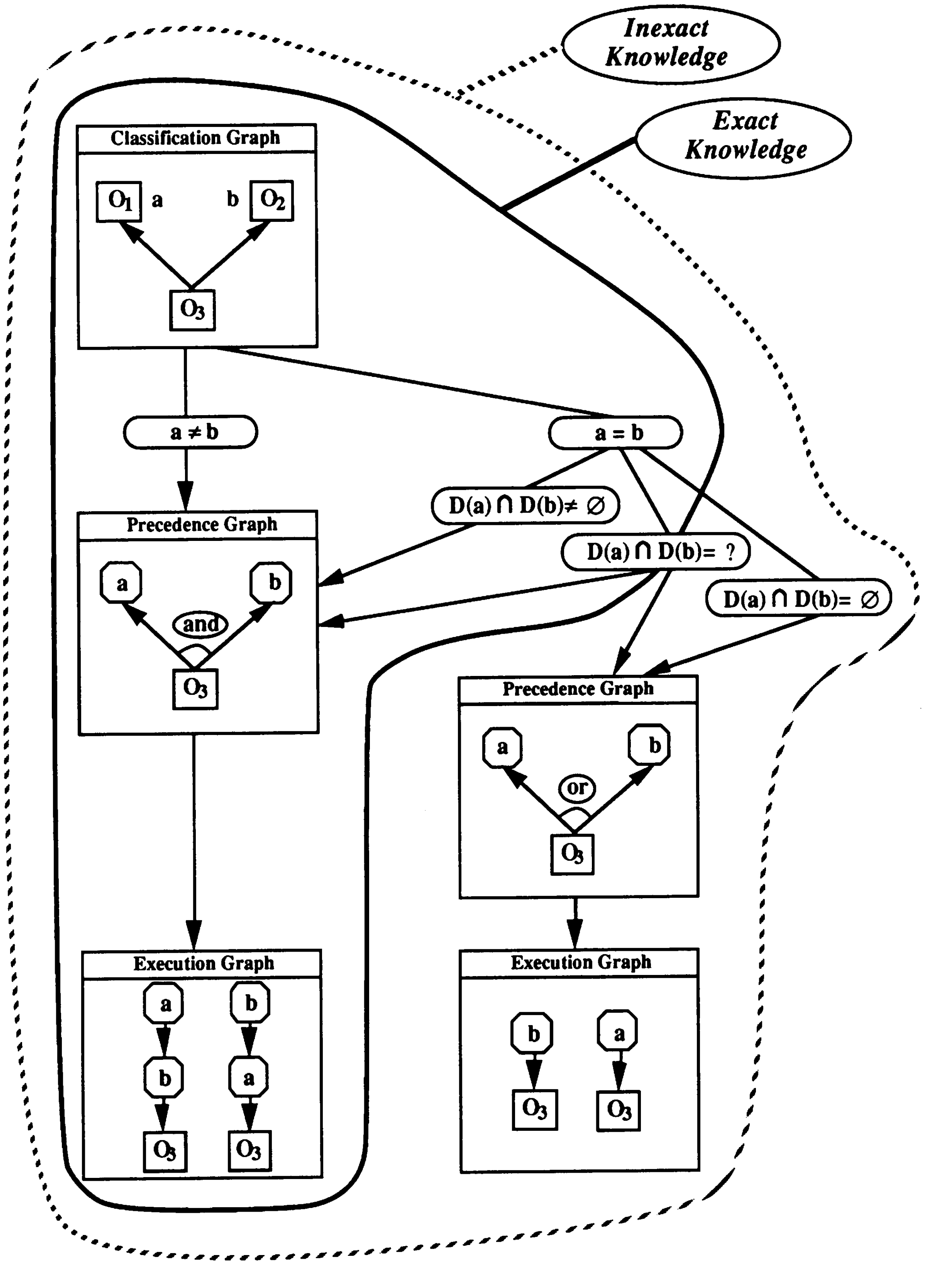


Figure 5: *Overview*

4. Exact Knowledge and Inexact Knowledge

The application of the previous results leads us to define two knowledge levels. The strictest level where knowledge is supposed to be coherent is the *exact knowledge* level (i.e. without conflict, according to Table 3). This is the default level. The *inexact knowledge* level allows the presence of conflicts and provides the designer with tools for handling these conflicts: the operators. Figure 5 shows the inclusion of exact knowledge handling in that of inexact knowledge.

The suitable precedence graph will be provided for the designer, depending on whether he chooses to work in exact or inexact knowledge. We are going to see that we can always get this precedence graph from the classification graph.

Let us consider the classification subgraph of figure 6. The homonymous attributes of objects comparable by the inheritance relation, are not in conflict since the attribute of the most specialized object masks the others. Thus, a of O_3 masks a of O_1. The conflict only exists for homonymous (and incompatible) attributes of incomparable objects, so for attributes b of O_2 and O_3 in figure 6 [Duc89a].

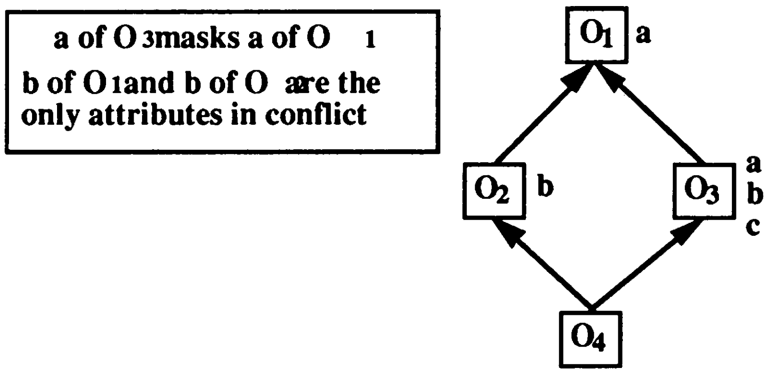


Figure 6: *A classification graph*

A new rule is applied, that we can express as the respect of the *modularity* of the object. The conflicting attribute continues to behave as a task; but the attributes of an object which are not conflicting with an attribute of another object, are all assimilated to a single task.

4.1. Exact knowledge

The knowledge type concerns cases (α), (β) and possibly (δ) of the previous chapter. The precedence graph corresponding to the classification graph of figure 6 is an *and graph* (see figure 7). It defines two chronologies which are syntactically and semantically acceptable.

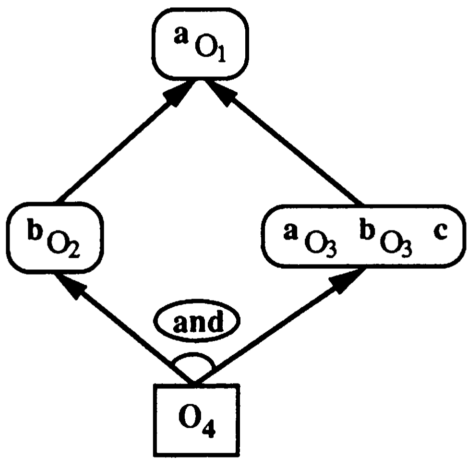


Figure 7: *The precedence grpah from the classification graph of figure 6, in exact knowledge*

We can prove that any syntactically acceptable chronology is semantically acceptable, in exact knowledge, by the commutative and associative nature of the union and the intersection of sets. Choosing one of them, which will be automatically applied by default, is sufficient. This obviously amounts to selecting a priority path on the classification graph.

4.2. Inexact knowledge

This time, cases (γ) and possibly (δ) of chapter 2 are concerned, and the precedence graph is an and/or graph (see figure 8). It defines sixteen syntactically acceptable chronologies. All of them are not semantically acceptable. Indeed, they may give a different result when executed. Designer intervention, by defining the semantics with an operation, allows to determine the chronology to be executed.

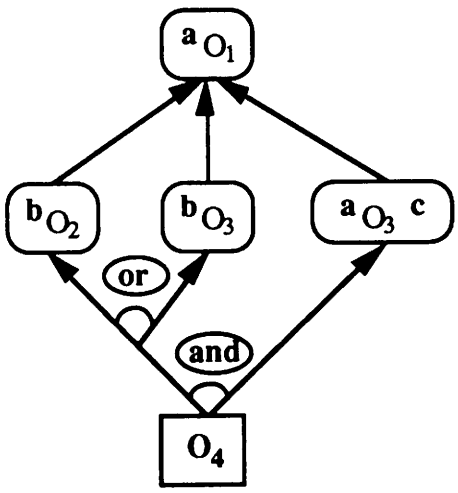


Figure 8: *The precendence graph from the classification graph of figure 6, in inexact knowledge*

The operation $a_{O_1} \wedge (b_{O_3} \wedge \neg b_{O_2} \wedge (a_{O_3} \wedge c))$ is a possible execution for the graph of the figure 8.

We would be able to prove that the precedence graph is unique. It can just as well be determined from the classification graph, in both exact and inexact knowledge. Moreover, the execution modes expressing the inheritance process respect the modularity of the classification graph according to R Ducourneau and M Habib [Duc89a].

5. Conclusions

The classification graph defined by the designer immediately induces an unique precedence graph. If the designer chooses to consider knowledge likely to allow conflicts, he is then free to define operations in order to resolve these conflicts. The system verifies that the operation symbolising the inheritance process to be executed, fully respects the precedence graph.

References

[Orl88a] "ORL," *Technical Report for EP2318 Esprit II EVEREST Project*, Nimes, France, September 1988.

[Bob86a] D G Bobrow and M Stefik, "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, vol. 6(4), 1986.

[Car84a] L Cardelli, "Multiple inheritance," *Semantics of Data Types*, pp. 51-67, Springer Verlag, 1984.

[Cas85a] Y Caseau, "Les objets et les ensembles de LORE," *Acts of 5th CARFIA*, pp. 101-118, Grenoble, France, 1985.

[Duc89a] R Ducournau and M Habib, "La multiplicite de l'heritage dans les langages a objets," *Technique et Science Informatiques*, vol. 1, 8, pp. 41-62, 1989.

[Dug86a] P Dugerdil, "A propos des mecanismes d'heritage dans un langage oriente objet," *Acts of the 2nd CIIA*, Marseille, France, 1986.

[Dug87a] P Dugerdil, "Les mecanismes d'heritage d'OBJLOG: vertical et selectif multiple avec point-de-vue," *Proceedings of 6th AFCET Congress*, Antibes, France, 1987.

[Fer83a] Ferber, "MERING: Un language d'acteurs pour la representation et la manipulation des connaissances," *These de Docteur Ingenieur*, Universite de Paris VI, 1983.

[Gro88a] Carnegie Group, *CRL Technical Manual*, Pittsburg, PA, May 1988.

[Int90a] IntelliCorp, *KEE 3.0 Training Manual*, Mountain View, California, 1990.

[Min75a] M Minsky and P Winston (ed), D Moon, and D Weinreb, "Flavors: Message Passing in the LISP Machine," *MIT AI Memo 602*, McGraw Hill, Cambridge, Massachusetts , November 1980.

BUD – Backtrackable UNIX Data Control

Zs. Hernath
M. Szokolov

Computer Centre
Eötvös Loránd University
Budapest
Bogdanfy u. 10/B
H-1117 Hungary

ABSTRACT

BUD is a C-interface resulting from research in database management that has been carried out at the Computer Centre of Eötvös Loránd University since 1986. Our original goal was to create a user-friendly low-level interface for developing higher level file systems to be used in a multi-user environment with transaction-oriented concurrency control. The interface was designed and implemented as a minimal extension of the low-level I/O and memory management facilities of standard System V. To test the system and to demonstrate concurrent processes running under BUD control, the BUD functions were also implemented as executable commands that can be activated from the menu of an interactive DEMO system. BUD is not an interface for the end user. It is a powerful tool for system and application programmers who implement data systems.

1. BUD Overview

The BUD definition specifies a set of C-functions that provide for manipulating data under a special user-defined control. The functions can be divided into three groups: control, memory and file operations.

Control operations provide for the building of a BUD control hierarchy within a running process. Under the BUD control the user can perform actions, transactions and chains of transactions. Actions are parts of transactions and can be nested. Chains can be nested too.

BUD actions can be undone. Undoing the effect of actions is based on the undo-sensitivity of most BUD objects (functions and memory blocks). BUD also provides the user with a method of making user-defined undo-sensitive functions.

Memory operations provide for manipulating memory blocks of different types according to the BUD control hierarchy. The user can create and manipulate data blocks (areas) as well as special blocks (queues and stacks) that provide for delaying the execution of functions.

File operations provide for the execution of the standard System V low level I/O under transaction-oriented concurrency control. The concurrency control is performed at two levels: between processes (external concurrency) and between nested transactions (internal concurrency).

Facilities are defined for the locking of files and segments of files. A whole file may be locked for shared, protected or exclusive use. A segment of a file may be read locked before reading or write locked before writing. Writing to files (by invoking *write*(2)) and unlocking locked segments are both delayed until the end of transaction.

2. Undo-sensitivity

A function is said to be undo-sensitive if the effect of its execution disappears when a BUD action containing the function call is undone. Undoing the effect of a function may not be perfect on the physical level, but it should always be perfect on the logical level represented by the function. The *link*(2) and *unlink*(2) system calls are good examples.

Most BUD functions are undo-sensitive.

The user can make undo-sensitive functions of his own by invoking undo-sensitive function calls, the *Fixundo()* function call and using BUD areas.

A memory block is said to be undo-sensitive if the effect of changing its contents or size within a BUD action disappears when the action is undone.

BUD areas, queues and stacks are undo-sensitive memory blocks identified by integer descriptors, not pointers.

3. BUD Control

The BUD control permits program instructions to be performed in blocks, created in the run time. BUD manages blocks of three types: actions, transactions and chains of transactions. These blocks together compose the BUD control hierarchy within a running process. The BUD control operations are as follows:

```
int Start()

int Finish()

int Release()

int Begin()

int End()

void Undo()

int Fixrelease (func, ptr)
int (*func)();
char *ptr;

int Fixundo (func, ptr)
int (*func)();
char *ptr;
```

A chain of transactions is a sequence of instructions beginning with a *Start()* and ending with a *Finish()* function call. *Start()* begins the first transaction within a chain and *Finish()* completes the last transaction. An intermediate *Release()* call completes the current transaction and begins a new one. A trivial chain consists of one transaction.

Chains can be nested. *Start()* beginning a nested chain suspends the chain currently running and the corresponding *Finish()* resumes it. The global integer *chlevel* indicates the level of the current chain within the chain hierarchy.

Within a transaction actions can be performed. An action is a sequence of instructions beginning with a *Begin()* and ending with an *End()* function call. Actions can be nested. The top of an action hierarchy is a transaction itself. In fact, a transaction is at the same time an action of level 0 which begins with an implicit *Begin()* invoked on a *Start()* or *Release()* function call and ends with an implicit *End()* invoked on a subsequent (within the same chain) *Release()* or *Finish()* function call.

An explicit *Begin()* call suspends the action currently running and the corresponding *End()* call resumes it. The global integer *actlevel* indicates the level of the current action within the action hierarchy in the current transaction.

At any time exactly one action, transaction and chain are running — those last started and not yet completed. They are the current action, transaction and chain respectively.

At any time the current action can be undone by invoking the *Undo()* function call. It means that the effect of undo-sensitive functions have been performed within the current action will disappear and the BUD resources (areas, queues, stacks and file descriptors) will be restored to the state in which they were when the current action began.

Each chain has its own queue and stack called release-queue and undo-stack. Both are made on *Start()* and removed on *Finish()*. Release-queue and undo-stack contain information about functions to be performed on a subsequent *Release()* or *Undo()*, respectively. Such information can be pushed onto a release-queue or undo-stack by invoking a *Fixrelease()* or *Fixundo()* function call.

4. Memory Operations

The BUD memory operations provide for manipulating BUD areas, queues and stacks. Areas, queues and stacks are undo-sensitive memory blocks identified by integer descriptors. Areas provide for storing data. Queues and stacks provide for storing information on functions to be performed later. A queue/stack is a proper fifo/lifo.

The BUD memory operations are as follows:

```
int mkarea (size, type)
unsigned size;
int type;

int mkqueue (type)
int type;

int mkstack (type)
int type;

char *areaptr (ardes, mode)
int ardes, mode;

long areasize (ardes)
int ardes;

long charea (ardes, diff)
int ardes, diff;

int perform (des)
int des;

int push (des, func, ptr)
int des;
int (*func)();
char *ptr;

int rmarea (ardes)
int ardes;

int rmstaque (des)
int des;
```

Mkarea() makes an area of specified type. *Size* is the size of the area in bytes. *Mkqueue()* and *mkstack()* make a queue or a stack of specified type. The available values for type are defined in the header file <bud.h>, which shall include:

B_ACTION	an area, queue or stack of this type can be accessed within the current action and will be automatically removed on the *End()* completing the current action.
B_TRANS	an area, queue or stack of this type can be accessed within the current transaction and will be automatically removed on *Release()* completing the current transaction.
B_CHAIN	an area, queue or stack of this type can be accessed within the current chain and will be automatically removed on *Finish()* completing the current chain.
B_GLOBAL	an area, queue or stack of this type can be accessed until the process has run under the BUD control and will be automatically removed on the very last *Finish()*.

Normally these functions return an area/queue/stack descriptor. The area assigned to this descriptor can be accessed through a pointer obtained from an *areaptr()* function call. The queue/stack assigned to the descriptor can only be manipulated by a *push()* or *perform()* function call.

Areaptr() returns a pointer to the area identified by *ardes*. If *mode* is B_RDONLY, the contents of the area only are supposed to be retrieved. If *mode* is B_RDWR, the contents of the area are supposed to be updated.

Areasize() returns the current size of the area identified by *ardes*.

Charea() changes the size of the area identified by *ardes*. The new size is the sum of the current size and the value of *diff*. No change takes place if the value of the new size is negative or zero. The contents of the area will be unchanged up to the lesser of the new and old size.

Perform() performs the queue or stack identified by *des*. A queue or stack consists of items pushed by *push()* calls. An item contains two pointers:

```
int (*func)()
char *ptr
```

where *func* is a pointer to a user-defined function and *ptr* is a null pointer or a pointer obtained from a *malloc*(3) or *calloc*(3) call.

Performing an item requires the following call:

```
(void) (*func) (ptr)
```

A queue/stack is performed item by item as a fifo/lifo. After performing an item, a memory block pointed to by *ptr* is freed. *Perform()* performs only the section of the queue or stack that was pushed within the current action and then truncates the performed section.

Rmarea()/rmstaque() makes the *ardes/des* inaccessible. Memory block(s) assigned to the descriptor will become free on a subsequent *Release()* completing the current transaction.

Memory operations with the exception of *perform()* are undo-sensitive.

5. File Operations

File operations provide for the execution of the low level I/O under transaction-oriented concurrency control. Most of these functions are BUD extensions of well-known Unix system calls. The BUD file operations are as follows:

```
int toopen (path, toflag [,mode])
char *path;
int toflag, mode;

int toreopen (fildes, toflag)
int fildes, toflag;

int tocreat (path, mode)
char *path;
int mode;

int toclose (fildes)
int fildes;

int toread (fildes, buf, nbyte, lockmode)
int fildes;
char *buf;
unsigned nbyte;
int lockmode;

int towrite (fildes, buf, nbyte, lockmode)
int fildes;
char *buf;
unsigned nbyte;
int lockmode;

int toappend (fildes, lockmode)
int fildes, lockmode;

long tolseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

int tolink (path1, path2)
char *path1, *path2;

int tounlink (path)
char *path;
```

The arguments are similar to those of the corresponding system calls.

Additional flags are defined for *toopen()* to specify a file level lock. A file may be (to)opened for shared, protected or exclusive use. If the specified lock cannot be set, *toopen()* will fail immediately. A file maybe (to)opened more than once, with different file level locks and usage modes. The external concurrency control takes into account the strongest usage.

Toreopen() is not equivalent to the *dup*(2). It provides for setting a new file level lock and defining a new usage mode for a file already (to)opened by a process and opens a new BUD file descriptor.

Tocreat() exclusively locks the file by default.

A BUD file descriptor opened by *toopen()*, *toreopen()* or *tocreat()* function call is a chain local object. It will be automatically closed on a subsequent *Finish()* completing the current chain, unless an explicit *toclose()* is performed. The *Finish()* will then free the system resources (including the reset of the file level lock) assigned to the file descriptor.

Toclose() closes a BUD file descriptor opened by a *toopen()*, *toreopen()* or *tocreat()* function call. *Toclose()* can be performed only at the same chain level as a corresponding *toopen()*, *toreopen()* or *tocreat()* call. System resources assigned to the file descriptor will become free on a subsequent *Release()* completing the current transaction.

Toread() and *towrite()* have an additional argument that provides for specifying locking information. The available values for lockmode are defined in the header file <bud.h> which shall include:

B_SETLK	The segment to be read/written should be read/write locked. If the lock cannot be set, *toread()/towrite()* will fail immediately.
B_SETLKW	This flag is the same as B_SETLK except that if the lock is blocked by other (external) locks, the process will sleep until the segment is free to be locked. On internal lock conflict *toread()/towrite()* will fail immediately.
B_TESTLK	No lock will be set. If a lock exists that would prevent read/write locking of the segment, *toread()/towrite()* will fail immediately.
B_NOLOCK	*Toread()/towrite()* will act without reference to locking.

Toread() and *towrite()* attempt to read/write lock the segment of a file to be read/written. Upon successful locking, *towrite()* stores the data to be written in a buffer that will be emptied on a subsequent *Release()* completing the current transaction. *Toread()* invokes the *read*(2) or uses buffer(s) created by *toread()* or *towrite()* calls invoked earlier within the current transaction.

Toappend() exclusively locks the segment of a file beyond the end of file and sets the BUD file pointer to the end of file. The available values for *lockmode* are B_SETLK and B_SETLKW.

The locks set by *toread()*, *towrite()* and *toappend()* calls will be automatically removed (reset) on a subsequent *Release()* completing the current transaction or on an occasional *Undo()* undoing a BUD action that contains the function calls.

Tolseek() is functionally equivalent to the *lseek*(2).

Tolink() creates the new link immediately.

Tounlink() delays the execution of the *unlink*(2) until a subsequent *Release()* completing the current transaction.

File operations are undo-sensitive.

6. BUD Programming

To demonstrate the use of BUD we present two examples. Both illustrate the method of making undo-sensitive functions. Note that the undo-sensitivity of the functions will be in effect until a subsequent *Release()* completing the current transaction. These examples also show that BUD supports the programming of atomic and reliable components.

```
#include <sys/bud.h>

        int     tolink (path1, path2)
        char    *path1, *path2;

/*
 *      Attempts to accomplish an undo-sensitive link(2).
 */

{
        char    *cp;
        void    free();
        int     rmpath_on();
        char    *save_path();

        if (link (path1, path2) == FAULT)      return (buderrno=errno, FAULT);
        if (!(cp = save_path (path2)))         return (unlink(path2), FAULT);
        if (Fixundo (rmpath_on, cp) == FAULT)  return (unlink(path2), free(cp), FAULT);

        return (SUCCESS);
}

static  char    *save_path (path)
        char    *path;

/*
 *      Attempts to allocate a memory block of size PATH_MAX+2 and stores the full
 *      pathname corresponding to the given path in the block.
 *      The memory block will be automatically freed on subsequent Release() or Undo().
 */

{
        char    *cp;
        void    free();
        char    *getcwd(), *malloc(), *strcpy();

        if (!(cp = malloc (PATH_MAX + 2)))      buderrno = ENOMEM;
        else if (path [0] == '/')               (void) strcpy (cp, path);
        else if (!getcwd (cp, PATH_MAX)) {
                                                free (cp);
                                                buderrno = errno;
                                                cp = NULL;
        }
        else                                    (void) sprintf (cp+strlen(cp), "/%s", path);

        return (cp);
}

static  int     rmpath_on (cp)
        char    *cp;

/*
 *      Attempts to unlink the pathname stored in a memory block pointed to by cp.
 */

{
        if (unlink (cp) == FAULT)       return (error_on_perform = TRUE, FAULT);

        return (SUCCESS);
}
```

Example 1. *This example shows a possible implementation of the tolink() function.*

```
#define BEGIN   if (Begin() == FAULT) return (FAULT)
#define END     (void) End()
#define UNDO    Undo(), END

int atomic_function()
{
  BEGIN;
        if (f1() == FAULT)    return (UNDO, FAULT);
        if (f2() == FAULT)    return (UNDO, FAULT);
           ...
           ...
           ...

        if (fn() == FAULT)    return (UNDO, FAULT);
  END;
        return (SUCCESS);
}
```

Example 2. *This example shows a typical program structure we used in the implementation of higher level file systems. The functions f1(), f2(), ... , fn() are undo-sensitive.*

The Educational On-Line System

Nick Williams

Imperial College
University of London
UK

njw@doc.ic.ac.uk

William Cattey
Robert French
Bruce Lewis

Massachusetts Institute of Technology
USA

wdc@athena.mit.edu
rfrench@athena.mit.edu
brlewis@athena.mit.edu

ABSTRACT

The Educational On-line System (EOS) is a suite of programs which enables students and lecturers to exchange course materials within the Project Athena environment at the Massachusetts Institute of Technology in Cambridge, MA. Early versions of this software used a command-line interface. This paper outlines the implementation of a user interface to the file exchange system, using the Andrew Toolkit (ATK) to produce an X Window System application.

1. Introduction

For the past two years, the MIT Writing Program has held some classes in the "electronic classroom", where twenty workstations are connected to the campus network. In a typical session the instructor might present a letter as an exercise for the students to take, rewrite and turn in. The professor can then hold a review session at his workstation, giving students immediate feedback. In a traditional classroom setting, students would have to go home, type and photocopy their letter, and shuffle papers around for the first ten minutes of the next class session.

Electronic means of file exchange have been available for a long time. When Project Athena, MIT's experiment in integrating computers into education, still used timesharing machines, there was a *turnin* program that was no more than a shell script that piped *tar* output into *rsh*. This meant that the instructor had to be UNIX literate to do anything with the files.

Another interface was introduced in the 1987–88 school year that used the Sun Network File System. UNIX file protection modes determined who could access which files. The user interface, when taking inventory of files turned in, had to do the equivalent of a *find* over the remote file system. As the number of files grew, this operation got more and more expensive.

This NFS-based system was more sophisticated than than the original turnin. It allowed the submission, editing, and pickup of files. It was designed to support a classroom paradigm for files, used by students and teachers who were not necessarily computer literate. Files were edited with GNU Emacs and exchanged either by a simple textual-command oriented interface, or from within the editor itself.

With the introduction of the Andrew Toolkit [Pal88a], which contains multi-media editing facilities combined with an Emacs-like command set, it was decided to create an application that was more user friendly. A design was produced by Hagan Heller [Hel89a, Bar87a], from which the new system was implemented.

This paper describes the latest version of the Educational On-line system, which aims to bring both the interface and the file exchange mechanism closer to the requirements of its users. [Env86a].

1.1. Hierarchy of Papers

The UNIX concept of a *file* is not designed to deal with the different states a paper evolves through in a classroom situation. The file exchange system provides a classroom–oriented paradigm for organizing submitted files into divisions of courses and types of paper. It is common to refer to files stored in the file exchange server as *papers*. The types of paper are:

TURNEDIN
: This is a paper which has been turned in by a student, but has not yet been graded.

TAKEN
: The paper is currently being graded.

GRADED
: The instructor has finished grading the paper and it is now available for the student to collect.

PICKEDUP
: The student has retrieved the graded version of the paper.

HANDOUT
: The paper has been submitted by an instructor for general use by the class.

EXCHANGE
: This type of paper may be submitted by any user and is accessible to the entire class. This enables groups of users to exchange papers among themselves, informally, for review.

These types determine who is allowed to read and/or write the various files. Graders have access to everything. Students are not allowed to write handouts or read other students' non-exchange files. The next section describes the flow of papers through these different types.

1.2. The Life Cycle of an Assignment

The student creates a paper in one of two ways: he or she produces one from scratch or, edits a copy of one of the available *HANDOUT* papers.

While editing, the student may submit the paper several times, using the *EXCHANGE*, facility, to allow other students in the same course to comment and discuss their work as it evolves.

After the student has finished editing his or her assignment, the paper is turned in for grading. The paper would have a status of *TURNEDIN*.

When the instructor finally chooses to grade the paper, he or she selects the paper from the list and the paper changes its status to *TAKEN*. After making annotations, the instructor returns the edited paper to its author, and the paper would become *GRADED*.

Finally, the student, who originally submitted the paper, would request to see if any papers had been returned from grading. The student would see this paper in the list and, after collection, the paper would change its status to *PICKEDUP*.

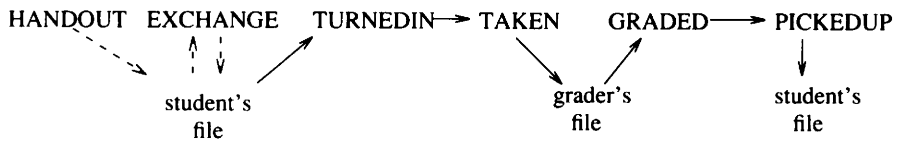


Figure 1: *The path and states of a paper in the system*

1.3. Overview

The interface can present two different views depending on the user of the program. The student can run the command *eos,* which provides functions for turning in papers to be graded and retrieving marked papers from the instructor. The instructors can type the *grade* command, which provides functions for reviewing which papers are available for grading, editing and annotating submitted papers, and then returning papers to their authors.

Two functions are common to both interfaces. These are represented by the buttons **Help** and **Guide.** The **Help** button invokes the Andrew help system in a fresh window starting at the help page relevant to the interface being used, and the **Guide** button creates a new window from which users can view the "Style Guide". A hypertext document containing advice on how to create a well-written document.

These interfaces rest upon the file exchange client library, a collection of functions to facilitate interaction with the file exchange servers, which in turn control access to course-related files by maintaining a database of courses, grades, students and files [Mar88a].

1.4. The Andrew Multi-Media Toolkit

The Andrew Toolkit solved a central problem in exchanging and annotating papers. It did so in a way that no other application, or toolkit could match. The problem is: How can a grader, looking at a complex formatted document, make annotations that are easily related to the paper which the author composed? The best way is to integrate the annotator, the formatter, and the editor. The grader sees the formatted document, and places notes in it where they are relevant. The author views the annotations in the formatted display, but receives a revisable document. The annotations themselves can easily be deleted, and their ideas incorporated into later revisions.

The Andrew Toolkit offers not only a multi-font text object and a text editor built on it, but also other media types such as line drawings, spreadsheets, and equations. It also offers the ability to add new media types by just compiling a new media module and putting it into the library of dynamically loaded modules. This was the method used to create a new media type of *note* which is used to annotate papers. Student authors can produce either simple text, richly formatted text documents, or complex documents containing text, equations and line drawings.

Having an annotator integrated with a WYSIWYG editor and then integrating that with a file exchange mechanism provides a synergy of functionality.

2. The User Interface

Normally, users would be editing text, but the facilities are provided for the user to edit raster images, animations and any other Andrew Toolkit object, all of which can be sent through the file exchange system.

The editing system is very similar to that of GNU Emacs, having the same type of buffer system in which multiple files can be stored and edited simultaneously. It also uses many of the same keyboard commands.

2.1. Exchange Papers

From the definition, an exchange paper has to be accessible from both the grading interface and the student interface, so that all groups can access the papers. To this end, both interfaces contain a button, labelled **Exchange,** that allows access to these papers. When this button is pressed, a new window is created that contains a list of the current exchange papers and also a set of buttons with which the user can submit papers of their own and take copies of other papers. Authors are permitted to delete their own papers by selecting a menu command that attempts to remove all the selected papers.

The list of papers describes each paper by giving the author and the title, both of which are defined when a user submits a paper for the first time. When a user wishes to edit or examine another paper, he or she first selects a range of papers using the mouse, and then clicks on the required function, listed below:

- Print, which sends a copy of the paper directly to the printer,
- Edit, which copies the paper into a file belonging to the user and immediately prepares the editor for editing the file.
- Save, to take a private copy, but not to do anything with the saved copy just yet.

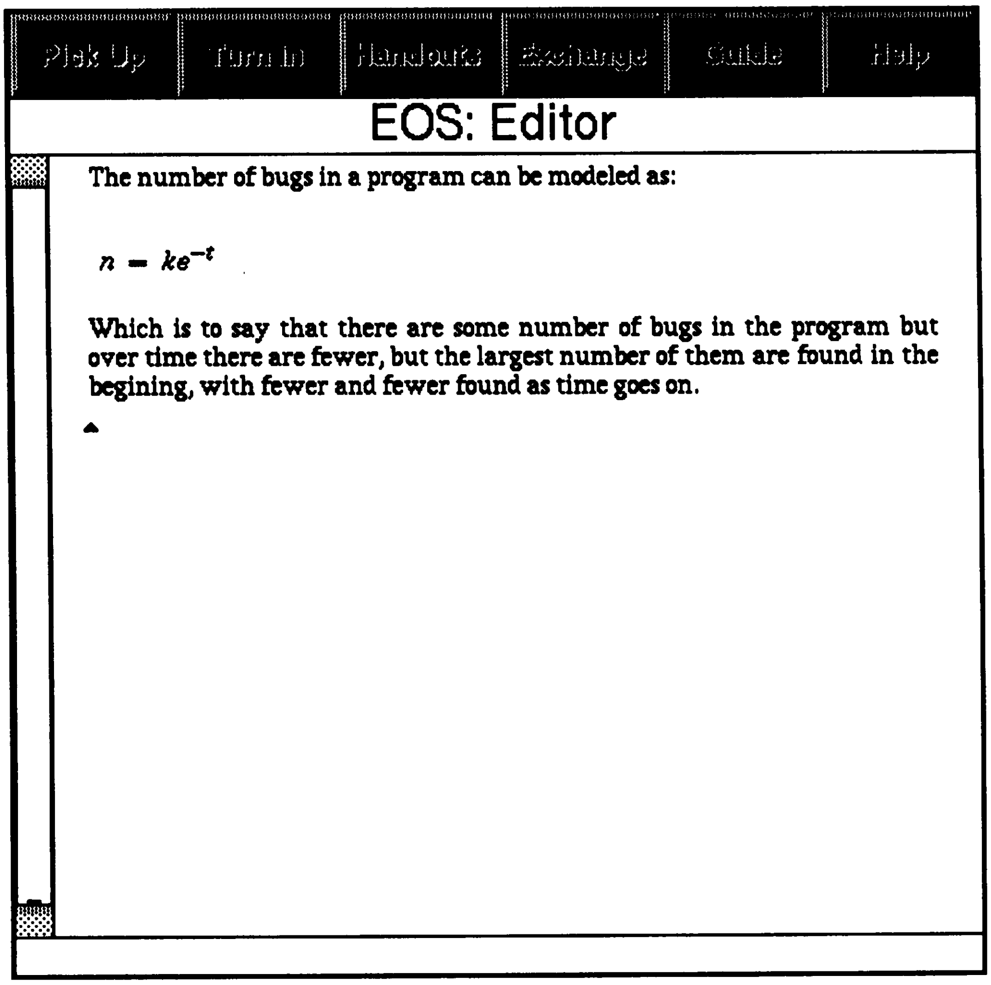
The same method of showing a list of papers and providing buttons for performing actions on those papers is used when looking at handouts and when an instructor is selecting which papers to grade.

2.2. The Style Guide

This is a document accessible from both interfaces, that contains information and hints on how to write a good paper. The document is designed in such a way that it is internally cross-referenced by use of the hyperlinks provided in the Andrew Toolkit. These allow a user to click on a labelled button that resides in the text and cause a new window to appear containing another document.

2.3. The Students' Interface

The student's window contains a row of six buttons and an editing region (see figure 2). The major functions provided in this interface are for submitting papers to be graded and for retrieving marked papers.



EOS: Editor

The number of bugs in a program can be modeled as:

$n = ke^{-t}$

Which is to say that there are some number of bugs in the program but over time there are fewer, but the largest number of them are found in the begining, with fewer and fewer found as time goes on.

Figure 2: *The Student's Interface*

2.3.1. Turning in assignments

To turn in an assignment, the student has a button labelled **Turnin** which pops up a new window asking for the details of the assignment to be turned in. The user can specify the name of a file to submit or use the contents of the buffer currently being edited. The student must also supply a number for the assignment that is used to categorize the papers for the graders.

Once the user is satisfied with all the details, he or she presses an **OK** button to submit the assignment.

A **CANCEL** button is also provided which allows the user to abort the submission procedure at any time before the **OK** button is pressed.

2.3.2. Collecting Assignments

To collect assignments, the student has a **Pickup** button. When this button is selected, a new window is created with the list (if any) of papers which the student can collect. The student can then select which papers for EOS to collect. When a paper is received, as well as placing it into a file, it is also placed into a buffer so that the student can view the file immediately.

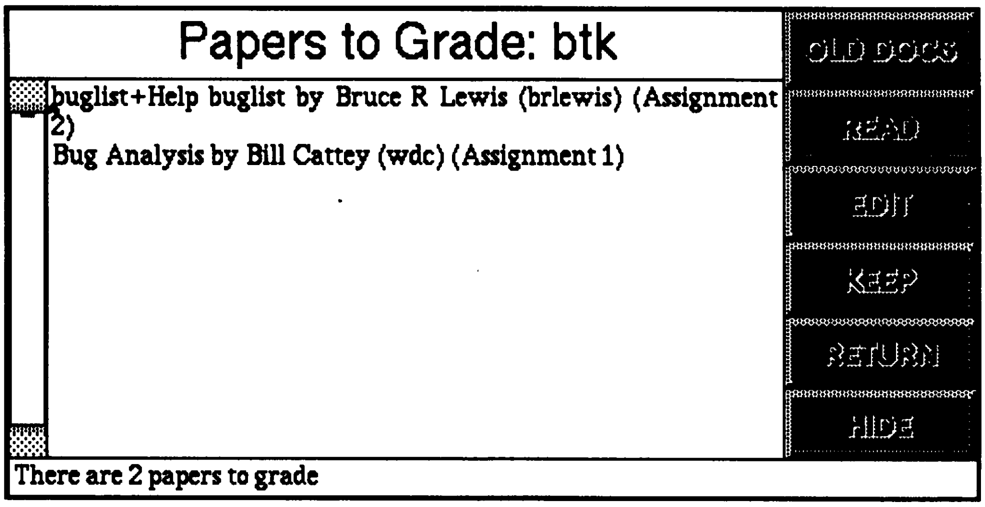
2.3.3. Picking up Handouts

If the user should want to collect any of the course handouts, then he or she would use the **Handouts** button, which creates a window of the same style as that used by the Exchange papers, although not providing the student with the facility for submitting papers. The student can then select any papers he or she wishes, and collect them for printing or editing.

2.4. The Lecturer's Interface

The lecturer's interface is very similar to the student's interface. As with the student interface there are buttons for Help and the Style Guide. The Exchange operation is exactly the same for both the lecturer and the student. The Handouts interface contains the additional **Submit** operation so that the lecturer can make new handouts available. This functionality is analagous to the **Submit** operation in the Exchange window.

In place of the **Turnin** and **Pickup** buttons, a single button **Grade** is present. The Grade window, produced on selecting this button, lists all papers that have been turned in. It allows the grader to select papers, annotate them, and return them to their authors.



Papers to Grade: btk

buglist+Help buglist by Bruce R Lewis (brlewis) (Assignment 2)
Bug Analysis by Bill Cattey (wdc) (Assignment 1)

There are 2 papers to grade

Figure 3: *The Grading window*

2.4.1. Annotating

Using the multi-media aspect of the Andrew ez editor, a *note* object was created. There are three menu commands appropriate to the note available within ez:

- "Add Note": To insert an annotation at the current cursor position.
- "Open Note": To display the text associated with the note icon currently selected.
- "Close Notes": To close all the annotations and show only the icons.

To annotate a paper, the grader clicks **Edit** from the grading window, and clicks the mouse to specify where in the paper the note should be placed. The grader then selects **Add Note** from the ez "Page" menu card. A small icon looking like two tiny pieces of paper, one on top of the other, appears. By clicking on the icon, the note is opened and the grader can type annotations. Clicking in the black region at the top of the note closes it.

When a student has picked up an annotated paper, he or she selects **Open Notes** from the ez "Page" menu card. Then he or she scrolls through the document reading all the notes.

3. How The File Exchange Library Works

The File Exchange Client Library (or, for short, the FX library) helps C programmers write applications which imitate the paper handling done in a classroom by providing procedures for storing and retrieving of papers.

Additional Procedures are provided to allow creation and modification of access control lists, to control who has grader permission and the ability to modify turned-in student papers.

The procedures were designed to balance three ideas: simple clients, supportable by a remote procedure call protocol, a simple server.

The "simple clients" idea means that it should be easy to write client programs that send and receive papers. This idea has been successfully captured in the FX library: a client program uses *fx_open()* to open a course, *fx_retrieve()* to get a file, *fx_send()* to send a file, and *fx_close()* to close a course. Access control lists are retrieved with *fx_acl_list()*, added to with *fx_acl_add()*, and deleted from with *fx_acl_delete()*.

Clients also need the ability to operate on lists of papers. These lists could be parameterized by author, paper name, or assignment number. A specific parameter such as **author "wdc"** can be named, or "*" for all authors can be named. The protocol defines a data structure called *paper* which holds each of these elements. The *fx_list()* operation returns lists of these structures that tell which specific papers are in the server that match the given paper parameters. The *fx_send()* and *fx_receive()* operations use the paper structure to identify which papers to copy.

The "remote procedure call protocol" enables the construction of a library that does its work on a remote host over the network. To the author of the client program, the library behaves as if everything is done locally, but it does look a little strange. Why would anyone use *fx_send()* when a simple UNIX *fwrite()* would do? The answer is that the additional complexity was added to make it a fairly simple matter to perform the operation over the network. The Athena File Exchange Service is written using the Sun Remote Procedure Call library. All data exchanged between the client programs and the server had to be translated into an external data representation that could be sent over the network. The Sun XDR (for eXternal Data Representation) generator took care of writing the necessary data translation routines.

The "simple server" idea is embodied in the file system and the database. The file system is the UNIX file system. Papers are stored in ordinary UNIX files. Lists of files belonging to particular authors and assignments are generated by the database. Since the server must provide service to several courses, and potentially hundreds of students, it must do the minimum amount of work.

3.1. Authentication

In order to reliably enforce access control, some method must be used to confirm that a user is who he or she says they are. This is done through the *Kerberos* authentication system. When a client issues the *fx_open()* call, the client side of the FX library assembles a request and calls the kerberos library to create an authenticator. This authenticator is then sent over the network to the server. When the server needs to make sure this user is who he or she says they are, it takes the authenticator and calls the kerberos library to confirm that it is a valid authenticator for that user.

3.2. Where To Find The Servers

The client programs, *eos* and *grade* need to be able to find and connect to the file exchange servers. How this information is specified affects load balancing and server cooperation. The server location is currently specified in one of two ways: an environment variable **FXPATH,** or through the Hesiod name service. For our limited work, having everyone in the course set an environment variable to lists the server hosts turns out to work. When we go to having many more users, the service location information is available with as much ease as one would ask for a host number to name translation through *bind.*

3.3. Server Co-operation

We are experimenting with cooperating servers. The previous version of the File Exchange service relied on NFS to store papers. The load was divided by putting only a few courses on any single NFS server. The problem was that there was only one NFS filesystem per course, so if a particular NFS server was down, those courses were out of luck. For the File Exchange service to be fully operational for all of MIT meant that all the relevant fileservers had to be running. This proved to be a strain on the operations staff near the end of the term. The new File Exchange service has a dedicated database to speed up list generation, so more courses can live on one server, and fewer servers can serve the same number of people. But we expect the need to arise for a degraded mode of operation where some servers are allowed to go down.

The current method of cooperation is to replicate the database of papers, but not the papers themselves. All servers that are up agree to hold a copy of the database. As long as the majority of the servers are up there is no problem. When a server realizes there is no longer a majority of servers running, it politely refuses to do any work until there is once again a quorum.

Once we are satisfied that the replication, synchronization, and quorum identification code is working, we will establish mechanisms for load sharing, graceful handoff to secondary servers, and clear explanation to lecturers of how to get papers that are on a server that is temporarily down.

Acknowledgements

Bruce Lewis and Rob French designed and implemented the file exchange libraries and servers. Nick Williams implemented the user interface from the design of Hagan Heller, and also wrote this paper with Bill Cattey, who had to endure coordinating the lot of us.

References

[Bar87a] E. Barrett, F. Bequaert, and J. Paradis, *Electronic Classroom: Specification for a user interface*, Athena Writing Project, Boston, MA, 4 June 1987.

[Env86a] The Committee on Writing Instruction and Computer Environment, *The requirements for the writing instruction computer environment*, 21 October 1986.

[Hel89a] N. Hagan Heller, *Designing a User Interface for the Educational On-line System*, Massachusetts Institute of Technology, Boston, MA, May 1989.

[Mar88a] Maria Camblor, Rob Shaw, Bruce Lewis, William Cattey, *File exchange for the educational on-line system design specification*, Athena Writing Project, Boston, MA, February 1988.

[Pal88a] Andrew. J. Palay, Wilfred J. Hansen, Mark Sherman, Maria G. Wadlow, Thomas P. Nuendorffer, Zalman Stern, Miles Bader, and Thom Peters, "The Andrew Toolkit - An Overview," *Proceedings of the USENIX Winter Conference*, pp. 9–21, USENIX Association, Berkeley, CA, February, 1988.

Dynamic Driver Loading for UNIX System V

Dieter Konnerth
Elmar Bartel
Oliver Adler

/dev Software, Munich
dieter@idefix.uucp

ABSTRACT

Dynamic Driver Loading (DDL) is a concept which makes it possible to add, remove and replace driver modules from the running UNIX System at any time. The paper presents this concept, related concepts and the design goals of our implementation for System V.3.2 386. Details of the implementation are explained. In the second part of the paper we focus on problems which we met in the course of implementation and use of the system. The problems appear to be partly due to the very basic design of the kernel/driver interface, but most are due to breaking the rules of this interface. We conclude that the system is an invaluable tool for driver development, that its usage forces developers to write clean drivers, which improves the quality and portability of driver code, and that the concept is usable in production systems if the drivers are carefully designed.

1. Introduction

1.1. Unix Device Driver Interface

UNIX device drivers implement the interface between the UNIX kernel and the hardware of a system. While most of UNIX is not hardware dependent, device drivers are. The portability of UNIX stems from this division. Therefore the interface between the UNIX kernel and device drivers was, and still is, of major importance to the development of UNIX.

This interface, as well as all other interfaces inside the UNIX kernel, is a functional interface, as opposed to more recent, message based, approaches. The interface has barely changed since the very beginnings of UNIX, it is still almost the same as it was in UNIX Version 7. This proves it to be a well designed and flexible concept. The newer concept of STREAMS drivers and modules has been added in System V Release 3. For System V.4 the interface has been slightly reworked, but it still has the same fundamentals.

One of the major disadvantages of this functional interface is that, at runtime, UNIX is a large monolithic object which resides unchangeable in main memory.

Our goal was to provide a tool which makes it possible to add and remove modules of the UNIX kernel at any time.

1.2. Dynamic Driver Loading

The idea of Dynamic Driver Loading is straight forward: A driver is loaded when it is needed, at runtime. The loading is triggered either explicitly by a utility, *drvload*, or implicitly, by an *open()* system call on the device special file representing the driver.

When a driver isn't needed any more, it can be unloaded explicitly by the command *drvunload* or, optionally, it is automatically unloaded by the last *close()* of the device.

A driver can be updated by unloading it, replacing the driver object file and then loading it again.

1.3. Related Concepts

The *Installable Drivers* feature introduced with UNIX System V Release 3.2 enables a user (system administrator) to configure/unconfigure drivers and rebuild the kernel by a set of easy to use commands. The configuration process still consists of generating and compiling configuration files, relinking the kernel, and rebooting the system.

A couple of UNIX implementations use boot time auto configuration. That means the booter recognises what hardware is present and includes driver code based on that information.

The concept used in MS-DOS is similar. Here the configuration description resides in a file, *config.sys,* and the operating system kernel includes the drivers configured by itself after booting. Also, a driver can insert itself in the system by staying resident and patching the interrupt table.

Paging UNIX Systems (all 32 bit systems are) traditionally can only page the user space. Also paging the kernel space is the latest attempt to cope with the continuously growing demand for memory from UNIX systems. This concept is the most powerful with regard to memory saving, but its implementation bears some nearly insolvable problems. It also does not speed up the process of updating driver code.

Recent operating system architectures (Chorus, Mach) have loosely coupled kernel architectures. On top of a small kernel, reside processes which implement file systems, pager, device drivers, and so on. The interface between these modules is message based. This is the most elegant approach, but we will have to live with UNIX for some time.

The *STREAMS* framework allows building specific driver configurations by pushing streams modules on top of streams device drivers or on top of other streams modules at runtime. This concept doesn't relate directly to our subject, but it has been a long living misunderstanding, that streams modules are loaded from the file system when they are pushed on a stream.

UNIX System V.4 provides dynamic binding. That means, a process can consist of a couple of modules which are linked on demand, at runtime. This is similar to our concept, but only implemented for user processes and not for the UNIX kernel itself.

There are also some implementations of dynamic driver loading, but as far as we know none meets all our design goals stated below.

- *Convergent Technologies* has implemented the concept of loadable drivers. The drawback of their implementation is that the drivers needed to contain dedicated code to support dynamic loading.
- For *Sun-OS* there exists a very simple but also quite restricted tool for fast downloading of drivers, intended only for the use as driver development tool.
- In *Interactive System Corporation*'s 386/ix product there seems to be non-released support for loadable drivers, which is based on a set of special system calls.

2. Design Goals

Several design goals led to the actual implementation.

The system had to be suitable for driver development, but also fit for a production system. Another planned application area were driver design workshops. Several requirements follow from these demands:

- Loading a driver had to be transparent for a process using it. It couldn't be expected, from an arbitrary program, to explicitly load the drivers it needs.
- It had to be fast. The time for loading a driver should be in the magnitude of the time needed to load and start a medium sized program.
- It had to be secure. Handling errors, even deliberate ones, shouldn't lead to kernel crashes.
- The system shouldn't use more memory in kernel space then it saves. Therefore memory consuming parts had to run in user processes, and only a necessary minimum in the kernel.
- The system should be able to handle all kinds of drivers, i.e. character, block, streams drivers, streams modules, optionally also file systems and other modules.
- The change/compile/download/test cycle of a driver should be repeatable without rebooting the system, even with faulty drivers, as long as they don't panic.

- Dynamically loaded drivers should be correctly handled by kernel debuggers.
- Especially for use in driver design workshops the system had to be multi-user, i.e. the process of loading/unloading a driver should be able to run in multi-user mode and started from any terminal.
- The concept should require neither changes in the kernel nor special code in the dynamically loadable drivers. It was desirable to have any drivers dynamically loadable, also third party provided drivers, where only object code was available.
- Last but not least, the concept should be compatible with AT&T's Installable Driver tools.

3. Implementation

First some detail about the kernel/driver interface: all driver functions are called through several switch tables. These are:

bdevsw	Entry table for block device driver
cdevsw	Entry table for character device driver
io_init	Init routines of the drivers, called at boot time
io_start	Start routines of the drivers, called at after Init
io_halt	Halt routines of the drivers, called at shutdown
kenter	Routines called before entering the driver code
kexit	Routines called before leaving the driver code
fmodsw	Streams module switch table
fstypsw	Entry table for file systems
ivect	Interrupt handlers

Normally these tables (their C definitions) are generated by the UNIX configuration tools from a configuration description. In some older versions of UNIX the C definitions of these arrays *are* the configuration description. To generate a UNIX system, the file containing these tables has to be compiled, then kernel modules, driver modules and this table should linked to produce the executable UNIX kernel.

On the other hand, drivers can access any kernel functions or data addresses directly. They can't (or better, shouldn't) access code and data of other driver modules directly. Modules, for which these rules aren't violated, can be safely removed from the configuration without producing undefined symbols when linking the kernel. This being possible, it should also be possible to run a kernel without those modules.

A driver module, configured as dynamically loadable, isn't included in the kernel at boot time. To add it to the running kernel, three basic steps have to be accomplished:

- Bind the driver object file to the address where it will placed in the kernel address space. The external symbols referenced must be symbols from the kernel, so their address can be taken from the symbol table of the running UNIX.
- Copy the driver's code and data into the kernel address space.
- Enter the addresses of the driver's entry functions in the corresponding slots of the switch tables.

The implementation is based on a client/server model. Clients are either the explicit load/unload commands or processes trying to open a driver which isn't currently loaded.

The server is the daemon process which does the real work of binding and loading the drivers. Clients and server communicate through a pseudo device driver, which also contains the necessary kernel support for the actual downloading of drivers.

Let's take a closer look at the two types of clients and at the server:

3.1. Explicit Load and Unload Clients

The commands *drvload* and *drvunload* are very simple. They scan the master configuration file, `/etc/conf/cf.d/mdevice`, for the driver name and build a request containing the name of the driver, its type, its major number if any, and the type of the request (load or unload). The request is then sent to the server by "writing" it to our pseudo device driver. The process then waits until the request is processed, either successfully or not.

3.2. Implicit Load

The first access to a driver is always a call of its *open* routine. For all major numbers for which no driver or loadable driver is configured, DDL enters the address of a common dummy open routine in the switch table. whenever this routine gets called, a load request can be built and sent to the server from within this function. After receiving a successful acknowledgement for loading the driver, the common open routine calls, through the *cdevsw/bdevsw* table, the open routine of the just loaded driver and then returns to the user process. Since the real open routine of the driver is entered in the switch table when the driver is loaded, the next open calls to the driver will go directly to its own open routine.

3.3. Implicit Unload

A driver can be configured with an "unload on close" flag. In this case the close routine of the driver isn't entered in the *bdevsw/cdevsw* table. A common close routine, also located in the DDL pseudo driver, is entered in instead. So each time the close routine of the driver is called, our common close will get control, call the real close routine, then check if all minor devices of the device driver are closed, and if so, initiate an unload request.

3.4. The Server

The server gets load and unload requests by reading from the DDL driver. The steps the server has to accomplish for a load request are:

- Locate module by its name or major number in the configuration files `/etc/conf/cf.d/mdevice` and `/etc/conf/cf.d/sdevice` and extract all needed information like type of device, functions of driver, interrupt channel.
- Read module object file (COFF file) and determine the size of its code, data, and bss sections respectively.
- Allocate memory in the kernel space by an ioctl call of our DDL driver.
- Bind the driver to the returned addresses.
- Fill a driver description structure with all the information needed to patch the kernel, i.e. sizes of code/data/bss, addresses of entry functions, interrupt channel and priority.
- Copy the drivers code and data in kernel space.
- Patch the kernel tables with the information provided in the driver description structure.
- If a kernel debugger is configured, update the in memory global symbol table used by the kernel debugger.
- Call the init and start functions of the driver.
- Acknowledge the client of completion of the request.

The sequence for unloading is shorter:

- If the driver is still open, refuse to unload it.
- Check if there are drivers depending on this driver. If so refuse to unload.
- Enter the address of the common open and close routines in *bdevsw/cdevsw* and delete all references to the driver in the various other tables.
- If the kernel debugger is configured, update the global symbol table accordingly.
- Free the memory allocated for the driver.

4. Limitations and Problems

Three main categories of problems occur with DDL:

4.1. Implicit Load/Unload Problems

Driver modules which are accessed before a call to their open routine, cannot be implicitly loaded. Such drivers are the streams modules and the cloning streams drivers. Here the data structures of the driver are setup before the open routine is called. Similarly after the close routine is called, the kernel still needs the drivers data structure to dismantle the stream.

4.2. Link Problems

In one case drivers cannot be dynamically loaded at all. This occurs when drivers are accessed in a non-standard fashion, i.e. symbols (data or functions) are accessed by kernel code or by other driver modules directly, without going through the switch tables. In this case, simply removing such a driver from the configuration would produce unreferenced symbols when linking the kernel. The solution in System V.3 for this problem is to provide "stub" files containing dummy functions and/or data declarations for those symbols accessed from outside. When the driver is removed from the configuration, such a stub file is compiled and linked to the kernel instead.

Loading such a driver at runtime would make it necessary to replace references to the stub functions with the addresses of the real functions. But these references are spread at arbitrary locations in the kernel code, and are not nicely kept together in well known switch tables. One way to cope with this problem would be to extract the information from the relocation table of the UNIX file, but this is discarded when linking UNIX. One might think of other solutions, but it gets tedious and probably slow. Anyway, in our opinion the best solution is to avoid this kind of construction. There is always a more elegant and more portable way.

A slightly different situation occurs when two or more drivers are dependent in a hierarchical manner. In this case, loading the lowest driver first and then adding its external symbols to the symbol table, makes it possible to load the next, and so on.

For example, an SCSI driver is usually divided in a low level driver (driving the SCSI controller) and one high level driver for each type of SCSI device, like tape, disk, scanner, and so on. The low level driver is accessed by the high level drivers through some arbitrary functional interface. When dynamically loading such a set of drivers the low level driver must be loaded first, then the high level drivers. When unloading the drivers, first all high level drivers must be unloaded, before the low level driver can be unloaded. The DDL system refuses to unload drivers as long as there are dependent drivers still loaded.

Drivers with cyclic dependencies cannot be handled by DDL. An example of such drivers are the master and the slave sides of a virtual terminal driver. These kinds of drivers could be dynamically loaded when DDL provided the ability to load drivers with multiple sets of interface functions, i.e. multiple major numbers. This is one the TODO's of this project.

4.3. Runtime Problems

The runtime problems are more complex and harder to fix. The common pattern of all these problems is that driver code often isn't "restartable". The requirement of the DDL system on a driver is that after the last close, a driver has to be stateless. The reason for this is, that after unloading and reloading a driver all the drivers data space is initialised to its original value (0 for bss, the compiled values for data) and so all the state information from its "previous lifes" are lost. We illustrate this by some typical examples. For some of these the DDL System provides solutions, but some are not solvable in general. They must be solved by redesign of the drivers themselves.

- Hardware initialisation: each time a driver is loaded, its init routine is called. This usually initialises the hardware. After the last close, a device should be in such a state that no harm is done by initialising the hardware again. Also the hardware has to be reset-able by software, and the driver *init* routine has to do it correctly and not rely on the assumption that the system has just been rebooted.
- Unfinished initialisation: drivers might initiate some hardware initialisation in their init or start routine and not wait for termination. When the hardware device finished its initialisation task, it sends an interrupt. The drivers interrupt routine then would finish the initialisation of the driver. This works fine, as long as between the call of the drivers init routine and the first call of open, enough time passes. But this isn't the case when a driver is dynamically loaded.
- Memory allocation: drivers usually allocate memory in their init routine. Then they usually don't free the memory after the last close. With DDL this would result in new memory being allocated after each loading of the driver.
- Late interrupts: devices producing interrupts after the driver is removed isn't really a problem, because the entry in the interrupt switch table referring to the drivers interrupt routine, is cleared and replaced by the reference to a dummy routine when a driver gets unloaded.

- Timeouts started by a driver must be correctly cleaned using *untimeout* after the last close. Otherwise, when a pending timeout hits after the driver is unloaded, it invariably leads to a crash. More generally, all of the kernel services with have references to the driver must be notified that the driver may be unloaded so that they won't attempt to access it anymore.

5. Conclusions

The DDL System has been used internally for several month. It has proven to be an invaluable tool for driver development. Imagine the following scenario on a X-Window work station:

In one window runs the editor, with the driver's source code. In an other window one could have *sdb* executing a test program in single step mode. After fixing a problem, or inserting some debug print statement in the source, the developer compiles, unloads and reloads the driver by running *make* from a third window. Then our programmer goes on, single stepping through his test program, testing the new version of the driver. And all that without getting the chance to light a cigarette!

Some examples of drivers the system has been tested with are: Several SCSI Drivers for scanner, optical disk, tape, a QIC streamer tape driver, floppy driver, serial driver, X21 driver, Lachman TCP/IP, TI streams modules and the kernel debugger.

For production systems the usefulness is limited to drivers very well known or very well tested together with DDL. The memory savings can be considerable for some systems, but sooner or later this will become obsolete, due to 4, 16, 64 MB chips.

Bibliography

AT&T, "Block and Character Interface (BCI)," in *Driver Reference Manual.*

AT&T, "Portable Driver Interface (PDI)," in *Driver Design Reference Manual.*

AT&T, "STREAMS Programmer's Guide."

Bach, Maurice J., "Design of the UNIX Operating System," Prentice Hall, 1986.

The XView User Interface Toolkit Object Model

Nayeem Islam

Sun Microsystems
2550 Garcia Ave.
Mountain View CA 94043
islam@sun.com

ABSTRACT

This paper describes the object-oriented programming model in XView, a user interface toolkit written in C and based on the X11 window system. The mechanisms by which XView supports two important aspects of object-oriented programming, data abstraction and inheritance, are discussed. This paper describes how the object-oriented programming model in XView maps onto the window systems environment. This approach is compared with the another object-oriented toolkit implemented in C, the *Xt Intrinsics* from the MIT X consortium.

1. Introduction

The C language itself does not support object-oriented programming, yet many environments built using C are object-oriented. A graphical user interface built on a window system is one such environment. The OPEN LOOK Graphical User Interface (GUI) [Sun88a] implemented on a windowing system, has many attributes that fit very well into an object-oriented programming model. For example, icons and windows on the screen in OPEN LOOK are objects that possess a state that changes as the end user, a person who sits in front of a workstation and uses the window system and applications, interacts with them. These objects have a set of operations that end users can then perform on them. For example, an end user of the window system may point to an icon by moving the mouse pointer on the icon. The end user may then grab the icon and drag it to the new location using the mouse. The icon now has a new location on the screen relative to the other objects, and its internal state will change to reflect this.

C was chosen because it is the most widely used language on UNIX systems. However, C lacks two essential features that are important for object-oriented programming: data abstraction and inheritance [Tes86a]. Data abstraction means that an object presents a client interface to object users, those who use the objects to write application programs, through a set of methods or routines. How these methods are implemented and the object internal data structures they manipulate is completely transparent to the user of the object interface. This separation lets object developers, those who implement the code for the object, to change the implementation of an object without requiring object users to change their code. Inheritance allows programmers to reuse preexisting pieces of software, to build new software by specializing or extending the existing software.

This paper explains the XView object model and how it supports the notions of data abstraction and inheritance. Since C does not easily support the concept of object-oriented programming several approaches to the problem have been proposed. Some of the modularity aspects of C have been explored, but this approach is based on static variables placed in one file [Dut83a]. Some approaches require a preprocessor or define an entirely different language, such as objective-C and C++. The approach presented in this paper simulates an object-oriented environment without language extensions. The approach is based on a set of guidelines that allows one to program in an object-oriented fashion. It is up to the object developer to expose the right interface and the object user to adhere to the style conventions outlined in this paper. The conventions for the object developer dictate how to partition data structures and methods into various files. Other attempts at object-oriented programming with unmodified C include the Xt Intrinsics [McCa].

2. The environment

OPEN LOOK is a specification for a direct manipulation graphical interface into the UNIX system. For example, there is an application called filemanager that provides a visual interface into the UNIX file system giving files iconic representations. There are different iconic representations for executables, directories and data files. Using the mouse, it is possible to click on a file and query its attributes such as ownership and access rights. It is also possible to double click on an iconic representation of the file and view it using an application or if the object is executable then the program corresponding to the icon is run. It is possible to drag a file (e.g. containing PostScript) from the filemanager into another application (PostScript previewer) that understands the file and does something with it (interprets PostScript and displays an image). This metaphor of grabbing, dragging, and dropping icons and windows is prevalent in OPEN LOOK. It strongly reinforces the notion that the end user is interacting with objects.

3. The XView User Interface Toolkit

XView is an X11 [McCa] based user interface toolkit that is object-oriented, event-driven, and implements the OPEN LOOK GUI. XView makes a distinction between the object and the description of the object. Many similar objects can be described by the same general description. This general description is called a *class* or *package*. Each object described by a class is called an instance [Rob81a]. Every object is an instance of some class. The class describes all the properties common to all instances of that class. The term object is used to refer to a collection of instances. Each instance contains the information distinguishing it form other instances of the same class. With XView, application programmers instantiate the standard classes that are part of the XView library, and use these instances to create the user interface part of an application. In addition, there is a mechanism to support the development of new classes through inheritance. This mechanism enriches the user interface.

XView is an event-driven toolkit. Callback procedures for an object are registered with a central **notifier**. When events arrive from the window server, the **notifier** determines which object gets the event. The **notifier** invokes the appropriate callback procedures. The **notifier** also allows objects to register callbacks for input on file descriptors, pseudo ttys and for UNIX signals [Jaca].

XView implements the OPEN LOOK GUI which is a functional specification that provides a standard look and feel for applications. This specification describes a large set of user interface concepts which can be thought of as objects. Examples of these concepts include icons, windows, control areas and drawing surfaces. The XView toolkit has classes that correspond to these OPEN LOOK concepts.

4. XView object model

From an object-oriented system's standpoint there are two structures of interest in a window system. The first is the (tree) hierarchy of the classes. This tree is a list of all the XView objects that can be created by an application developer. This tree is static and represents how the various classes share data and functions. XView classes include scrollbars, text editors, terminal emulators, control areas, bitmap images, fonts, icons, windows, and menus. The second structure is the instance tree that is created when the classes are instantiated and linked. The instance tree is what the end user of the window system actually sees on the screen. This tree is a dynamic structure and is manipulated by the XView programs during execution. An application, typically consists of a tree of instances rooted at the main application window. This window in turn is a child of the root or background window.

There are two types of classes in XView: *primitive* and *composite* classes. Instantiated primitive classes can only be the leaves of an instance tree. An instance of a composite class can be either a leaf or an interior node in an instance tree. As an interior node, a composite instance may have several child instances. Composite instances are responsible for the geometry management (layout) of its children and in some cases any drawing to the children is clipped to the boundaries of the parent. These definitions are similar to those used in the Xt intrinsics. For example, a menu instance could be the parent of several menu button instances. The parent is responsible for determining the sizes and positions of the menu buttons. In addition, any drawing in the buttons would be clipped to the boundaries of the parent. The menu button instances are *primitive* classes, whereas the menu is an instance of a *composite* class. This composite instance can be moved to different parts of the instance hierarchy.

Each instance contains a part that is unique to that instance and a part that is shared with all instances of the class. XView lets programmers create a subclass statically. At compile time, the class hierarchy and composition are defined and cannot change. This means that no new class can be created and no new properties can be added to a class at runtime. Object users get handles to instances of a class. The instance possesses certain attributes that defines the behavior it will exhibit to the user of the instance. Once a programmer creates an instance, he can query the instance to determine the state of its attributes or he can change the state of an instance by setting the value of attributes. The object user does not know how the object implements or effects the requested change. The exposed types of the attributes have nothing to do with the actual implementation. With XView the class developers choose from a large set of types for the attributes including integer, character, pointers to a character, and function pointers. It is also possible to create a list of attribute-value pairs and use it in multiple XView calls.

5. Data Abstraction

Data abstraction refers to the process of hiding a data structure behind a set of predefined methods or access routines. These methods are the only routines that can access the data. The object developer presents the object user with an interface of methods and attributes. Object users change the state of an object through these methods with the appropriate attributes. The object developer has *abstracted* the concepts of the object in the interface of the object. The object user has no idea of how the attributes are implemented, or what internal procedures and data structures the object developer has used. XView achieves data abstraction at the instance level by providing an *opaque* handle to the object user. The opaque handle is a handle to an object structure but the user of the handle does not know the definition of the data that it represents. Thus, by convention the object user should not access the internal data of the object. (Unfortunately, in C it is possible to access anything one wants to through pointer arithmetic.) This abstraction is reinforced by exposing only one public *.h* file to object users and object developer subclassing from existing classes.

All objects export five public-methods:

1	*xv_create*	create an instance of a class with the specified attributes,
2	*xv_set*	set the values of attributes of an instance,
3	*xv_get*	query the values of attributes of an instance,
4	*xv_destroy*	destroy an instance, and
5	*xv_find*	find an instance with the specified parent and attributes.

These five actions will be referred to as the public-method interface as they form the only interface to all objects in XView. Object creation or object search gives the object user an opaque handle to use in calls to *xv_set*, *xv_get* and *xv_destroy*. Class developers present an interface through attributes and this is what differentiates classes.

6. Inheritance

In many object-oriented programming systems, code can be reused by a mechanism know as *inheritance*. Through inheritance it is possible to define a new class as a variation of an existing class. The new class may specialize the old one or it may extend the features in the old one. The new class is called the *subclass* of the old and the old class is called the *superclass* of the the new class. Each class adds new features by creating new attributes, or it can *override* attributes of the superclass by modifying the attribute-value lists before they are processed by the superclass. Only single inheritance has been implemented, which means that a class has only one superclass, whereas with multiple inheritance a class may have more than one superclass. An exception to this is the Term class, which provides terminal emulation with sophisticated text editing facilities. This class is a Tty (terminal emulation class) except that it uses some of the procedures of the Text (text editing class). This is not a general mechanism for multiple inheritance and was a makeshift effort for obtaining the necessary capabilities. Figure 1 shows the hierarchy of classes that are part of the standard XView toolkit. The hierarchy shows the composite classes frame, menu, text, tty, openwin, canvas and panel. All the other classes are primitive classes. In the XView class tree there are some *hidden* internal classes that are not exposed to object users — these are convenience classes that can only be used by other class developers are not meant to be instantiated. The **drawable** and **openwin** classes, show in parentheses in figure 1, are examples of these hidden internal classes.

7. The structure of a class and defining a new class

To define a new class requires at least two *.h* files and two *.c* files. The requisite information for a new class is in the header files of the superclass and the XView system header files. Excerpts form the system header files are shown in figure 2. The first two typedefs, *Xv_opaque* and *Xv_object*, are used to refer to the opaque handles used to reference objects. The third typedef, *Pkg_private*, is used for variables that are shared by files in a class. The structure *_xview_pkg* is the **Class** part structure. All classes have the same

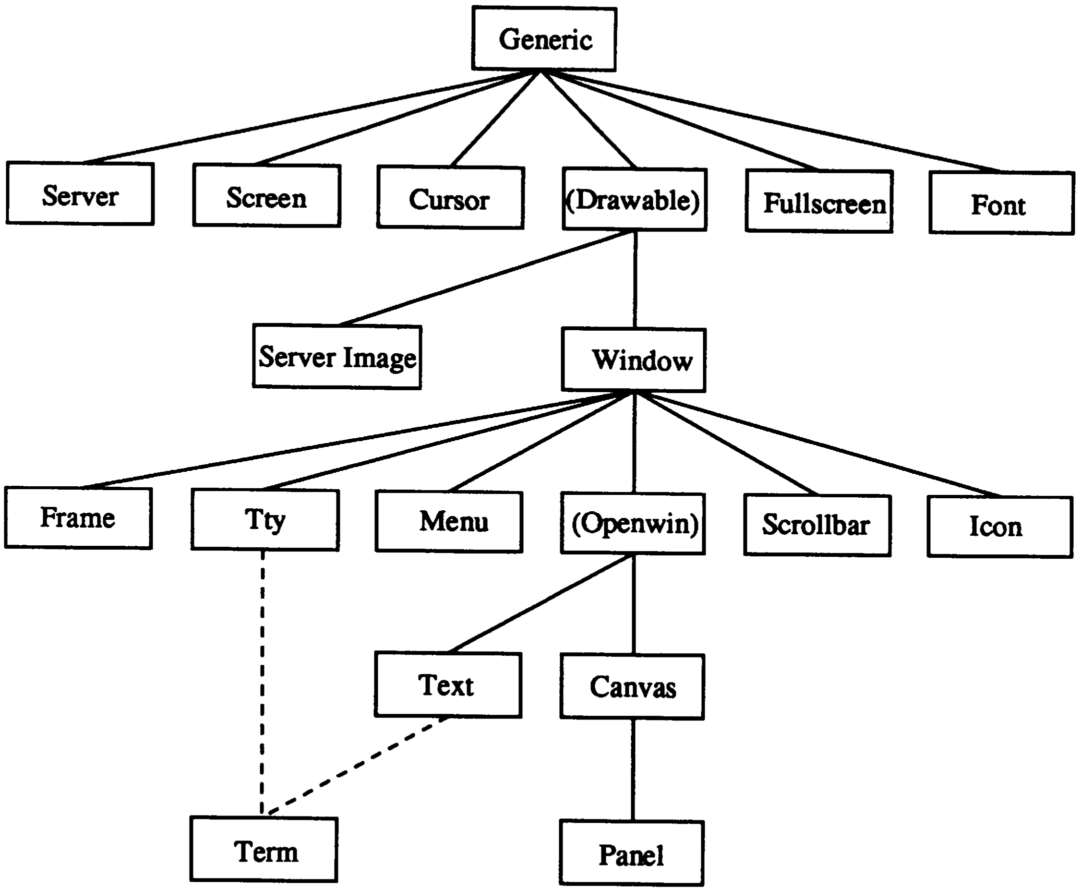


Figure 1: *XView Class Hierarchy*

structure for the class part but the contents of the structure vary from class to class. On the other hand, all instances of the same class share the exact same structure including its contents. The class structure has nine fields which are explained in figure 2.

```
typedef char      *Xv_opaque;
typedef Xv_opaque Xv_object;
typedef extern    Pkg_private;

typedef struct    _xview_pkg {

    char      *name;                  /* name of the class */
    Attr_pkg  pkg_id;                 /* class identifier */
    unsigned  size_of_object;         /* size of the public structure */
    struct    _xview_pkg *parent_pkg; /* pointer to parent class */
    int       (*init)();              /* initialize method for class */
    Xv_opaque (*set)();               /* set attributes method for class */
    Xv_opaque (*get)();               /* get attributes method for the class */
    int       (*destroy)();           /* destroy method for the class */
    Xv_object (*find)();              /* find method for the class */

} Xv_pkg;
```

Figure 2: *Excerpts from XView system header file*

For example, to create a class called **Window** (this is not the standard window class in XView) requires a *WindowClass.c*, *Window.c*, *WindowPrivate.h*, and *WindowPublic.h* file. This class structure is placed in a *.c* file and is determined before compiling the source for the class. The *WindowClass.c* file contains the class declaration as shown in figure 3. This structure is shared by all instances of the **Window** class and all instances of any subclass of **Window**.

```
WindowClass.c                  /* class declaration - shared by all
                                * instances of the class and all
                                * instances of any subclass
                                */
#include "WindowPrivate.h"     /* Class definition structure,
                                * shared by all instances of the class
                                */
Xv_pkg  xv_window_pkg = {

        "Window Class",          /* class name */
        ATTR_PKG_WINDOW,         /* Window class  identifier - unique */
        sizeof(Xv_window_struct), /* size of public structure*/
        &xv_drawable_pkg,        /* parent class is the class drawable*/
        window_init,             /* create / initialize method */
        window_set_avlist,       /* set method */
        window_get_attr,         /* get method */
        window_destroy,          /* destroy method */
        NULL,                    /* no find method - set to NULL */

};
```

Figure 3: *Class definition for Window*

The five public-interface methods defined are the only public methods of any object.

Those methods call the appropriate set of private internal functions in each class' class structure. More than one internal function can be called when a public method is invoked because the private methods are chained in a list with the corresponding superclass and subclass private functions. This chaining is automatic for all the procedures in the class definition. This is referred to as *superclass chaining* and there are two variations to it: upward and downward superclass chaining. In upward superclass chaining all the private chained methods are called in subclass to superclass order all the way to the base class (for example, in the order panel-canvas-openwin-window-drawable-generic). In downward superclass chaining the private methods are called in superclass to subclass order (for example, generic-drawable-window-openwin-canvas-panel).

A class can inherit all the private functions of its superclass without adding any of its own by inserting a NULL in the appropriate place in the class declaration structure. In figure 3, the class **Window** will inherit the superclass find methods of its ancestors. A class automatically inherits behavior from its superclass by not processing the attribute, whose behavior the class wants to inherit. Additionally, a class can override the behavior of an attribute by acting on it and then deleting it from the attribute list before its superclass examines the attribute list. Since *xv_set()*s and *xv_get()*s are downward chained a subclass processes an attribute value list before its superclass processes it.

The five interface methods map to private methods in the following ways.

1. Create an object of type *Bclass*, with the specified **parent** and with its attributes set to the values indicated in the attribute-value list. The method invocation is:

   ```
   Xv_object = xv_create(parent, class, attribute-value pairs)
   ```

   The parent is used to insert the instance in the correct place in the instance tree. All the initialize procedures are called in downward superclass chaining order and the create time attributes are parsed at that time. Create time attributes are those attributes that only have meaning at create time. Then all the set routines are called in upward superclass chaining order to parse the remaining attributes and finally, all the set routines are called with the only one attribute, *XV_END_CREATE*, in downward superclass order to allow for post creation processing. This method returns a handle to the instance of the class.

2. Destroy the object. The method invocation to destroy an object is:

   ```
   int xv_destroy(object)
   ```

   the return value indicates whether or not the object was actually destroyed. The destroy procedures of the classes are called in upward superclass chaining order.

3. Get the values of the attributes and put data into the indicates structures or return a pointer to the structure. XView currently has both models. The method invocation is:

   ```
   Xv_opaque xv_get(object,attribute-value list)
   ```

   The get procedures of the objects are called in upward superclass chaining order.

4. Set the values of the attributes of the object to the values specified in the attribute-value list. The method invocation is:

   ```
   Xv_opaque xv_set(object,attribute-value pairs)
   ```

   The set procedures of the objects are called in upward superclass chaining order.

5. Find an object of type class, with parent and matching the attribute value list specified. The method invocation is:

   ```
   Xv_object xv_find(parent,class,attribute-value pairs)
   ```

   The find procedures of the objects are called in upward superclass chaining order. If the find fails then this reverts to a create.

The variable-length attribute-value list interface to these functions is particularly useful. Incomplete specifications allow for a natural style of programming. Those attributes that are of interest are specified in the method invocation. The create function has parent object specified. If this is NULL the parent is set to the root or background window.

Figure 4 shows the *WindowPublic.h* file which is the only file a user of the class or developer of a subclass of the **Window** class needs. This approach,exposes the same *.h* file to both new object developers and object users.

With the ATTR macro, shown in figure 4, it is possible for two packages to have attributes defined to the same value. If one class is an ancestor of another then the class identifier of the class structure will differentiate between the two attributes. [Sun89a] The structure *Xv_window_struct* is the structure used in linking all instance pointers together. Since it is used by those who will subclass the **Window** class, it is referred to as a public structure. This is a public structure since subclasses may access this structure. It has a pointer to the private to instance part. Figure 5 shows *WindowPrivate.h* that contains all the private data declarations and the private function prototypes. This file contains the instance part of the object, which is called *Window_private_struct*. Each instance can have different values in its instance part structure.

Figure 6 shows sample implementations of five private functions that are called when the public-interface methods are invoked. An object user never calls these functions directly. For simplicity these functions have all been put in to one file, *Window.c*. Each is often placed in a file of its own. They are private functions and are not accessed by object users or other classes subclassed from **Window**. When a subclass wants the superclass to process something, it will call the superclass through the five interface-methods. This approach hides class implementation from the object developer as well as the object user.

If the class developer leaves out a function in the class definition structure, by inserting a NULL in the appropriate place in the class structure, then the superclass chained functions are used instead. For example, in the class definition structure for the **Window** class the field for the FIND procedure was NULL. The chained FIND method list of the superclasses will be called.

The public-header file contains all the defines necessary for creating an application or creating a new class. figure 7 shows the use of the XView public method interface for an object user or application programmer.

This application creates a window with a call to *xv_create()*, of height and width 20 pixels by 20 pixels, with the root window as the parent. The call to *xv_set()* adds an event handler to the object. The call to *xv_get()* queries the window instance about its size. The call to *xv_destroy()* then destroys the window object.

8. XView objects in a window system

The objects in the screen are related to each other through a tree structure. This is a dynamic tree of XView created and changed by an XView application programs during execution. It is these objects that the end user interacts with and identifies as the user interface of the system. The end user typically initiates the change in the tree by performing the appropriate action on the XView application. For example, when the end user presses the menu button on a text surface the XView toolkit will create the relevant menu, attach it to the instance tree and display it. figure 8, shows the instance tree for the an application called textedit. The figure shows the parent-child relationships between the instances. The figure shows instance of the the composite classess **Frame**, **Panel** and **Text**.

```
WindowPublic.h                        /* public header file for object user
                                       * and developers of new classes
                                       */

#include "GenericPublic.h"            /* base class, root of class tree */
#include "DrawablePublic.h"           /* hidden class */

extern Xv_pkg     xv_window_pkg

                                      /* WINDOW is used to identify class in
                                       * call to xv_create()
                                       */
#define WINDOW    &xv_window_pkg

typedef Xv_opaque Xv_Window;

                                      /* choose unique id identifying this
                                       * class - requires knowledge of
                                       * superclass ids
                                       */
#define ATTR_PKG_WINDOW (ATTR_PKG_UNUSED-1)

                                      /* Attribute definition macros:
                                       * assigns identifiers for enums
                                       */
#define WINDOW_ATTR(type,ordinal) ATTR(ATTR_PKG_WINDOW,type,ordinal)

                                      /* attributes defining programming
                                       * interface of the object
                                       */
typedef enum {

    WINDOW_HEIGHT      = WINDOW_ATTR(ATTR_INT, 1),
    WINDOW_WIDTH       = WINDOW_ATTR(ATTR_INT, 2),
    WINDOW_EVENT_PROC  = WINDOW_ATTR(ATTR_FUNCTION_POINTER, 3),

} Window_attribute;

                                      /* required only by someone subclassing
                                       * the Window Class: this is where the
                                       * subclassing is occuring.
                                       * This structure concatenates all the
                                       * public structes structures all the
                                       * way to the generic class. Each public
                                       * structure has a pointer to a private
                                       * part
                                       */
typedef struct {

    Xv_drawable_struct parent_data;   /* concatenate parent structure */
    Xv_opaque          private_data;  /* pointer to private data for an
                                       * instance of window class
                                       */
} Xv_window_struct;
```

Figure 4: *public header file for object developers and object users*

An important aspect of a distributed window system is its client/server model. In this model, clients send requests to the server and receive events from the server. XView has a notification mechanism that handles system events from outside the process. Objects and applications can register procedures that will be called when events are received. For example, to handle events from the server XView has a special type of callback known as event handlers. The window object has event handlers and all subclasses inherit them. These event handlers, are registered by the appropriate objects at initialization with the **notifier**. When input is available from the server the appropriate handlers are invoked. Event handlers, like the users of objects, affect the state of objects. In fact, just like users of the objects, they use the five interface methods to change the state of an object. These event handlers are managed on a per instance basis. In XView, the standard window class has a default event handler installed at initialization. This event handler is stored in the instance part. In fact, the window subclasses canvas and panel both add event handlers. These handlers are chained, each new subclass can interpose its own event handlers ahead of the event handlers registered by all superclasses. If an object does interpose its own event handler ahead of other event handlers, then it has to decide whether or not to call the other event handlers for every event it receives. Chaining fits well into the object-oriented scheme and lets instances partition the handling of

```
WindowPrivate.h                 /* private to Window class*/

                                /* superclass public header files: used by
                                 * class developers as well as class users.
                                 */
#include GenericPublic.h
#include DrawablePublic.h
#include WindowPublic.h

                                /* macros to transfer between public and
                                 * private parts of the structure
                                 */
#define WINDOW_PUBLIC(item)  XV_PUBLIC(item)
#define WINDOW_PRIVATE(item) XV_PRIVATE(Window_private_struct,Xv_window_struct,item)

                                /* instance part structure. Each instance of a
                                 * window object has a copy of this.
                                 * This is also referred to as the instance
                                 * specific part of an instance
                                 */
typedef struct {

        Xv_window public_self;  /* back pointer to public struct */
        int       height,width;
        void      (*event_proc) ();

} Window_private_struct;

                                /* class private methods */

Pkg_private int window_init();
Pkg_private int window_destroy();
Pkg_private Xv_opaque window_set_avlist();
Pkg_private Xv_opaque window_get_attr();
```

Figure 5: *Private header file for the Window class*

```
Window.c /* Private class methods for Window class */

/* Intialize method. Allocate instance part data structures */

Pkg_private int

window_init(parent, window_public, avlist)
        Xv_opaque      parent;
        Window_public *window_public;
        Attr_avlist    avlist;
{
        Window_private_struct *window_private = (Window_private_struct *) xv_alloc(Window_private_struct);
        Xv_window_struct *win_public = (Xv_window_struct *) window_public;

        /* pointers between private and public */

        win_public->private_data    = (Xv_opaque) window_private;
        window_private->public_self = (Xv_opaque) win_public;

        /* other data structure initializations */

        window_private->parent = parent;

        /* parse create only attributes */
        /* create an X Window */

        return XV_OK;
}
```

Figure 6a: *Private methods of the Window Class*

events received from the server. It is also possible to register other types of handlers for signals and input from other file descriptors such as from pseudo ttys.

```
/* this is the destroy routine called when an object is destroyed. There
 * are two types of destroy in XView - in the first type the status flag
 * is set to DESTROY_SAVE_YOURSELF and the client does not have the option
 * of stopping this. In the second type the object has a chance to veto
 * the destruction and destruction is then a two phase process. Phase one
 * is - DESTROY_CHECKING and phase two is DESTROY_PROCESS_DEATH or
 * DESTROY_SAVE_YOURSELF.
 */

Pkg_private int

window_destroy(window_public, status)
        Window_public  window_public;
        Destroy_status status;
{
        Window_private_struct *window_private = WINDOW_PRIVATE(window_public);
        Destroy_status status;
        if (status == DESTROY_CHECKING) {

                /* gives objects a chance to veto its destruction */

        }
        else if (status == DESTROY_PROCESS_DEATH) {

                /* this process is going away do minimal cleanup */

        }
        else if (status == DESTROY_CLEANUP) {

                /* clean up all object data strutures and free up server
                 * resources used by this object
                 */

        }

        else if (status == DESTROY_SAVE_YOURSELF) {

                /* if we have data to be saved for next time system
                 * is started do so now
                 */

        }

        return(XV_OK);

}
```

Figure 6b: *Private methods of the Window Class*

9. Comparison of Xt intrinsics and XView

The Xt intrinsics shares the base model of objects and attributes with XView. These objects and attributes, which are call resources can be queried and set. Unlike XView, the Xt intrinsics does not have a uniform interface to all objects. There is a proliferation of different convenience routines for creating and manipulating objects. This tends to complicate the interface. There is a stronger separation between the object developer and user of objects in Xt. In Xt the object developer uses a different set of header files from the object user. The Xt intrinsics does not have chained event handlers. The object classing is through structure concatenation rather through pointer chaining as in XView. This difference means that every time a new class is defined it must include all the class structures of all superclasses. XView requires only a pointer to one superclass structure which itself has a pointer to its superclass all the way to the base class. This means that an instance of a class A and an instance of its subclass B will not share the Class parts of A, whereas in XView they would.

One disadvantage of XView is that every class requires a unique identifier to be supplied. It is used in distinguishing attributes between a class and its ancestor class. It requires knowledge of all its superclass' identifiers. All the functions in the Class declaration structure are chained in XView. In the Xt intrinsics some are chained and some are not. Those that are not must be explicitly inherited. The Xt approach is more flexible and allows subclasses to get rid of superclass methods statically.

```
/* routine to query the value of an attribute of the object. Note that the
 * superclass attribute DRAWABLE_ID is overridden by this class by
 * consuming the attribute before the superclass drawable get a chance to
 * process it.
 */

Pkg_private Xv_opaque

window_get(window_public, status, attr, args)

        Window_public     window_public;
        int               *status;
        Window_attribute attr;
        Attr_avlist       args;

{
        Window_private_struct *window_private = WINDOW_PRIVATE(window_public);
        Xv_opaque v;

        switch(attr) {
        case WINDOW_WIDTH:
                v = (Xv_opaque) window_private->width;
                break;
        case WINDOW_HEIGHT:
                v = (Xv_opaque) window_private->height;
                break;
        case WINDOW_EVENT_PROC:
                v = (Xv_opaque) window_private->event_proc;
                break;
        case DRAWABLE_ID:
                v = (Xv_opaque) NULL;
                ATTR_CONSUME(attrs[0]); /* override an attribute defined */

                break;                  /* in a superclass */
        default:
                xv_check_bad_attr(&window_pkg,attrs[0]);
                break;
        }
        return(Xv_opaque) v;
}

/* set the attributes to the values specified */

Pkg_private Xv_opaque

window_set_avlist(win_public, avlist)
        Xv_Window               window_public;
        Window_attribute        avlist[];
{
        Window_private_struct   *window_private = WINDOW_PRIVATE(window_public);
        Attr_avlist     *attrs;

        for(attrs = avlist; *attrs; attr_next(avlist)) {
            switch((int) attrs[0]) {
                case WINDOW_HEIGHT:
                    window_private->height = (int) attrs[1];
                    break;
                case WINDOW_WIDTH:
                    window_private->width = (int) attrs[1];
                    break;
                case WINDOW_EVENT_PROC:
                    window_private->event_proc = (void (*)()) attrs[1];
                    break;
                default:
                    if (xv_check_bad_attr(&window_pkg,attrs[0]) == XV_ERROR)
                        *status = XV_ERROR;
                    v = (Xv_opaque) 0;
                        break;
                }
        }
        return XV_OK;
}
```

Figure 6c: *Private methods of the Window Class*

```
#include "WindowPublic.h"


int height;

window = xv_create(NULL, WINDOW,
                        WINDOW_HEIGHT, 20,
                        WINDOW_WIDTH,  20,
                        NULL);

(void) xv_set(window, WINDOW_EVENT_PROC, event_handler, NULL);
height = xv_get(window, WINDOW_HEIGHT);
xv_destroy(window);
```

Figure 7: *Sample XView application*

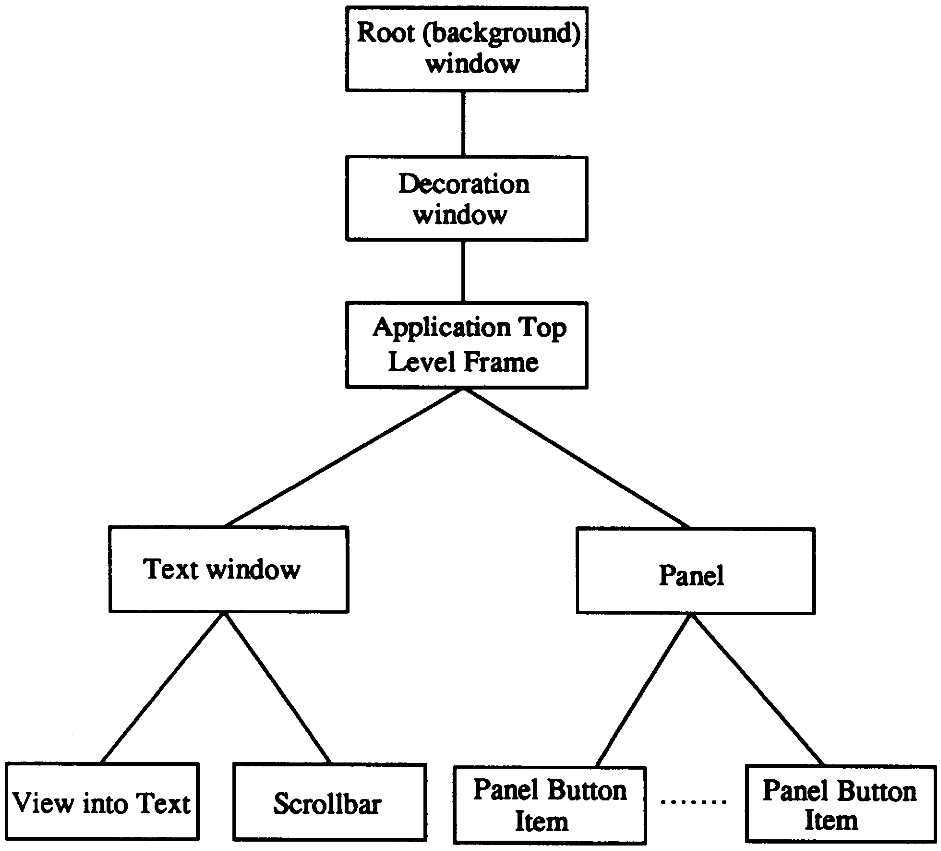


Figure 8: *The object instance tree of an XView texteditor*

10. Conclusions

An entire toolkit was built using the object model presented in this paper. There is large set of applications that use only the predefined classes. There are also tools that have subclassed from existing classes. The common object interface is a very powerful model since it forces a level of consistency at the object usage level, and the variable length attribute value interface is particularly useful for incomplete specifications of interfaces.

One of the weak points of XView is the separation of the event handlers and the *xv_set/xv_get* methods. Consolidating these two models would have been ideal. Both affect the internal state of the object from external stimuli: one emanates from the object user through the application programmer's interface and the other from the interacting processes in the systems, such as the server. In such a consolidated model the event handler would be a chained private class method and a chained instance method (with the instance event handler being called before the class handler) and there would be only one interface into the object – *xv_send*, which would send an event to an object. Objects are manipulated by sending messages to them. There could have been a stronger distinction between object developers and object users by providing two separate header files and conventions for each.

References

[Sun88a] AT&T and Sun Microsystems, *OPEN LOOK Graphical User Interface Functional Specification*, 1988.

[Dut83a] K Dutta, "Modular programming in C: an approach and an example," *SIGPLAN notices*, vol. 20, no. 3, March 1983.

[Sun89a] Sun Microsystems Inc, *XView programmers guide 1.0*, August 1989.

[Jaca] T W R Jacobs, "The XView Toolkit: An Architectural Overview," *3rd Annual X Technical Conference*.

[McCa] J McCormack, P Asente, and R R Swick, *X Toolkit Intrinsics, C Language X Interface*, Digital Equipment Corporation. For X11R2.

[Rob81a] D Robson, "Object Oriented Software Systems," *BYTE Magazine*, vol. 6, no. 8, August 1981.

[Tes86a] L Tesler, "Programming Experiences," *BYTE Magazine*, August 1986.

Interface between UNIX and HyperCard

Mikael Wedlin

Dept of Computer and Information Science
Linkoping University
mwe@ida.liu.se

ABSTRACT

This work describes a method of using the benefits of UNIX as a developing system and targeting the result to, in my opinion, a more user friendly system, HyperCard on a Macintosh. HyperCard can best be described as a programmable information handler.

UNIX is an operating system written by programmers, for programmers and few ordinary users will ever have any advantage of the complex design of UNIX. They often use some simpler personal computer instead, often with a window based interface instead. These systems are often difficult to program due to the lack of standardisation and a simpler operating system. Window systems are also rather difficult to program due to their complex interaction with the user.

My goal was accomplished by writing a program that could convert a UNIX program to a code resource that was linked to a HyperCard stack. This program then works as an external command (XCMD) in HyperCard.

1. What I have done

The origin of this program was a project I worked on, whose purpose was to create a development environment for a set of transputers [Inm88a]. The result so far from this project is described in the last example in this paper.

As a target for this system the Macintosh was chosen with HyperCard as the interface to the user. This was done mainly for two reasons. First, a Macintosh is relatively simple to interface to other systems via the serial ports and so cheap that it can be dedicated to the task. Second, the user interface on a Macintosh is easier to work with for the user than the traditional UNIX environment.

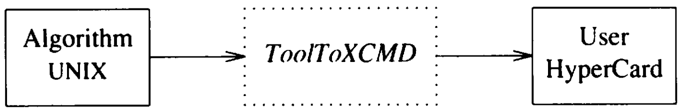


Figure 1: *The glue between UNIX and HyperCard*

Normally I work in a UNIX environment, and the first thing I discovered was that I didn't like programming the Mac one bit. The solution was to do as much work as possible on the UNIX system and then move the programs to the Macintosh environment.

2. Why bother?

I used to write all my programs in a UNIX environment and readily got rather fond of the nice feeling of having a lot of compilers, debuggers, and all the other tools available [Ker78a]. Suddenly one day this whole world collapsed when I had to write programs that were supposed to be used by people with very little or no experience with computers. They did not enjoy UNIX one bit, Macintosh was much better, slam the door, period. After a while I had to agree with them, UNIX is not designed for ordinary users, other systems are much better in that respect.

These users don't usually care about how advanced their operating system is, all they see is the user interface, and the user interface of UNIX is not very easy to grasp for an ordinary user. The trend today is towards a "click and drag" interface with windows, mouse and menus. Even the trend for UNIX today is to build a workstation with some sort of window system built on top of the UNIX kernel [Fri87a].

A number of these interfaces have evolved during the years and there has not yet emerged any standard. Every small computer has its own set of routines and os-calls. There is however a trend towards X-windows [You89a] as some sort of standard for windows today, at least for UNIX systems. There is still a long way to go, though. It is, however, still a painful experience to program windowing applications, at least on small computers like the Macintosh. They often need some sort of object oriented approach to programming that is crunched into a poor pascal or C compiler with virtually no debugging support. Most of the work is therefore spent on trying to set that window on the screen at the right position on the right time without the system crashing. Macintosh gave me nightmares about "La bomba"† after no more than a week.

2.1. UNIX vs. Macintosh

UNIX was primarily designed to ease the life of system developers and to give the programmer an easy-to-use view of the hardware. The small assumptions UNIX makes about the hardware and the uniform structure of device drivers makes it easy to port UNIX to nearly every system that exists on the market, from PC to Cray.

The situation for systems like the Macintosh is a bit different. A main goal for them is often to make as much use of the hardware as possible, and since that changes a lot, the interface to the hardware also changes a lot. An example of that can be the numerous soundchips that exists in home computers, all of them with a different set of capabilities. UNIX has only one standard way to give a sound, the bell character, and even that is not really part of the operating system. Personal computers often have a lack of performance that makes it worth the effort to have a special interface, especially with graphics that is rather slow on most computers today.

Another difference is that there are a lot more users without computer experience using the Macintoshes than there are users programming them. That leads to, compared with UNIX, poor development environments for them. It is more profitable to write end user applications.

2.2. A complex system makes life easier

The internal and external complexity of UNIX is used to simplify the life of the skilled programmer. However, it takes a while to see the beauty of short commands. Most people find it annoying to try to remember commands like *ls*, *cat* and *lpr*. On the other hand, UNIX is made up of a lot of small commands that can be put together using pipes or shell scripts to do powerful tasks.

One reason for not having UNIX on small computers is the way UNIX handles files. The tree-structure with mounted file systems is nice to have on large disks but doesn't feel that necessary on smaller systems where it is better to address the files with a device instead, e.g. a floppy station.

Standard UNIX io is handled with device drivers that are constructed to interact efficiently with devices like terminals and disks. Every other IO-unit must be mapped to look like one of these two kinds of special files. This makes it easy to build a UNIX for some hardware but it makes it difficult to put other sorts of devices in a UNIX system and have them work efficiently. It is true that there is a lot of different devices connected to UNIX systems, (i.e. plotters, image scanners, sound devices) but they must be crunched into a device which looks like a disk or a tty. A lot of overhead is introduced here and they can seldom be very efficient [Tan87a].

2.3. Instead of processes in the Mac world

The design of Macintosh started with making an effective user interface and that leads to a design that hardly can be described as very portable. However, the portability between the different members of the Macintosh family is remarkable, despite their different architectures.

† When it detects an error, Macintosh shows you a window with a number of bombs in and tell you that it needs a re-boot due to some error. The error is given as a number, very instructive!

The Mac doesn't contain any real OS, but rather a large collection of function calls that user applications can use for different tasks. They are divided into a collection of graphic procedures called QuickDraw, and a rather unusual file system.

Files have an interesting structure on the Macintosh: they are divided in two parts, one called the data fork and the other the resource fork. The data fork is the part of the file that mostly corresponds to ordinary files and the resource fork is the file part with some structure built in. This part contains, so called, resources and they are grouped into resource types. A resource type is a 32 bit word but generally treated as 4 ASCII characters. There are a lot of predefined resources, like fonts, cursors, icons, strings and so on. Every resource also has a unique ID number and an optional name. When resources is needed they are first loaded into memory and then the address of it is returned to the application [Ros85a].

To write applications for the Macintosh is normally a pain (as I see it), mostly due to the following reasons:

- A large part of the application has to deal with interaction with the user. The kernel defines a fair interface to a window system, but it is a lot of routines, and most of them are difficult to use. Programs that must deal with mouse and keyboard events are difficult to do right. It is also very difficult to test a program properly because the order of the events is very unpredictable. This, however, is true for window systems in general, and not just the Macintosh.
- Unlike UNIX, the Macintosh has a very specialized interface to the hardware and applications written for it can not be easily ported to another system. This implies, with the fact that it is a cheap system mostly designed for end users, that all compilers and development systems must be written more or less exclusively for the Macintosh. Lack of performance and lot of bugs is a common experience in many programmers lives.
- "Small is beautiful" has never been a star to follow for Apple. The latest system ROM is 512K bytes, with a collection of approximately 900 system calls. "Inside Macintosh" is a "bible" that no programmer can live without, and no-one (I believe) can remember the calling sequence of even a fraction of the system calls.
- Macintosh is a typical single user system with one process taking up all available resources. There are, however, some things that have to be done in the background, for example, the networking code. Apple solved this problem by introducing a linked list of tasks executed at every video scan interrupt (60 times a second). This list can be manipulated by the user and requires a special format of the function to be executed.
- When a UNIX process does something it shouldn't do, it usually gets some signal and dies, often with a core dump that can be examined for the source of the error. When the same thing happens within a Macintosh application, the whole system dies. No core dump, 1 minute of startup time, sometimes a corrupted file system. "La Bomba" strikes again.

2.4. Relocation in Macintosh

The Macintosh has a very complicated format of the executable code. It is relocated at run-time and divided in several smaller blocks with a table of computed code (a jump table) as a glue between them. This scheme can be run on cheap hardware at a cost of a difficult interface for the programmer. In most UNIX systems, at least the larger ones, relocation is done in hardware, and the programmer can treat the address space as a continuous memory.

2.5. Summary

In this section I have tried to point out the different design issues that led to my decision to develop the functions in a UNIX environment and then move it to a system with a more user-friendly interface, Macintosh and HyperCard.

Most of the problems I encountered have to do with the user interaction and the problems become larger as the possibilities of the user interaction increases. UNIX is rather clean in this respect because a process interacts with users in an ordered way through just two files, standard input and standard output. The only way in UNIX to bypass this is to send a signal to the process, and this is often the messy part of the implementation that the code writers would rather forget about.

3. How to do it

ToolToXCMD is the program (or tool as MPW calls them) that takes a program that looks like a common UNIX program (with *main*, *argc* and *argv*) and converts it to an "XCMD" or "XFCN" resource. The trick to do this is that the program is not linked with the usual library. The main entry point, often referred as *crtso*, is changed to a new entrypoint that converts the parameters from HyperCard to the usual *argc*, *argv* pair. It also sets up a new area for the global variables. The new runtime library also contains new functions for input and output.

The process that used to build a HyperCard application can be divided into the following 5 steps:

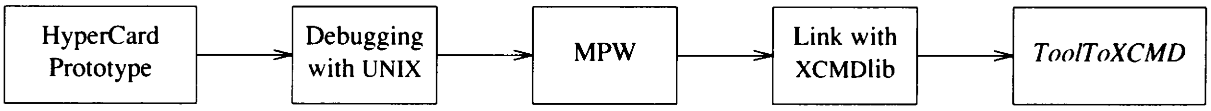


Figure 2: *The course of action*

1. Make a prototype in HyperCard
   Start with the external behavior by making a stack that looks like the final, but without doing anything. The strength of HyperCard shows itself here. It is a very interactive process where buttons and fields are spread out on a collection of cards. The background images can be used to give an extra dimension to the stack. It can either be painted with the painting tools or scanned with a scanner. Finally decide what functions to include as external commands.
2. Write and debug the external commands
   Use a UNIX system to write and debug the external commands as separate programs. Personally I prefer to use a Sun-workstation with editor, compiler and debugger from the GNU project, but this is not important. The big gain is that it is a UNIX-system with all the programming aids that are found there. I can also use program generators like *lex* & *yacc* [Aho86a].
3. Move the external commands to MPW
   Move the programs to Macintosh and MPW. This is not very difficult, except for some problems with missing libraries.
4. Link with a new library
   Compile and link it as a common tool to some file, but link with the library *XCMDRuntime.o* instead of the libraries *CRuntime.o*, *StdIoLib.o* and *CInterface.o*. Please note that this makes the tool unrunnable!
5. Convert to a XCMD resource
   Convert the CODE resources to a XCMD resource with *ToolToXCMD* and place it in the previously created stack.

Voila! The new functions can now be used in the stack as HyperCard functions.

3.1. Using HyperCard for making a prototype

HyperCard is the main reason for me considering to use the Macintosh at all. It is an application that Apple sends out bundled with every new Macintosh sold. It can best be described as a tool to organize information in a simple and graphically appealing way.

HyperCard has solved a lot of problems with having a user interface that can be programmed graphically. The main entity is the card. Every card contains a number of fields and buttons, where fields contain the stored information and buttons initiate some action. A number of cards are then collected together into a stack, and a stack in HyperCard is stored in a separate file.

HyperTalk [Sha88a], the programming language that HyperCard uses, is simple and sort of object oriented. Every line of code sends a message to an object in the hierarchy. The objects can then specify handlers for these messages.

With hierarchy I meant that the messages is always sent "upwards". It starts on a button or fields and moves up to the card, the background, the stack, the home stack and finally HyperCard itself.

There are a lot of system messages sent when the user does something. For example, when the user hits the mouse button a *mouse down* message is sent to the object under the cursor. Note that this can be a button or a field, but also the card if it does not point at anything in particular. The first object that has a handler for the event is then invoked, and that handler sends new messages upwards. It all ends in a series of messages that reaches the application itself.

This language is easy enough to program for simple tasks, but becomes very slow on larger tasks. In some way it has the feeling of an old basic-interpreter for larger projects, slow and hard to understand.

3.1.1. External HyperCard commands

There is, however, a way to use compiled code from other languages in HyperCard. They are called external commands and are stored in a resource of type *XCMD* or *XFCN* in the resource fork of the stack. The difference between *XCMD* and *XFCN* is that *XFCN* 's are functions that returns a value, while *XCMD* 's are procedures that perform some action. These resources should contain executable code with start point at the first byte. When HyperCard doesn't find a handler for a message on the present background it looks in the stack for an XCMD resource with the same name as the message. If that is found a pascal style function call to the code in the first byte of the resource is made with one argument, the pointer to an argument block that is declared as follows:

```
typedef struct XCmdBlock {
    short   paramCount;
    Handle  params[16];
    Handle  returnValue;
    Boolean passFlag;

    void    (*entryPoint)();
    short   request;
    short   result;
    long    inArgs[8];
    long    outArgs[4];
} XCmdBlock, *XCmdBlockPtr;
```

The three first arguments deal with the parameter passing. *paramCount* is the number of parameters and *params* represents the parameters. They are stored as pascal strings (with their first byte representing their length). *Handle* is a type that is special for the way Macintosh uses its heap. A handle is a pointer to a so called *master pointer*. This master pointer then points at a relocatable block of memory in the heap. When the heap gets too fragmented the memory manager can reorganize the heap and only change the master pointers.

returnValue is the value the XCMD or XFCN returns to HyperCard. All parameters and the returnvalue are ASCII strings. *Passflag* should be set to true if the message shall continue up in the hierarchy after this XCMD has returned.

The rest of the parameters deal with a kind of callback capability. *EntryPoint* is an address of a procedure in HyperCard for a number of interface routines. In my system there is an include file, *XCMDglue.h*, that takes care of defining a more natural interface.

One problem with writing your own XCMD is the way Macintosh treats global variables and CODE resources. Normally an application stores its code in resources of type CODE and entrypoints to functions in them are pointed at in the jump table. (See Inside Macintosh for a more detailed explanation.)

This jump table and the area for global variables is pointed at by the processor register A5 and it is up to the compiler to put the global variables in this area. Unfortunately this area can't be used for an XCMD when it is running because A5 points at the global variables of HyperCard itself and it would be very unwise to do anything at all in this area.

Even more unfortunate is it that the MPW C compiler treats constant strings as global variables. It is not fun to write programs without any global variables or constant strings. This is exactly why I developed the *ToolToXCMD* program that cheats HyperCard into creating a new area for the *XCMD* 's global variables. How this is done is described in section 3.4.

3.2. Special considerations when writing external HyperCard commands under UNIX

One problem is that MPW does not have all the nice library routines found on "real" systems so one has to be a bit careful about which library routines that are used. It is often easier to write the missing ones than to rewrite the code, especially if the port is to be made of already working programs (i.e. the sed port described in the first example). I have also tried to collect some (in fact all I could lay my long fingers on) free or public domain procedures in a separate library.

Another problem that has to be solved is the other way around. How shall I debug the interface through the callbacks to HyperCard. I solved that problem by simply ignoring it. I just wrote trace information to *stderr*. It would probably be better to have some library routines that simulates the HyperCard behavior.

3.3. The Macintosh Programmers Workshop (MPW)

MPW [Inc85a] (Macintosh Programmers Workshop) is a development system with the feeling of UNIX. Part of the reason I used it was the libraries that contain functions that are equivalent to the system calls in UNIX V7.

It is also a nice feeling to have a familiar environment to work in. The key window is something called a shell. In fact every window recognizes shell commands, and tries to interpret the current line when the enter key is hit. Strange error messages are seen when the program tries to interpret your documentation!

The MPW shell does not have the UNIX environment to work in so there has to be some compromises with the appearance of the interface. The main application is the Shell, and this is the Macintosh application (program) that is the nucleus for the other functions.

Commands in MPW are separate executable files, as in UNIX, called Tools. The main difference is that MPW tools can not take any advantage of the process abstraction available in UNIX and that leads to a shell that must deal with the loading of CODE resources, sequencing of commands and the other tasks needed to run the commands.

Pipes are created by sequencing commands and storing the intermediate results in temporary files. The system calls used on the UNIX system are not available either. This must also be solved with intermediate files. Make, for instance, looks up the dependencies and writes a file with commands for later execution.

Programs are not directly portable, but it is not so much work involved to make it inpractical. The makefile has a different syntax and some commands have other names, but it is fairly simple to port commands to this system. MPW is enough for my needs in this project, since I just use it for compiling already debugged programs and it is rather simple to create the needed makefiles.

3.4. *ToolToXCMD*, the solution to all the intricate problems with writing XCMD's.

ToolToXCMD is the main program that I have created. It takes the object code of a program compiled with the usual UNIX conventions for arguments (*argc*, *argv* pair) and converts it to a HyperCard XCMD. To understand the function of the final XCMD we must first take a deep look at the inside of the object code format on a Macintosh.

3.4.1. Macintosh object code format

The executable code on a Macintosh is contained in a number of resources of type CODE and they are loaded as needed and dynamically relocated. When an application first starts, the CODE resource with ID 0 is loaded. This resource is special and it contains pointers to all entrypoints in the other CODE resources. All jumps to code in other code segments are done to a slot in this table. When a resource is not loaded, this slot contains a trap to a system call that reads in the CODE resource and fills in the slots with absolute jump instructions to the procedures in the resource. The next time a procedure in the resource is called, the previously calculated jump is executed.

When an application starts, the first thing it does is to load the CODE resource 0. Then it jumps to the first slot and finds a trap that reads in the CODE resource that contains the start symbol.

This is a fast way of starting up applications, but the application programmer has to deal with the splitting of his code into separate resources and also make decision on when they are no longer needed and should be written back to the disk. Various kinds of overlays have also been found on some older UNIX systems, but generally this can be more efficiently done with hardware and I think that it has been done in this way to keep the production cost down to a minimum.

Global variables is another issue that has to be solved when programs can float around. Apple solved this problem by reserving a processor register, A5, as a pointer to a block in memory that is used for the global variables. All references to global variables are references through this register. This leads to a limit in the maximum static area to 32 Kbytes because the processor only uses 16 bits for displacements.

3.4.2. Inner structure of an XCMD created with *ToolToXCMD*

It takes quite delicate cooperation between the code written in assembler and the code written in C to create a working XCMD from the object code in the tool.

The structure of an XCMD created with *ToolToXCMD* is divided in 3 parts:

- XCMDinit

  An assembler procedure that builds the global data areas and the jump table. This code is the same for all XCMD's.
- XCMDdef

  Some constants that define the size of the global data, the number of entries in the jump table and the name of the XCMD. There is also a table after the constants with one word (4 bytes) for each entry in the jump table. These are offsets from the start of the constant area to where the jump should be to.
- CODE resources

  A merge of all the CODE resources.

Let's take a deeper look at some parts of the XCMDinit code. This assembler procedure is responsible for the creation of the memory areas that is needed. It should also do some other booking operations.

```
Start   MAIN                            ; This is the entry point of the xcmd.

        ENTRY   startUser               ; Start of user data.
        IMPORT  XCMDmain                ; Main xcmd function

* Create a temp area on the stack for local variables.
        WITH    StackFrame
        LINK    A6,#frameSize           ; Set up the frame pointer.
        MOVEM.L D2-D7/A2-A4,regSave(A6)  ; Save the other registers
```

This is the starting point in the XCMD. Sets up a frame pointer in the calling convention for the call to the C programs.

```
* Create a new data area on the heap for the global data and the jump table.
        LEA.L   startUser,A3            ; A3 points at the constants just after this code
        MOVE.L  Xcmd.AboveA5(A3),D0
        ADD.L   Xcmd.BelowA5(A3),D0     ; Size of global area in D0
        ADD.L   #xenv.size,D0           ; Size of extra space needed.
        _NewPtr                         ; Allocate a new block on the heap.
                                        ; Pointer in A0 and result code in D0.
* Check for errors.
        CMPI#noErr,D0
        BNE.S   XcmdError               ; Jump on error.
```

NewPtr is a system call that allocates some memory and returns a ponter to it. Note that this is not a handle through a master pointer. This memory block is static.

Next comes some savings of old register values that has been deleted here. The only interesting part here is the change of the callback routine back to HyperCard. This new routine is responsible for setting the A5 register back to point at the global area of HyperCard before calling HyperCard.

```
        MOVE.L  A5,A2
        ADDA.L  Xcmd.JmpOffset(A3),A2   ; A2 points at the start of the jump table offsets.
        LEA Xcmd.jmptable(A3),A1        ; Address of jump table offsets in A1
        MOVE.L  Xcmd.jmpSize(A3),D0     ; Number of jump table entries in D0

@0                                      ; Build jump table
        MOVE.W  #0,(A2)+                ; Put a zero in the SegNo entry;
        MOVE.W  #JMPOP,(A2)+            ; Place the opcode for a JMP.
        MOVE.L  (A1)+,D1                ; Get the relative offset address.
        ADD.L   A3,D1                   ; Absolute address to entry in D1
        MOVE.L  D1,(A2)+                ; Put address into jump table.
        SUBQ.W  #1,D0                   ; Decrement the number of offsets.
        BNE.S   @0                      ; Loop if not zero
```

Move the entries from the jump offsets here to the jump table and add the address of startUser to them in order to make valid jump addresses in the jump table. Each entry in the jump table consists of 8 bytes.

```
        PEA.L    Xcmd.cmdName(A3)         ; Push the name on the stack.
        MOVE.L   XCMDPtr(A6),-(A7)        ; Push HyperCard block on the stack;
        MOVE.L   A5,A2
        ADDA.L   Xcmd.jmpOffset(A3), A2 ; Find the address of the first entry in the jumptable.
        JSR 2(A2)                         ; Call the function.

* Release the data and return to HyperCard.
        MOVE.L   A5,A0
        SUBA.L   Xcmd.BelowA5(A3),A0      ; Address of the memory for global data.
        _DisposPtr                        ; Dispose it

* Check for error.
        CMPI#noErr, D0
        BNE.S    XcmdError
```

Create a new set of parameters on the stack for the main function *XCMDMain* in the runtime library *XCMDRuntime* and call the first entry in the jump table. This is the entry to the function *XCMDMain*.

The special treatment of callbacks to HyperCard is handled with this routine that temporarily restores register A5.

After this first code comes a parameter block that is calculated by the *ToolToXCMD* command and linked in just before the jump table.

```
****   Define the block that is placed last in the code.   ****
Xcmd      RECORD  0    ; Place this record after the code
cmdName       DS.B32   ; 32 bytes   The name of the XCMD with a terminating
                       ; zero. Pointed at by argv[0].
aboveA5       DS.L 1   ; 4 bytes    "Above A5" size (From CODE 0)
belowA5       DS.L 1   ; 4 bytes    "Below A5" size (From CODE 0)
jmpsize       DS.L 1   ; 4 bytes    Number of entries in the jump table
jmpoffset     DS.L 1   ; 4 bytes    Offset to the jump table from
                       ; location pointed to by A5.
jmptable DS.L 1        ; Jump table offsets (four bytes each)
; Each offset is a count of bytes from startUser to the address to witch
; the jump should be to.
              ENDR
```

This data-block is used for the inits local storage and it is placed just above the jump table for the XCMD.

```
xenv      RECORD  0
oldA5         DS.L1    ; The old value of A5.
oldA6         DS.L1    ; The frame pointer of this function.
toHyperCard DS.L1      ; The entry back in to HyperCard.
size      EQU      *   ; Size of this data area.
              ENDR
```

3.4.3. How *ToolToXCMD* works

The inner structure of *ToolToXCMD* is rather simple, it just has to calculate some information about the CODE resources and then concatenate them with the init resource and the parameter block in the output resource.

3.5. XCMDRuntime, a new runtime library

The runtime library consists of two major parts. One that remaps functions in the old library to new ones. The other part is a set of new functions that implements a simplified view of the callback routines.

3.5.1. Remap of old functions

Only two functions are vital to change, main entry point and exit. I have also written new implementations of *onexit* and the *read* and *write* functions for convenience reasons. Lets start looking at *XCMDmain*, that is the function that is called from the first assembler part:

```
XCMDmain(paramPtr, cmdName)
    Ptr      paramPtr;    /* Input parameters */
    char*cmdName;         /* Name of the XCMD. */
{   /*
     * This is the entrypoint in the new xcmd.
     */

    int i;
    extern  int main();

    _DataInit();             /* Initiate the global variables */
    XCMDinit(paramPtr);      /* Save the pointer to the parameter block. */
    LockXCMDParams();        /* Lock the XCMD parameters in the memory. */
```

The first function to call has to be *_DataInit.* This function is responsible for the initialisation of the global data area. It is put (by MPW) in an own CODE resource called "%A5Init" whose only purpose is to move the rest of the resource to the area pointed at by A5. This resource is linked with the other and doesn't require any special treatment except for the call to *_DataInit.*

```
    /* Build argv. */
    argc = XCMDParamCount()+1;
    if ((argv = (char **)NewPtr((Size)((argc+2)*sizeof(char *)))) == NULL) {
        /* Out of memory. */
        SendCardMessage("answer '!!! Out of memory.'");
    } else {
        argv[0] = cmdName;
        for (i=1; i<argc; i++) {
            argv[i] = XCMDParam(i-1);
        };
        argv[argc] = NULL;   /* Last entry is a NULL pointer. */
```

The arguments is already null terminated strings, but the argv vector has to be constructed. LockXCMDParams has already locked the strings in memory (they were referenced through handles before) and they can be dereferenced without any danger of moving around.

```
        /* Call main */
        if (setjmp(_jmpEnv) == 0) {          /* Return from setjmp. */
            /* Call the main function. */
            (void) exit(main(argc, argv));   /* Always exit with exit(). */
        } else { /* Return from longjmp. */
            DisposPtr((char *)argv);         /* Give the argv vector back. */
        }; /* if setjmp. */

    }; /* if out of memory */

    UnlockXCMDParams();                      /* Unlock the parameters. */
};
```

Main is called in a way that exit always is called. Exit just calls the functions registered with *onexit* and performs a longjump back here. The memory main has claimed for argv is then released with *DisposPtr* before the main exits.

Read and write was changed to remap the standard streams into a more suitable form. This was just done to ease the transformation of standard programs that read *stdin* and writes a result to *stdout.* The only change needed to these programs in this respect is to prepend them with a call to *set_stdin* that register a string to read for stdin, typically one of the arguments. *Stdout* collects the output in a string that is returned from the XCMD when it exits. *Stderr* is a bit special in that respect that it presents each write call with the HyperCard function "ansver", that shows the message in a popup window and requests the user to hit an OK button. All other read and write calls is mapped to a call to the original calls.

3.5.2. New calls

The set of new library calls are included is used as glue to the callback procedure and an interface to the other fields in the parameter block. I started with a package written by Dan Winkler for ADPA.† This package was written in pascal and translated to C. The main difference for the user of my functions is that I have the parameter block saved as a global variable and that all string parameters follow the C conventions.

† ADPA stands for Apple Programmers and Developers Association.

In the original files the parameter block has to be sent as the first parameter and the string parameters were as sent as they were to HyperCard. It took quite a long time to figure out what kind of strings were expected in each call. I think it is better to do this once in a library and take the extra time penalty required to convert the strings.

3.5.3. Tampering with object files

Since I did not want to rewrite all the other library functions, I had to have some possibility of extracting the other functions from the original library files.

In UNIX, libraries are just a collection of files in an archive. The program *ar* is used to delete obsolete files and to add new ones. MPW has a different approach, libraries are just object files. The archiver supplied with MPW is just capable of combining a set of object files to new ones and I want to change name of functions and other properties as well. The solution I came up with was to write an object code editor called *objedit.*

Objedit can change some of the flags and even the names in object modules and other references in object code files. This is indeed a dirty trick and I can understand why Apple did not supply such a program in the distribution of MPW.

It is dirty, but in this case very useful. The building process of my runtime library is to simply catenate the old library with my new functions and change the entrypoint to *XCMDmain*. The names of the old *read* and *write* is also changed to *old_read* and *old_write*.

4. Examples

I have three different examples of how I've used the system. Each of them describes some advantage of developing programs in UNIX and to move them to HyperCard.

4.1. Common programs in a different environment

A lot of commonly used programs in UNIX can be used in HyperCard too. The new IO library defines a new read and write pair that checks for the standard streams. Standard output is collected for the return value, standard error gives a window with a warning message for each line and standard input is taken from one of the parameters. This approach makes it possible to convert standard programs with very few changes.

A tool that I have compiled this way is the stream editor *sed.* The only change that had to be made for the XCMD-package was some small changes to the parameter parsing for a different set of parameters. The original code was from the GNU project and most of the other changes were to make it fit into the MPW environment. One reason for converting *sed* is that the editing capabilities of HyperTalk are both slow and not very easy to use. At least for the kind of editing that *sed* is written for.

The only problem I have seen so far is in the file functions that don't work properly. Pretty good for such a large and complicated program I think.

There is however one other problem with this implementation that is not very obvious, the problem of memory leakage. *Sed* and many other programs allocates memory with *malloc* without explicitly freeing it afterwords. They count on the process handling of UNIX to clean up the memory after the program has died.

In my XCMD this is not automatically true, when the XCMD returns, all memory claimed with it is still claimed and will never be released. If the XCMD is called a lot of times, the HyperCard application will eventually run out of memory.

An American netfriend of mine (David Grossberg, grossber@paul.rutgers.edu) solved this problem for the package by writing new versions of *malloc* and *free*. They handled a list of pointers to all memory allocated with malloc and not yet released. *Malloc* registered the function to be called on exit that did free all memory that was still hanging around.

4.2. Using program generators like Lex and YACC

The GNU project uses a documentation system that has only one file for both a printed manual and online help. The online help consists of *information* files that can be read by some information reading program or with GNU emacs. The latter is the most common.

Info files are divided into a tree of nodes with infomation and menus that can refer to nodes further down the tree. This tree structure is not necessary, other structures can be implemented as well.

This looked very much like a HyperCard stack to me, with each node being a card, and menus for references to other cards, so I decided to make a function that reads an information file and creates a stack out of it. To design the stack was simple but to interpret the text in the *information* files is much more difficult. This is a typical job for a scanner generated with *lex* [Aho86a].

It would probably have been much more complicated in raw C code, and very impractical to write in HyperTalk. Now let's look at the HyperTalk handler in a button that uses this XCMD:

```
on mouseUp
  -- Create the new stack
  set cursor to busy
  set lockMessages to true
  go to card "Template" -- This card contains the image of a node-card
  put long name of this stack into fromstack
  put "Select a new name for the stack"
  doMenu "new stack..."
  put empty
  if long name of this stack is fromstack then
    exit mouseUp
  end if
```

This is done to open a new empty stack that is a copy of the background of this stack.

```
  -- Initiate it
  set cursor to busy
  put long name of this stack into tostack
  go to first card
  set cursor to busy
  set name of this card to "Top"
  put empty into field "Menus"
  put empty into field "MenuNodes"
  put empty into field "Notes"
    ---- Plus an extra 24 lines of similar code ----
  if thisfile is empty then
    go to first card of fromstack
    exit mouseUp
  end if
  set cursor to busy
  hide message box
```

The word volume in HyperTalk leads ones thoughts to the Cobol language. Despite that, it is rather simple to understand and write.

```
  -- This is the call to the XCMD that reads a stack
  readinfo thisfile
```

This is the place where my XCMD is called. It reads the file *thisfile* and creates new cards and fills them with information with the help of callbacks to send messages to HyperCard. All that is left to do, after this, is to link the new stack to the root stack.

As can be seen, there is still a lot of noise around the call to *readinfo*, but most of it is just interactions with the user to find out the parameters. *Readinfo* could have been done entirely in HyperTalk, but it had been very slow and much longer that way.

4.3. Larger projects

Larger projects can also have a face-lift with a more graphical user interface. One project that I have tested is a set of tools for assembling code to a network of transputers [Inma]. The toolset is:

tas – Transputer Assembler
: Assembles the source to an object file. This assembler was originally written by Inge Wallin.

art – Archiver for transputers†
: This is the archiver that was written in the GNU project. The only difference against the original is that *art* uses a different set of magic numbers to recognize an object file.

† The natural name for this command should have been *tar*, but this name was not chosen for obvious reasons.

tld – Transputer linker

Takes a number of object files and libraries and links them to an executable source. *Tld* is also taken from the GNU project *ld*, but this command needed a total rewrite due to a completely different approach in the layout in the object files.

Object files in *tas* is in essence just a compressed and (from syntactic errors) checked form of the assembler code. The actual calculating of references and the converting to executable code is done at load time.

It can be seen that the functional splitting that feels natural in UNIX also works when splitting in separate XCMD's. This kind of stacks needs a lot of thinking before deciding what to put in external commands and what to write in HyperTalk. The goals I use is to do as much of the "real work" as possible in external commands and use HyperTalk just as glue routines.

References

[Aho86a] Alfred V Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques, and Tools*, Bell Telephone Laboratories Inc., Canada, 1986.

[Fri87a] P Fritzson, *Window System Architectures - an Overview*, Dept. of Computer and Information Sience, Linkoping University, Linkoping, Sweden, 1987.

[Inc85a] Apple Computer Inc., MPW – Reference manual, Cupertino, California, 1985.

[Inm88a] Inmo, *The Transputer Databook*, Inmos Limited, 1988.

[Inma] Inmos, *The transputer instruction set - a compiler writers' guide.*

[Ker78a] Brian W Kernighan and Dennis M. Ritchie, *The C Programming Language*, Bell Telephone Laboratories Inc., Canada, 1978.

[Ros85a] Caroline Rose, *Inside Machintosh*, Addison-Wesley, Cupertino, Californa, 1985.

[Sha88a] Dan Shafer, *HyperTalk Programming*, Hayden Books, Indianapolis, Indiana, USA, 1988.

[Tan87a] Andrew S Tanenbaum, *Operating Systems Design and Implementation*, Prentice-Hall Inc., London, 1987.

[You89a] Douglas A Young, *Window Systems Programming and Applications with Xt*, Prentice-Hall, Inc., 1989.

Weaknesses in Shared Memory and Software Interrupts in UNIX

D L Jenkins
R R Martin

University of Wales College of Cardiff
Department of Computing Mathematics
PO Box 916
Cardiff
CF2 4YN
Wales
United Kingdom

djl@cm.cf.ac.uk
ralph@cm.cf.ac.uk

ABSTRACT

The analysis of a sketch requires an application capable of recognising basic graphics primitives, for example lines or circles, and determining geometric relations that might exist between them. However, sketching is highly interactive requiring any application to be very responsive during input whilst the sketch analysis is given a lower priority. This paper presents a brief description of a sketch analysis program called Easel, an overview of possible architectures, UNIX mechanisms to support them, and a detailed discussion of two implementations. In addition, the problems associated with supporting these models with UNIX is then outlined.

1. Easel

Outline

Easel is a sketch editing program implemented on a Sun 3 workstation under version 4.0.1 of SunOS. It is highly interactive, not only allowing the user to draw a free hand sketch with a mouse, but also simultaneously analysing the picture in an attempt to understand what has been drawn. As analysis progresses, the original sketch is replaced by a more geometrically exact diagram which can then be re-edited by user.

Program Objectives

First, Easel must behave in a natural manner which mimics as closely as possible sketching with pencil and paper.

Secondly, analysis must not interrupt the user whilst sketching, either too frequently or for too long.

Thirdly, when the user is idle, CPU use must be maximised so as to produce the results of analysis as quickly as possible.

Sketch Structure

From the application's point of view, each sketch is composed of a large number of objects, with each object consisting of a number of strokes or lines. Each stroke is made up of the original (x,y) points on the line drawn by the user, and is defined as a mouse button down action, followed by a dragging of the mouse to form the desired shape on the screen, terminated by a mouse button up action (see figures 1.1 and 1.2).

Architecture

Ideally, running simultaneously with the sketch editing process, a second process continually examines the current state of the picture performing some basic filtering of each stroke, and then attempts to fit a pre-defined primitive, for example a straight line, or a circle, to the original (x,y) data. Then, in the most intensive processing stage, strokes are compared pair-wise to determine whether geometric relations such as perpendicularity exist between them.

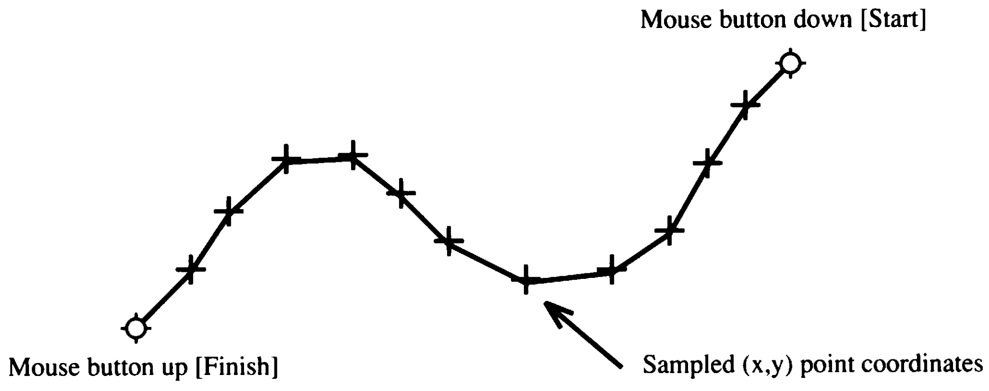


Figure 1.1: *Composition of a single stroke*

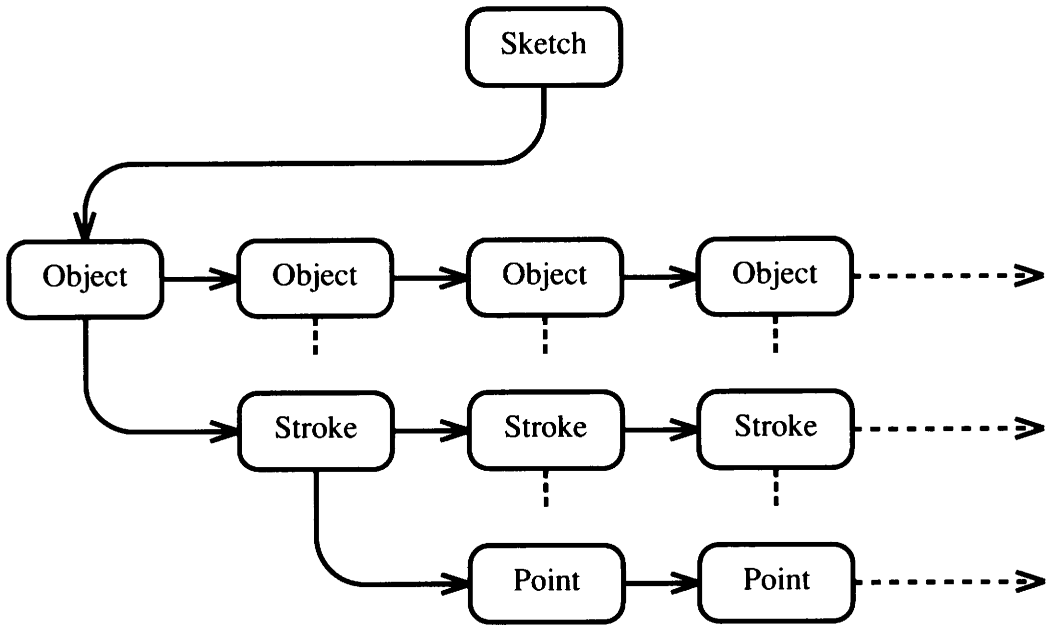


Figure 1.2: *Data structures associated with each sketch*

2. Overview

2.1. Potential Program Architectures

Possible Models

The basic architecture requires two tasks working on common data. However, several different alternatives are possible involving one or more processes, and one or more copies of the sketch data. In this section, some possible approaches to this problem are presented at a high level.

2.1.1. Two processes, two data structures.

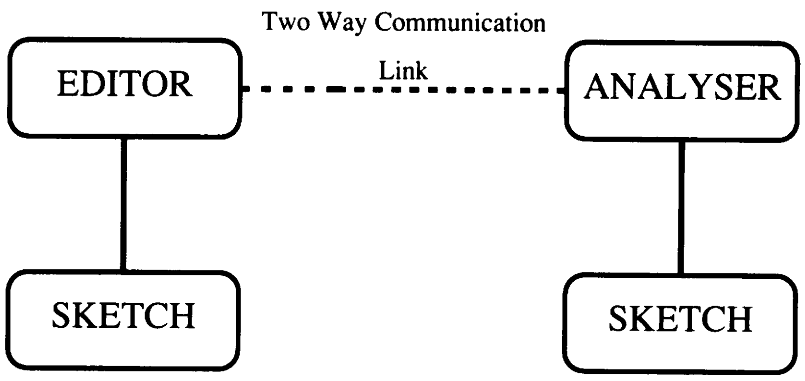


Figure 2.1: *Model with two processes and two separate data structures*

In this model (see figure 2.1), the editor and the analyser are separate processes, and each creates and maintains its own independent model of the sketch. Any changes one process makes to its model of the sketch is then broadcast to the other, which in turn updates its model to ensure consistency.

Advantages

- This model is ideally suited to a parallel or distributed implementation since both programs are completely independent.

Disadvantages

- Maintaining the consistency of the two models.
- Alterations to the model must be done in the correct order so as to prevent, for example, one process accessing a piece of data already discarded by the other.
- The bandwidth of the communications link may be too slow, especially if the programs are running on two machines which are part of a network with a high data flow.
- Maintaining the integrity of the sketch may prove difficult, if one process fails without informing the other of some the alterations it has made to the sketch.
- Creating and maintaining two separate sketch models wastes storage and puts an unnecessary burden on a single CPU machine severely reducing the performance of the workstation. In addition, on a machine with limited memory, processing is lost during swapping, and supporting the communications overhead.

2.1.2. Three processes, one data structure.

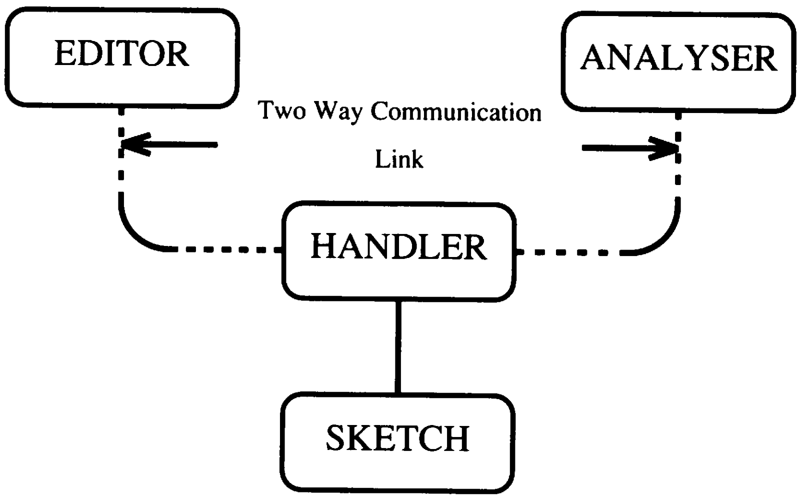


Figure 2.2: *Model with three processes and a single data structure*

In this model (see figure 2.2) a third process called the handler receives updates from both the editor and the analyser processes, and then carries them out on a single model of the sketch which it creates and maintains. Only the sketch handler has direct access to the model.

Advantages

- Again this model is suited to a parallel or distributed implementation.
- The single model of the sketch eliminates the problem of consistency present in the previous architecture, and also maintains the integrity of the sketch if either the editor or the analyser fails.

Disadvantages

- This model also has the problem of synchronising alterations present in the previous model. In addition, some contention policy has to be decided so as to prevent simultaneous accesses to the sketch by both the editor and the analyser.
- On a single machine with limited memory, three programs running simultaneously would increase the processing required for swapping, hence reducing the speed of the entire system.
- The network bandwidth problems found in the previous model are also present in this model. This is especially true if each of the three process are made to run on separate machines.

2.1.3. Two processes, one data structure.

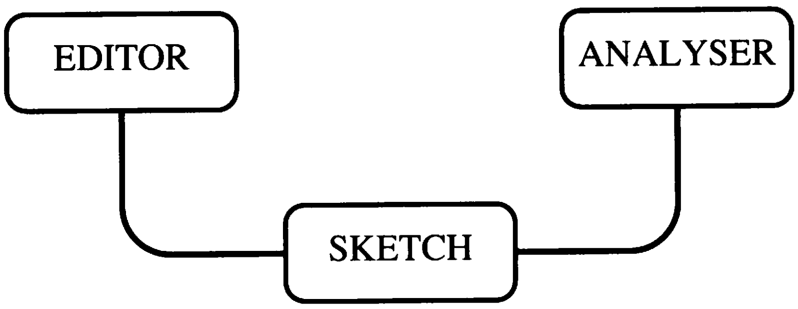


Figure 2.3: *Model with two processes and a shared data structure*

In this model (see figure 2.3), the two processes, which must reside on the same machine, share a common sketch data structure which both of them can access.

Advantages

- The functional independence of editor and the analyser allow them to be implemented as separate processes on a single machine.
- Since there is only one model of the sketch, there are no consistency problems to cope with.
- No network communications problems.
- This is easier to implement compared to the other models, since there is no handler required, nor is there a need for code be written to support the communication link.

Disadvantages

- The difficulty of synchronising alterations present in all the models.
- The need to share a common sketch model means that this model is not ideally suited to a distributed implementation across several machines.

In all of the above models, some method of triggering the analyser needs to be developed, as does a means by which the analyser can be locked out whilst user interaction takes place.

2.2. Overview of UNIX Mechanisms For Supporting These Models

Fork()

The UNIX system call *fork()* [Sun88b] allows one process to create a new child process. This new process is an exact copy of the calling process except that the child has a unique process ID. The child has its own copy of the parent's descriptors, which reference the same underlying objects, so that for instance, file pointers are shared between the parent and the child so that an *lseek()* on a descriptor in the child process can affect a subsequent *read()* or *write()* by the parent.

Inter-process Communication (IPC)

A *pipe* [Ker84a] is a way to connect the output of one program to the input of another program without using any temporary files. A pipeline, therefore, is a connection of two or more programs through pipes.

Signals [Ker84b] are simple interrupts which are passed to processes on an occurrence of a particular event, for example a mouse move.

Sockets [Eng87a] are defined to be bidirectional end points of communication. Names may be bound to sockets and each socket in use has a particular type, and one or more associated processes. Sockets reside in communication domains, two of which are supported by SunOS: the UNIX domain, which allows processes to communicate on the same machine, and the Internet domain, which allows processes to communicate across a network.

Inter-process Communication [Eng87b] effectively means *sockets*, since both *pipes* and *signals* are restricted either in terms of flexibility, pipes are unidirectional and must reside on the same machine, or in terms of capacity, signals have a limited data capacity.

Remote Procedure Call (RPC)

RPC [Eng87c] is an integral part of Sun's NFS, Network File System, and is built around the client-server model, with both the client and the server being separate processes. Servers must register the processes that are to be made available to other processes on the network. Clients can then execute these by making an RPC. This effectively means that the server and client can be either on the same machine or on different machines. An RPC results in a message being sent to the server, on receipt of which it performs the necessary processing and may return a result, via the network, back to the client.

Interrupts

Software interrupts are provided under SunOS [SPG88a] and allow the user to define a periodic call to a procedure, which must reside in the program being interrupted. After the interrupt has been serviced, control returns to the point in the program before the interrupt was raised.

Shared Memory

SunOS supports a limited implementation of shared memory, in which a process can create a shared memory block which then behaves as if it had been created by the UNIX system call *alloc()*. The sharing processes, which unfortunately must reside on the same machine, may then open this shared memory block for reading and writing.

2.3. Evaluation

A single sketch model would be desirable because of the problem of consistency, and of the extra CPU effort required to maintain two models of the same exactly the same sketch. In addition, to maximise the performance of the CPU, the number of processes needed to maintain the model must be minimised. In the light of both of these points, model 3 appears to be the best solution.

Fork() is hopelessly inadequate at supporting this model. Since the editor and analyser are different programs, *fork()*'s cloning of the parent process is fundamentally unsuited. Also, with the exception of file descriptors, no sharing of data is allowed.

IPC and RPCs provide similar facilities and are well suited to model 1 and particularly model 2 if one of those were chosen, but are not so well suited to model 3 since there is no real communication flow present.

This leaves two possibilities to investigate further: a single program containing the editor, the analyser and the sketch data, with the analyser being triggered by an interrupt issued by the editor; or two programs using shared memory with flags used to ensure synchronisation.

3. The Interrupt Alternative

Interrupts are used in the current implementation of Easel. Under SunView, Sun's propriety windowing environment, it is possible to set up a timeout event which will call a procedure after a set time has elapsed.

In this method a variable known as the *Altered* flag is defined and initialised to false indicating that the sketch has not been modified. If the user then edits the sketch, the altered flag is changed to true in order to indicate to the analyser that the sketch needs to be analysed, or re-analysed.

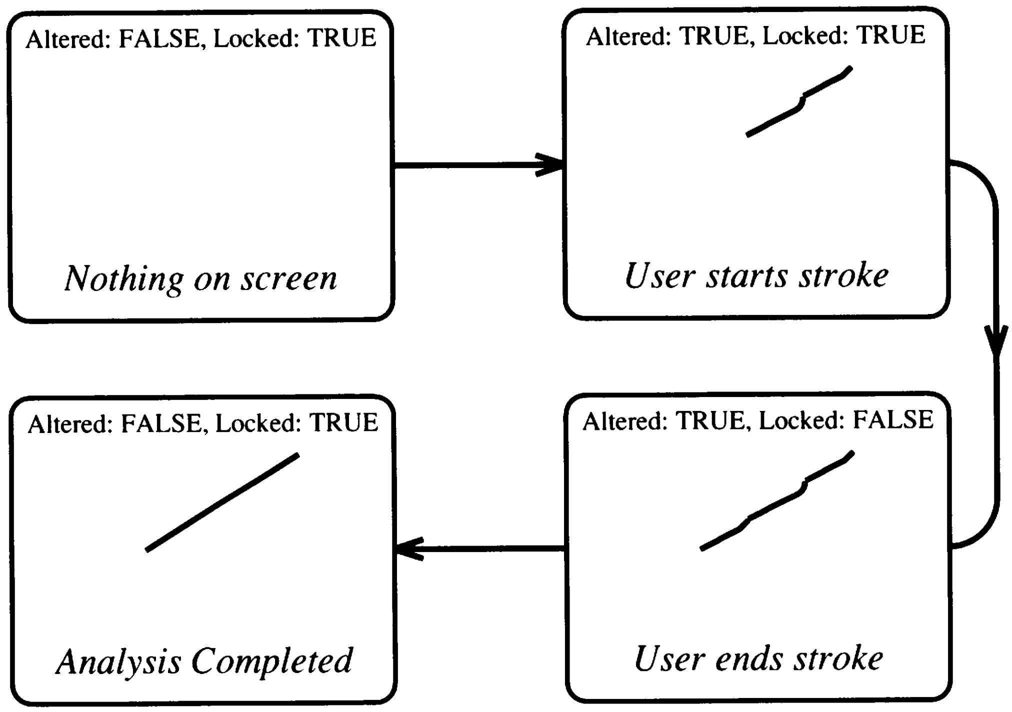

Altered: FALSE, Locked: TRUE
Nothing on screen
Altered: TRUE, Locked: TRUE
User starts stroke
Altered: TRUE, Locked: FALSE
User ends stroke
Altered: FALSE, Locked: TRUE
Analysis Completed

Figure 3.1: *The sequence of events during an analysis interrupt*

A timeout event is set up under SunView so that every 0.1 seconds an interrupt occurs and the *Altered* flag is examined. If the *Altered* flag is true, then Easel starts the analysis procedure. However, to prevent Easel from analysing the sketch whilst the user is drawing, a second variable, the *Locked* flag is maintained. This acts as a sketch lock preventing analysis from starting until the user has finished manipulating the sketch. Without this, analysis may start when the user is in the middle of an operation, for example, drawing a line, or moving a circle and so on. The *Locked* flag is initially set to true indicating that the sketch is locked and becomes false only after the user has completed an operation (see figure 3.1). However, during analysis, control effectively passes from SunView, which handles all the input, to the analysis routines. This means that the user will be prevented from starting a new stroke whilst the sketch is being analysed. This prohibition period will be as long as it takes for the entire sketch to be the analysed.

To avoid unnecessary interruptions and also to make more efficient use of the CPU during periods when the user is idle, a queuing algorithm could be used to improve the basic interrupt method further, as described below.

In this idea the analyser is allowed to operate in *burst mode*, that is instead of attempting to complete the entire analysis in one go, the analyser would be given sole ownership of the CPU for a finite time t_b, where t_b is less than, say, one second and very much less than the time required to analyse the sketch completely t_A. Each task could be high level in nature, that is analysing a single stroke, or at a lower level, be defined as filtering a stroke, fitting a primitive to a stroke, or checking if two lines are perpendicular and so on. A task queue would be then created, initially containing no members, which would be added to as new tasks are created and added to the end of the task queue. Meanwhile, at the front of the queue the analyser would creep along performing each task in turn, thereby removing it from the queue, for as far as it could in the allocated time t_b. After this time has elapsed, ownership of the CPU would then return to the editor and again the task queue would be added to. This cycle repeats itself every time an idle period arrives.

A number of points are raised when choosing the nature of the task. A high level task queue consisting of strokes would mean that the analyser would only have to be informed of the addition of new strokes, and would result in a queue length of $O(n)$. In a low level system, the analyser would have to be informed of the existence whole series of new tasks relating to each new stroke entered leading to a queue length of $O(n^2)$. The lower the level of the queue the more flexible it is. If a stroke is manipulated, for example moved, then it may be that only certain analysis need to be repeated. Hence, using a high level queue results in some unnecessary processing. However, this flexibility is achieved at the expense of a more complicated input routine, which needs to determine how each stroke is to be processed. This appears undesirable, since any complex decision making in the input routine will make the system less responsiveness to the user.

In both cases above, the analyser's position in the queue needs to be remembered, which could be simply achieved by remembering the task, or stroke to be processed next. The problem of handling half-finished tasks when t_b expires presents an additional problem.

Alternatively, the interrupt mechanism could be reversed so that it is the analyser instead of the editor that is interrupted. In this idea, an user action would trigger an interrupt suspending analysis whilst the sketch is edited. During this period new tasks or strokes would then be added to the queue. When the user becomes idle once again, control would be returned to the analyser and the analysis would continue on from the point at which it was interrupted.

Advantages

- This is very easy to implement, since SunView supports timeout events directly making it only a matter of initialising a SunView structure with the initial timer value and the period time value. A procedure name can also be incorporated into this structure which SunView will then call after the timeout period has elapsed.
- For less complex sketches, typically less than thirty strokes, total processing times are a few seconds, making this method is quite suitable for the testing and debugging stages of Easel.
- The use of queuing could dramatically improve the efficiency of CPU use compared to the simple non-queued interrupt, and also reduce the period during which the user is denied access to the sketch.

Disadvantages

- The processing times for more complex sketches result in the user being locked out of the program for a few tens of seconds, which is unacceptable since it conflicts with the goal of making the system behave in a manner as close possible to that of natural sketching, that is with the minimum level of interruption.
- Queuing does alleviate some of the above difficulties, but in turn raises other problems.

  In the burst mode method, the period t_b will need to be far less than the time taken even for the most modest of processing steps. This implies that not only will each of these analysis routines need to be broken down into simpler functions, which may not always be possible, but each of these routines will have to include its own timer support capable of detecting when the period t_b has elapsed and of returning control to the editor.

In addition to the above point, it is possible that the queue will become so long that the analyser will become bogged down. This results from the fact that the queue is growing faster than the speed by which the analyser can remove tasks from the queue. Consequently, some policy has to be decided to ensure that the analyser does not lag too far behind the end of the queue. One obvious method is to encourage the user to take "rest periods", during which the analyser can catch up sufficiently to allow interaction to continue.

Reversing the interrupt mechanism, so that it is the analyser and not the editor that is interrupted, eliminates the first two problems above, but not the last. This solution favours the burst mode method since the analysis is sliced up into the smallest machine entity, that is individual machine instructions. However, reversing the interrupt effectively means that when the user affects the sketch, the editor will be less responsive since the analyser first has to relinquish control, and then transfer it to the editor. In addition, SunView operates on the principle that it is the user who is interactive, and accordingly all of its routines support events associated with the user: moving the mouse, pressing buttons and so on. This fact makes it awkward to reverse the interrupt since more complex code will be required to interrupt SunView whenever analysis takes place. Note also, that the analyser still has maintain flags to prevent the editor from accessing the sketch whilst analysis is in progress.

- It takes no advantage of the natural distributed architecture present in model 3. The sketch editing and sketch analysis parts of the program could be made completely separate, conceivably allowing the two to run on separate machines. Interrupts are unsuited to this model.

4. The Shared Memory Alternative

An alternative approach is to have two separate processes sharing a common data structure. In this idea, both processes would run simultaneously with the editor signaling the analyser when to commence analysis, rather than explicitly calling it, as in the interrupt method.

In UNIX, each running process has its own region of memory allocated to it for data, which is kept separate from another process's data by UNIX's memory management. Consequently, it is impossible to use the memory allocation system call *alloc()* to set up a common memory block to be accessed by more than one process.

Fortunately, some versions of UNIX provide a shared memory facility. In this arrangement, a process creates a shared memory block which can then be accessed by another unrelated process. To achieve this the following steps must be carried out.

First of all, a shared memory segment identifier is found for the shared memory block to be created, using the UNIX system call *shmget()* [Sun88c].

```
        SMSI = shmget ( Key, BlockSize, Flags );
```

`SMSI:`	is the shared memory segment identifier associated with *Key*. If *shmget()* fails SMSI will be set to −1.
`Key:`	is a positive integer number of type *key_t*, which is unique to this set of the sharing applications.
`BlockSize:`	is the size of the requested common memory block in bytes.
`Flags:`	indicates the type of operation to be performed by *shmget()*. For block creation, this should be initialised to *IPC_CREAT|0600*. The "0600" refers to the permission on the shared memory itself, that is in this case read and write by the user only [Sun88d], in a similar way to the UNIX file modes.

Having allocated this common memory block, it now has to be attached to the process's data segment. This can be achieved by using the UNIX system call *shmat()* [Sun88a].

```
        Location = shmat ( SMSI, 0, 0 );
```

`Location:`	is a pointer to the start of the shared memory segment relative to this process's data segment. If *shmat()* fails *Location* will be set to −1.

The other process can then open this shared memory block by using the UNIX system call *shmget()*.

```
        SMSI = shmget ( Key, BlockSize, Flags );
```

`SMSI:`	as for creation.
`Key:`	this key must be the same as that used when creating the common memory block.
`BlockSize:`	the size may be smaller that the block created, but cannot be greater.
`Flags:`	this should be set to *IPC_ALLOC* to open the shared memory block.

Next, this common memory block can be attached to this application's data segment by using the UNIX system call *shmat()*.

```
        Location = shmat ( SMSI, 0, 0 );
```

`Location:`	is a pointer to the start of the shared memory segment relative to this process's data segment. If *shmat()* fails *Location* will be set to −1.

A means of synchronising the programs is needed, as in the interrupt method. In the development system this takes the form of including *Altered* and *Lock* flags in the shared memory block, which operate in the same way as in the interrupt method.

In addition, since there are now two processes running simultaneously, some sort of priority can be assigned to each. Priority should be given to the editor since this is the most interactive. This can be achieved using the UNIX call *nice* which lowers the priority of a process.

However, the priority of the process is not the only thing that determines when a process will be run. To determine which process should be ran next, the scheduling mechanism in the UNIX kernel uses a formula that takes into account each process's priority, how much CPU time each process has received recently, and how long it has been since each process has run [Nem89a]. This means that switching between the editor and the analyser will not be as quick as if they were both part of the same program.

In addition, when the load on the system is light, UNIX will use *paging* [Nem89b], that is, it will move out individual memory segments that have not been recently referenced out to a swap partition to free up space. When the load is heavy, UNIX may begin *swapping* [Nem89b] that is systematically hunting through the memory to find all segments that belong to a particular process and then removing all at once, hence preventing the process running for a comparatively lengthy time. UNIX attempts to pick a swapping victim which is a process that is unlikely to run in the near future. This fact means that if the user stops sketching, there may be a comparitively long delay, as the analyser is swapped back into physical memory, before analysis commences.

Advantages

- This is the ideal solution in principle, since both the sketch editing and analysis applications can share a common data structure independently of each other, thus allowing the possibility of simultaneous processing.
- Allocating blocks of shared memory is easy, and the blocks returned can be handled in the same way, as least as far as the creator process is concerned, as if they had been created using UNIX's *alloc()*.
- Prioritising is easy, and puts into effect the emphasis of the sketching part of the system.
- The coding of the editor is easier, since an explicit call to the analyser is no longer needed. Only a signal to start analysis is required.

Disadvantages

- The primary disadvantage of this method is in the way that shared memory is mapped with respect to the application's data segment. For example, say the following C structure exists:-

```
struct Object
    {
    int Data;
    struct Object *Next;
    };
```

When the creator application requests a shared memory block it might be mapped into its own data segment say from memory location 10 onwards (see figure 4.1).

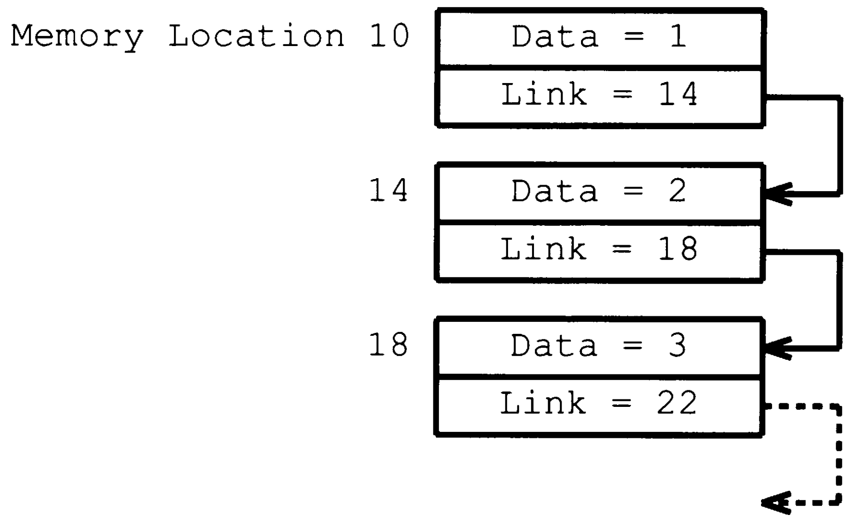


Figure 4.1: *How a linked structure is mapped in the creator application's data segment*

However, when the sharing application now opens the same memory block, it may be mapped to a different address in its data segment, say from location 14 onwards. This means that all the addresses in the pointer fields of the linked structures now no longer point to the correct location in memory (see figure 4.2).

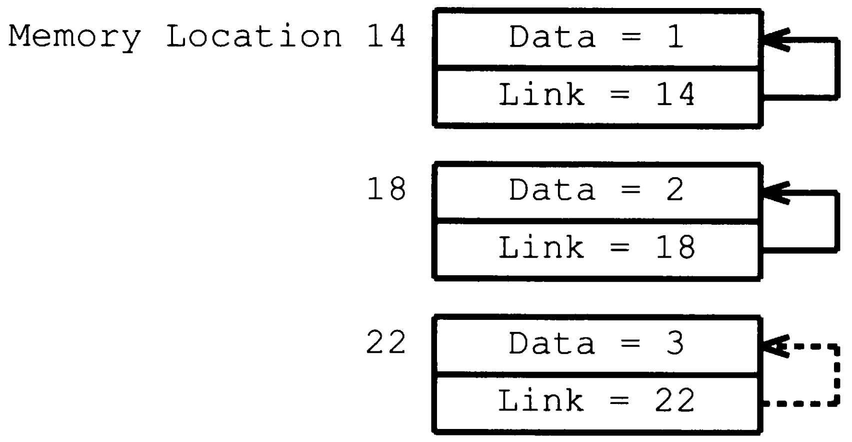


Figure 4.2: *How a linked structure is mapped in the sharing application's data segment*

- This mapping problem means that it is not possible to have a conventional implementation of a linked structure in the sharing application. To cope with this, the offset address of each shared memory block needs to be kept, and some simple mathematics performed to compute the correct address in the sharing application's data segment. For example, in the above case, the offset address of the shared memory block in the creator's data segment is 10, whilst in the sharing application it is 14. Thus to achieve the correct location for the first pointer in the sharing application's data segment we simply add 4 (14 – 10), to the link address (14) to give us the correct address 18 (14 + 4).

  However, this method in turn has its problems. If more than one shared memory block is used then a table of offsets must be kept, which also needs to be stored in a block of shared memory. Due to the addressing problems with linked structures, this table would have to be static in size, meaning that it would have to be sufficiently large to cope with most complex expected sketch. Each link reference would then have to find the correct offset address of its block and then perform the calculation stated above. The combination of these two steps results in a significant increase in access times for linked structures.

- Each reference to a linked structure must be enclosed within a mapping function, for example, suppose *NextObject* and *Current* are pointers to a structure *Object* which is held in the shared memory block.

  ```
  struct Object *NextObject, *Current;
  ```

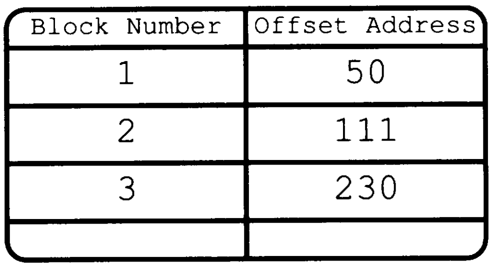
  In a conventional data segment, the following statement would make *NextObject* point to the next node in the linked list.

  ```
  NextObject=Current->Next;
  ```

  This would work in the creator's data segment, but would fail in sharing program due to the addressing problems highlighted earlier. This means that some user defined function, say *MapLocation()*, needs to be included into the statement to return the correct address.

  ```
  NextObject = MapLocation (Current->Next);
  ```

  The purpose of this *MapLocation()* function is two fold. First, it must determine to which shared memory block this address belongs and then return its offset address. Secondly, it must perform the calculation described above to obtain the correct address within that shared memory block. For example, suppose three shared memory blocks exist each of size 50 bytes (see figure 4.3).



Block Number	Offset Address
1	50
2	111
3	230

Figure 4.3: *An offset table in which three shared memory blocks exist*

  Suppose now the mapping function is looking for address 119. It starts examining the first shared memory block. This has a memory range from 50 to 100 (50+50). Clearly 119 is outside this range. Now the second memory range is examined. It goes from 111 to 161, which is the correct block. So *MapLocation()* takes this offset, 111, and uses it in the above calculation to compute the correct address.

- The use of the *MapLocation()* function, in turn, leads to another problem, namely, the sketch editing program cannot share source modules with the analysis program, unless of course, both programs use the *MapLocation()* function to determine the correct address of linked structures. This is inefficient, since in the creator application, all addresses are correctly mapped to begin with. To cope with this, replacing the C function call with a macro would allow a null definition in the creator application and a full definition in the sharing application. This would also improve the speed of the *MapLocation()* calculation. However, since the *MapLocation()* function is going to be extensively used, the size of any program using it as a macro would be much larger, in terms of code length, than would be the case if it were to be defined as a C function.

- Using relative addressing instead of absolute addressing could also be used to improve the performance of the memory allocation routines. Here, a shared memory block is organised with each address stored as a relative offset from the start of the shared memory block itself (see figure 4.4).

Block Address In Data Segment ⟶

Relative Address	Data	Relative Link address
+00	2145	04
+04	3245	12
+08	678	16
+12	12	08
+16	3	20
+20	672	NIL

Figure 4.4: *A shared memory block with relative addressing*

However, both the creator and sharing process will still need a *MapLocation()* function to explicitly calculate pointer addresses, although since all the addresses are now relative, the mapping function is simplified, becoming the addition of the relative pointer address to the address of the shared memory block, for example with the following structure:-

```
struct Object
    {
    int Data;
    int Next;
    };
```

Then the address of the next link will be:-

```
ThisBlockAddress+Next
```

Where *ThisBlockAddress* holds the address of the shared memory block. If there is only one shared memory block in existence, then the contents of *ThisBlockAddress* will be the same for all link address calculations.

However, problems arise when multiple blocks exist — the address of each block still needs to be kept, either in an offset table similar to that described earlier, or possibly as part of the linked structures itself, for example:-

```
struct Object
    {
    int Data;
    int Next;
    int WhichBlockAddress;
    };
```

Where the address of the next *Object* can be computed as:-

```
WhichBlockAddress+Next
```

- Neither the absolute nor relative addressing methods discussed are satisfactory. Each requires modifications of the code to overcome the limitations of UNIX's shared memory, either by inclusion of the *MapLocation()* function, or by embedding extra information into each data structure. Furthermore, the use of the *MapLocation()* function in both cases incurs a performance penalty, and the addition of a block address field in each linked structure wastes storage space. However, the relative addressing alternative does make it easier to share common modules, since both the editor and the analyser require the *MapLocation()* function, but means that the creator process suffers from the same problem of reduced performance and increased storage needs as does the sharing process.
- Once a shared memory block has been opened, the next problem is developing some sort of memory allocation method for efficiently using it. This effectively means that equivalents of *alloc()*, and *free()* need to be implemented. With single shared blocks this is not much of a problem, but greater difficulties arise when attempting to implement a system which can cope with multiple shared memory blocks. The task here is to ensure that cross-block links are correctly maintained, and that only the minimum number of shared memory blocks exist at any one time. Note that each new shared memory block causes an additional iteration in the *MapLocation()* function.

- Selecting a common key for block identification at first seems trivial, but there appears to be no consensus on how to pick a key other than by the arbitrary method of choosing a random number. This leads to a problem. For example, suppose another application has already selected a certain key. Then attempting to open a different shared memory block with the same key will fail. One solution to this is to repeatedly increment the key until successful, and pass the new key onto the sharing application inside a standard file. More desirable would be a key chosen by UNIX which could be, for example, a hash function of the names of the participating applications. Also, if multiple shared memory blocks are requested, then a sequence of keys will be needed, which for the above reason, may not be continuous. This means that these keys, as well as the block addresses, will have to be stored in the offset table.
- Sketches are dynamic, and therefore, it is evident that as the sketch becomes larger the editor will demand and create additional shared memory blocks. This raises a number of issues. First, as each new memory block is created the sharing application needs to be informed of its existence, and on receiving this information has to open the new block and update the offset table. To reduce the number of iterations inside the *MapLocation()* function, the size of each shared memory block needs to be large, typically greater than 100 kilobytes, otherwise as the sketch grows the number of entries in the offset table will become too numerous, resulting in a slowing down of the analyser.

5. Conclusions

UNIX provides a number mechanisms for supporting this type of interactive processing, but unfortunately they either do not work well, or are difficult to use.

Interrupts are straight forward to implement and work well for small amounts of data, but as complexity of the sketch increases, problems with user lockout arise, which effectively make this option unsuitable for the chosen model. Reversing the interrupt appears a neat solution, but its interaction with SunView makes the code more complex and increases the delay when sketching.

Shared memory appears to be the ideal solution and would be if it were not implemented in such an awkward way. An ideal shared memory implementation would include the following:-

- Most importantly, it should handle addressing correctly. There are a number of ways to achieve this: the common memory block could be external to both processes with addresses being mapped to this data segment. More unlikely, changes to the memory management hardware could lead to special flags on shared blocks so that addressing instructions correctly map memory locations. Less desirably, special compiler options which implicitly embed the *MapLocation()* function directly into the code produced.
- A better key arrangement, ideally incorporating the names of the participating processes, and with some mechanism to prevent other processes choosing an identical key.
- Ideally, UNIX should support shared memory versions of both *alloc()* and *free()*, removing the need to request large blocks and implement one's own memory allocation and de-allocation functions.

Acknowledgements

The authors would like to express their thanks to Robert Evans for his helpful discussions, constructive criticisms whilst preparing this paper, and for his time spent converting it into *psroff* format.

References

[Eng87a]	in *A Sun User's Guide*, ed. D England, p. 181, Macmillan Education, 1987.
[Eng87b]	in *A Sun User's Guide*, ed. D England, p. 179, Macmillan Education, 1987.
[Eng87c]	in *A Sun User's Guide*, ed. D England, p. 172, Macmillan Education, 1987.
[Sun88a]	"System Calls," in *SunOS 4.0 Reference Manual*, p. 760, Sun Microsystems, 1988.
[Sun88b]	"System Calls," in *SunOS 4.0 Reference Manual*, p. 661, Sun Microsystems, 1988.
[SPG88a]	in *SunView Programmers Guide*, p. 294, Sun Microsystems, 1988.

[Sun88c] "System Calls," in *SunOS 4.0 Reference Manual*, p. 758, Sun Microsystems, 1988.

[Sun88d] "System Calls," in *SunOS 4.0 Reference Manual*, p. 622, Sun Microsystems, 1988.

[Ker84a] B W Kernighan and R Pike, in *The UNIX Programming Environment*, p. 31, Prentice Hall, 1984.

[Ker84b] B W Kernighan and R Pike, in *The UNIX Programming Environment*, p. 225, Prentice Hall, 1984.

[Nem89a] E Nemth, G Synder, and S Seebas, in *UNIX System Administration Handbook*, p. 60, Prentice Hall, 1989.

[Nem89b] E Nemth, G Synder, and S Seebas, in *UNIX System Administration Handbook*, p. 52, Prentice Hall, 1989.

INGRID: A Graphical Tool for User Interface Construction

L Carriço
N Guimarães
P Antunes

INESC
Rua Alves Redol, 9, 6D
1000 Lisboa
Portugal

lmc@inesc.inesc.pt
nmg@inesc.inesc.pt
paa@sabrina.inesc.pt

ABSTRACT

INGRID is an interactive tool for user interface construction. The tool enforces a specific user interface model that considers both the functional composition of the user interface elements and an object-oriented approach as the fundamental design and development methodology.

The implementation of INGRID highlights three main components: a run-time support system for interactive programming (in C++), a toolkit that defines the abstract interfaces to the several components of the UI allowing integration of multiple graphical toolkits, and the user interface of INGRID itself.

1. Introduction

As user interfaces (UIs) become more sophisticated and easy to use, they also become harder to create. It is clear that the construction of quality user interfaces requires the existence of an iterative refinement cycle of prototyping and validation. Therefore, interactive and easy to use tools for UI construction must be provided. Also, the definition of an adequate architectural model along with a powerful interactive support, are essential, both to guarantee the coherence of an UI design and optimize the iterative refinement process.

This paper presents an overview of a graphical, object-oriented, UI editor (INGRID – INteractive GRaphical Interface Designer). User interfaces built with INGRID are based on a proposed architectural model (4D). The construction process is supported by an environment for interactive programming (ICE).

The first two sections of this paper describe the models adopted for the UI and its construction with our editor. Complete application and interface models are fundamental to ease the task of UI design and construction. The models must be adapted to the requirements of direct manipulation, multi-thread dialogue control and semantic feedback, while keeping a comprehensive interface organisation.

The editor, described in section 3, includes multiple facilities requested by developers such as: graphical construction of interfaces with minimal coding effort; reduced iterative refinement cycle, with on-line testing, verification and validation of prototypes; direct manipulation of resources and attributes; guidance during the construction phase, with help and browse.

The next sections of the paper are dedicated to implementation aspects. Section 4 focuses on the ICE run-time support that offers the required platform for interactive programming. ICE supports most of the interactive features of the editor: on-line creation, customisation and inspection of interface entities, and dynamic support for object inter-communication.

Section 5 describes the 4D toolkit and the way it conforms to the proposed model and supports the editor functionality. The toolkit aims to provide transparent integration of existing and "standard" graphical toolkits allowing multiple options of functionality and "look and feel".

1.1. Background

INGRID is a tool that results from the evolution of the IMAGES UIMS [Sim87a, Mar88a, Sim88a] developed within the SOMIW ESPRIT project [SOM85a] and is a component of a global object-oriented Application Development Environment under development at INESC [Mar89a]. IMAGES provided a valuable experience in the subject of UI creation and gave important guidelines about the developers and user expectations from interface construction tools.

The fundamental conclusions drawn from the work with IMAGES were:

- UI development environments must support "standard" user interface toolkits to achieve a significant degree of quality, user acceptance and avoid replication of existing work.
- The use of specification languages (the IMAGES approach) does not represent a major breakthrough in the process of UI construction, which reinforces user preference for interactive tools.

2. The Models

INGRID assumes two fundamental models in the process of building a UI: a generic model for the whole application and a specific model for the user interface itself.

As currently accepted by users and developers of UIs and UI Development Systems, the generic model considers that an application is separated between an interface component and a computational component (dialogue independence, as defined in [Har89a]). An object-oriented approach to this model leads to the definition of both components as abstract data types. Their interfaces are highly dependent on the communication needs between them.

2.1. The Interface Model

The interface model is fully dedicated to functional aspects within the interface component of an application.

Some preliminary discussions over INGRID implementation carried up the idea that, if good functionality was envisioned, it was fundamental to provide a model with a consistent view of a UI along its design, development, and execution phases.

The object-oriented technology has proliferated in the UI area and has already shown to be helpful for UI developers in the design and construction process. Some systems reflect this approach and provide a flat object-oriented structure: MacApp [Sch86a], ET++ [Gam88a], or InterViews [Lin87a]. The main drawback that emerges from this structure is that it gives few guidelines for the design of a UI [Har89b].

Other systems, e.g. GWUIMS [Sib86a] or IMAGES, rely on classical UI models like Foley [Fol82a], Seeheim [Gre85a], or the reference model [Lan87a] and apply object-orientation specifically to identify UI components and their relationships. However, the strictly layered structure of the underlying UI models imposes communication overheads that disqualifies them to support fast interactions [Mye89a].

Other proposals address the two needs – design methodology and functional separation in a UI – by providing the same separation of the layered UI models, presentation, dialogue control and semantic support, still organizing this layers in a nested structure. MVC [Gol83a], PAC [Cou87a], and the 4D model are included in this last group.

2.1.1. Components

The 4D model defines four UI components, each one grouping objects with the same functionality. These components are:

Display – The graphical presentation (input and output) of the UI

It groups graphical objects: button, menu, scrollbar and composite Display objects. A composite Display object has a collection of objects also organised in accordance with the four elements of the model. This composition originates a recursive model [Cou89a].

Data – Application abstract information

This component provides common usage data structures (int, file, list, ...) with active values [Sze88a], i.e. values that automatically propagate their changes. Some Data objects also represent semantic actions of the application.

The primary reason for the existence of Data objects is to provide a clear interface between UI and computational parts while including semantic information on the UI side, in order to be able to support semantic feedback [Dan87a].

The existence of Data and Display objects enforces the separation between data and view. The support to different view types that use the same data increases the degree of flexibility of a UI.

Dialogue – The syntactic structure of the human-computer interaction

It groups dialogue control objects. Multiple implementations of dialogue control are feasible: dialogue languages, dialogue cells, etc.

Driver – Conversion of Data information in the format accepted by Display

Is composed by objects that perform data conversion (such as integer to a string).

Several Drivers using the same Data object allow multiple views of its contents.

Drivers introduce also one extra degree of flexibility by reducing the needs of new Data and Display objects. As an example, a "drawing" Display may be driven to represent a table, replacing the need for a "table" Display. In this example, one Driver should have to be coded instead of a Display, a major difference since the Display would be much more hard to code.

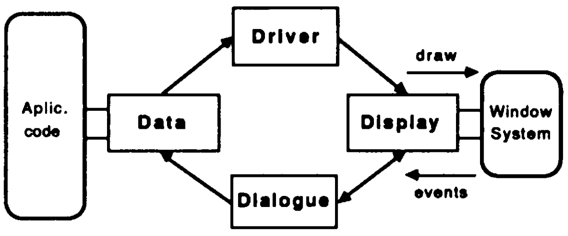


Figure 1: *The 4D model*

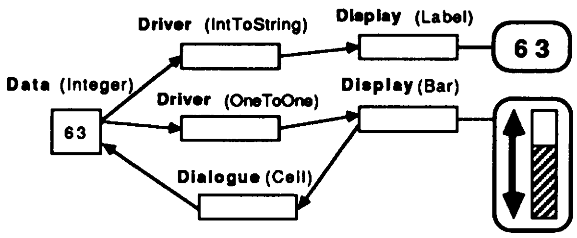


Figure 2: *An example*

The names of the four elements, Display, Data, Dialogue, and Driver led to the "4D" designation. Figure 1 illustrates the 4D model.

2.1.2. Links

Communication between objects is modeled in 4D through the concept of link. A link can be viewed as a permanent communication channel. One important difference between using links or method invocations is that an object, when sending a message, does not require information about the receiving objects, but only the outgoing links (the same difference of having a permanent communication channel or a datagram communication facility). This increases the degree of flexibility and reusability by providing well defined and autonomous behaviour for objects.

Components and links define how user events are handled, how feedback is provided, and which actions are executed. Figure 2 shows an example of this conduct. Events associated with the scrollbar are delivered to the Bar Display. The Bar notices mouse selections on the bar position and sends a message to the Dialogue with the new position. The Dialogue, in turn, is programmed to invoke the Integer Data object to update its value. Since Data objects have active values they invoke automatically linked Drivers to report changes in their values. Finally, Drivers convert the new value and update both the Label and the Bar.

3. INGRID

INGRID is designed according to the following main objectives:

Interactive Construction of the Interface

The process of creating an interface includes selection of the appropriate UI entities (category, component, class), its instantiation, parameterisation of its attributes (e.g. colour, text) and definition of their relations with other objects (links).

Although these operations can be available through an underlying language description – general purpose or UI specific – we believe that they should be executed interactively. For each selected action the UI programmer should be immediately aware of the change that was performed, either on the interface aspect or on its behaviour. Naturally, INGRID should also provide mechanisms to verify, and validate that behaviour, enforcing the incremental process of UI development.

Direct Manipulation of Entities

Experience has shown that direct manipulation of UI entities is better accepted by developers and drastically reduces the construction effort.

If direct manipulation is a good approach to some problems raised in UI construction, namely when the displayable part of the interface is being handled, it is sometimes difficult to apply to generic purpose customisation of objects, for example the attributes of an abstract data type. In these cases, solutions must be adopted that keep in mind the graphical aspect of INGRID. A hierarchical chain of menus, providing adequate guidance mechanisms, should be preferred to the direct coding approach.

Model enforcement

The importance of a good model for UIs has been stressed in the last section. INGRID adopts the proposed 4D model to describe the interface it presents to the designer, and enforces it through all the construction process of new interfaces.

Multi-thread Dialogue

Multi-thread dialogues, understood as the ability for the user to interrupt an action, execute another and return to the first, has been used successfully in the process of human-computer interaction. INGRID should offer it to the UI designer.

Binding to the Computational Component

INGRID is responsible for the creation of the interface component of the application. However, mechanisms must be provided to bind it to the computational part. Those mechanisms should emphasise the separation between application components, enforcing the fundamental concept of dialogue independence. They should also offer a flexible front-end that supports connections to multiple programming languages without requiring language specific knowledge from the UI programmer. Besides C++ and C, bindings to the computational part should also be available for languages like Pascal.

Help and guidance

The flexibility provided by interactive programming, with graphical direct manipulation and multi-thread capabilities, may present to the designer a very large set of options for which decisions are required. INGRID should provide guidance capabilities through the construction process, either by pointing the most adequate solution, or defining a structured approach to the interface organisation coherent with the model. On-line help facilities fall in the scope of these needs.

3.1. INGRID Components

INGRID has five main components which closely follow the adopted model (see figure 3):

- **Display sub-editor** – This editor handles objects with display properties, like position, size, text and bitmaps for which the direct manipulation paradigm seems most appropriate. The editor is divided in three main areas: *palette*, *canvas*, and *attribute-programmer*.

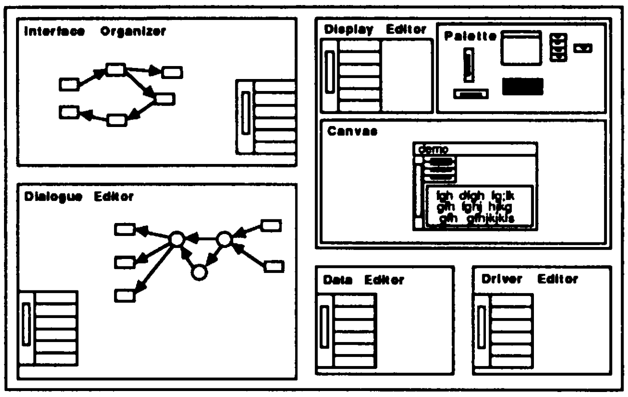


Figure 3: *INGRID components*

The palette is an area where 4D Display classes are represented and can be picked to create the UI. Picking such objects is equivalent to create a new instance that can be positioned in the working canvas. Several palettes can be available representing categories of related Display classes.

The canvas is the board where the programmer defines the UI external aspect. Multiple canvases can be used to give the notion of depth in the UI behaviour.

The attribute area is a general purpose programmer composed by a set of menus that allow the definition of attributes that cannot be customised within the canvases (e.g. fonts and colours).

- **Data sub-editor** – Data objects represent abstract structures not visible to the programmer. For these, the Data sub-editor provides an identification mechanism, which associates a name to each object. Later, these objects can be selected from a menu of created instances and parameterised through an attribute-programmer similar to the one available in the Display sub-editor. Instantiation can be performed from a collection of menus that refer to the 4D toolkit Data classes.
- **Driver sub-editor** – The nature of driver objects makes difficult the adoption of a direct manipulation paradigm for their parameterisation. In this case are also available menus for instantiation of Driver classes, attribute-programmers and an identification service similar to the one of Data objects. Composition of cascade Driver objects can be edited with a specific graph oriented editor.
- **Dialogue sub-editor** – The dialogue sub-editor depends on the dialogue control model that is made available by the underlying toolkit. Our approach defines an object-oriented event driven dialogue for which an event or a conjunction of events arriving to an object may trigger a pre-defined action. This kind of dialogue can be represented as a directed graph, suitable for interactive programming using direct manipulation techniques. The dialogue sub-editor, besides the graph editor, provides an attribute-programmer and an action-programmer which is used to associate tokens and arguments to graph arrows.
- **Interface organiser** – While the above sub-editors provide the definition of each component of the 4D model, the interface organiser allows global access to the interface entities. It is the front-end of INGRID, and will allow the establishment of links between the model components, the composition of objects into composite Displays, access to a generic attribute-programmer and global functions for store/retrieve, testing, etc.
- **Application connector** – The application connector is a specialised attribute-programmer that provides transparent access to the computational component of the application. That component is viewed as a set of Data objects, that defines the necessary translations between interface requests and calls to the computational component. The connector also provides a name service that can be used by the computational component to invoke interface entities.

3.2. Underlying structure

INGRID is build according to the layers shown in figure 4. The background layer of INGRID is the ICE run-time support that provides the interpretation capabilities needed to achieve a fast interactive design of the UI.

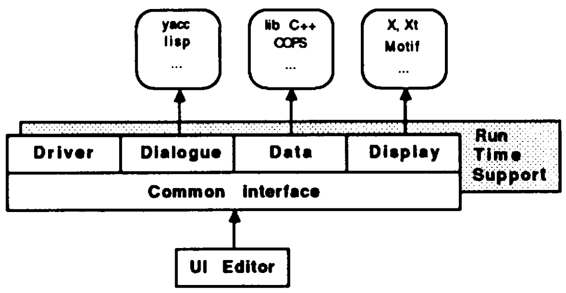


Figure 4: *INGRID structure*

The next layer, is composed by a set of toolkits which follows the 4D model concepts. Those toolkits provide a library of objects which implement the several components of the interface. When the UI designer picks an object from the Display editor palette, it actually instantiates, through the ICE support, an object belonging to a class available in one of the toolkits.

4. ICE – Support for Interactive Programming

ICE has been developed as a run-time support to provide interpretation features within C++ [Str86a]. Essentially, it makes available mechanisms for:

- **Type information** with type identification, conformance and conversion testing, and knowledge over instance data and member functions.
- **Communication by message** based on the available type information, provides a standardised paradigm of communication between objects. Dynamic binding is an immediate consequence of this communication paradigm.
- **Object storage and retrieval** again supported by the type interface description, enables transparent storage/retrieval facilities.
- **Object identification** enables the assignment of human readable names to objects, which enforces a coherent mechanism of interaction between the user and the programming entities. Being a global service it also provides the functionality of a run-time instance management table.
- **Interface uniformity** offers an abstract handling of objects, essential to ease the development of flexible programming environments, i.e. if all objects are accessible through a common interface, other types can be easily integrated without code modification.
- **Common use facilities** like error handling, debugging, ...

4.1. Implementation

The ICE functionality is implemented by a library of C++ classes, which can be classified under three main categories: type classes, name service, and common interfacing.

Type classes: The basic concept of ICE is the notion of *type-object*. It is an object that fully describes a C++ type. A type-object can be viewed as an extension of the Smalltalk's *class-object*, to all C++ types: classes, basic types (char, int, float, ...), pointer types and function types. The introduction of this broader notion, enables an uniform, coherent, access to the heterogeneity of C++ types. Type-related mechanisms like storage/retrieval and communication by message are available both to user defined classes, and C++ basic types.

The type classes closely follow the C++ type model:

- **IType** is an abstract class that defines the generic interface of type-objects. It declares methods for type checking, member function execution through message passing, instance creation, storage, and retrieval.

- **IBasicType** implements the interface for C++ basic types (e.g. operators +, –, ...).
- **IFuncType** provides type checking capabilities for functions.
- **IPointer** defines the generic behaviour of pointers.
- **IClass** provides the functionality associated with class and struct types: defines member function invocation by message with type checking, member overloading, and default arguments; considers inheritance both in conformance tests and message sending; uses constructors for object instantiation; uses instance data information for automatic storage and retrieval facilities; provides access to members according to their protection attribute (public, protected or private).

To have ICE complete functionality, an user-defined class must have at run-time its type-object. Such an object must contain very exhaustive information for which even the availability of suitable macros (as in OOPS and ET++) require a large effort from the class programmer.

ICE includes a parser for C++ definitions, that automatically generates code for the corresponding type-object.

Name Service: The name service is an instance of INameService class which provides a dictionary for translation of user readable names into object pointers and vice-versa. INGRID heavily uses this service to perform actions on objects specified by name.

Common Interface: ICE offers two different approaches to achieve common interfacing to objects. One provides a complete integration (similar to the one used in Smalltalk [Gol83b], OOPS [Gor87a] and ET++ [Gam88a]), by specifying a common base for all classes, named IObject. The other allows the integration of types that do not derive from that class. It is accomplished by means of an encapsulating object that enables the access to the corresponding type-object and offers an interface similar to IObject.

5. The 4D Toolkit

The 4D toolkit provides a set of classes (see figure 5) upon which INGRID creates and customises a UI. The toolkit must be rich enough in order to satisfy most of the needs of INGRID as well as developers and users.

ICE Support		IObject —— IClass		
4D		DObj		
Model	DataObj	DriverObj	DisplayObj	DialogueObj
4D	Int	IntToString	DisplayXt	Fork
Toolkit	Float	FloatToString	DisplayRoot	C-Dialog
	String	FileToString	Label	Cell
	File	OneToOne	Command	...
	Proxy	...	Box	
	...		...	

Figure 5: *Toolkit Classes*

With the purpose to support the 4D model and its components and links, all objects have a base class, DObj, that defines a common behaviour to establish, verify and maintain lists of links, using links to send and receive messages. Links rely on the ICE message passing mechanism provided by the IObject base class.

Four classes are derived from DObj: DataObj, DriverObj, DisplayObj and DialogueObj. An object belongs to a specific component if its class inherits from one of these.

Display objects have access to the Window System. The consensus around the X Window System [Get88a] made available several graphical toolkits (Xt [McC88a], Andrew [Neu88a], InterViews). The current implementation of the 4D toolkit uses Xt (and several widget sets: Athena, Motif, Open Look). This provides 4D with features like system-independence, portability, network transparency, multiple options of functionality and "look and feel". Furthermore, and since these toolkits are often difficult to use, an independent high-level access to its entities is available.

One class, DisplayXt, defines the interface to the Xtoolkit intrinsics. Another class, DisplayRoot, is the root class in the display hierarchy. The other Display classes bind to the specified widget sets (Label, Command, ...) and are derived from DisplayXt. Tools are available to parse widget declaration files and automatically produce these classes.

A specialised Data class, Proxy, is dedicated to interface with the computational part. A Proxy defines a set of semantic actions of an application. This object uses the ICE run-time support to dynamically create its interface.

Essential for interactive construction of the UI is the capability of doing a snapshot of its run-time structure. This save/retrieve facility has two concerns, one related with the run-time structure within the 4D model, mainly the created objects and their links; the other related with graphical parameterisation of Display objects that escape from the modeled objects to the specific widgets. In the first case, the ICE run-time information is sufficient to produce a snapshot; in the second case, a database with graphical information must be generated, using directions from the Window System.

6. Conclusion

The final goal of INGRID is the creation of a flexible interactive environment for user interface creation, in the context of an object-oriented framework. The target applications are the small and medium sized applications typically found in the engineering and office environments. We believe that the requirements defined for INGRID provide a solid ground to achieve this goal:

- The interactive behaviour provides a definite advantage in the construction of medium scale user interfaces. Actually, automatic mechanisms such as a specification language may have to be considered in larger scale applications, that is however out of the scope of this work.
- The support for the integration of graphical toolkits provided by the 4D toolkit is essential. This integration assures conformance with "standard" behaviour and avoids the effort involved in the development of a graphical toolkit.

  The 4D toolkit has been used independently from the editor. Its use did not reduce significantly the amount of code but provided a homogeneous structure to the several components of the interface, added functionality, greater independence from the graphical toolkit and the advantages of the C++ language.

  The issue of toolkit independence needs further validation by including access to Andrew or InterViews. This integration will provide a clearer definition of an adequate interface for the Display components and help to filter out any bias introduced by Xt.
- The development of a run-time support system in C++ provides a fairly open platform for integration of other software systems, either the graphical toolkits mentioned above or other available libraries.

  The run-time support system has been fully tested and approved by the implementation of INGRID itself. As a general purpose interactive/interpreted environment, its use can be considered in applications that require these characteristics and usually rely on object-oriented languages or environments such as Clos or Smalltalk.

The INGRID components have reached a stage of reasonable maturity and a first interface for INGRID has been built. INGRID will now evolve according to user requirements. From our viewpoint, one of the advantages of an interactive tool is the capability to reach a larger number of users, at least for experimentation purposes. This is definitely the major drawback of a linguistic tool.

We expect that user demands will concentrate on current limitations, namely higher direct manipulation capabilities, performance and compatibility and integration with other widget sets/toolkits.

References

[SOM85a] "Secure Open Multimedia Integrated Workstation," SOMIW Esprit 367 - Technical Annex, 1985.

[Cou87a] J Coutaz, "The Construction of User Interfaces and the Object Paradigm," in *ECOOP'87, European Conf. on Object-Oriented Programming*, pp. 121-130, Paris, June 1987.

[Cou89a] J Coutaz, "Architecture Models for Interactive Software," in *ECOOP'89, Proc. of the Third European Conf. on Object-Oriented Programming*, pp. 383-399, Nottingham, July 1989.

[Dan87a] J R Dance, "The Run-time Structure of UIMS-Supported Applications," *Computer Graphics*, pp. 97-101, April 1987.

[Fol82a] J Foley and A Van Dam, *Fundamentals of Interactive Computer Graphics*, pp. 220-222, Addison-Wesley, 1982.

[Gam88a] E Gamma, A Weinand, and R Marty, "ET++ - An Object-Oriented Application Framework in C++," in *Proc. of the Autumn 1988 EUUG Conf.*, pp. 159-173, Portugal, October 1988.

[Get88a] J Gettys, R Scheifler, and R Newman, *Xlib - C Language X Interface, X Window System X11R3*, 1988.

[Gol83a] A Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1983.

[Gol83b] A Goldberg and D Robson, *Smalltalk-80: The Language and its implementation*, Addison-Wesley, 1983.

[Gor87a] K E Gorlen, "An Object-Oriented Class Library for C++ Programs," in *Proc. of the USENIX C++ Workshop*, New Mexico, November 1987.

[Gre85a] M Green, "Report on Dialogue Specification Tools," in *User Interface Management Systems*, ed. Gunter E Pfaff, pp. 9-20, Springer Verlag, 1985.

[Har89a] H R Hartson and D Hix, "Human-Computer Interface Development: Concepts and Systems for its Management," *ACM Computing Surveys*, vol. 21, no. 1, March 1989.

[Har89b] R Hartson, "User-Interface Management Control and Communication," *IEEE Software*, pp. 62-70, January 1989.

[Lan87a] K Lantz, "Reference Models, Window Systems and Concurrency," *Computer Graphics*, pp. 87-97, April 1987.

[Lin87a] M A Linton, P R Calder, and J M Vlissides, "The Design and Implementation of InterViews," in *Proc. of the USENIX C++ Workshop*, New Mexico, November 1987.

[Mar88a] J A Marques, L P Simoes, and N Guimaraes, "A Uims and Integrated Environment for the Somi Workstation," in *Proc. of the ESPRIT'88 Conf.*, pp. 1001-1019, Brussels, November 1988.

[Mar89a] J A Marques and P Guedes, "Extending the Operating System to Support an Object Oriented environment," in *OOPSLA'89 Conf. Proc.*, pp. 113-122, New Orleans, October 1989.

[McC88a] J McCormack, P Asente, and R Swick, *Xtoolkit Intrinsics - C Language Interface, X Window System X11R3*, 1988.

[Mye89a] B A Myers, "User-Interface Tools: Introduction and Survey," *IEEE Software*, pp. 15-23, January 1989.

[Neu88a] C Neuwirth and A Ogura, *The Andrew System Programmer's Guide to the Andrew Toolkit*, ITC - Carnegie-Mellon University, January 1988.

[Sch86a] K J Schmucker, "MacApp: An Application Framework," *Byte*, vol. 11, no. 8, pp. 189-193, August 1986.

[Sib86a] J L Sibert, W D Hurley, and T W Bleser, "An Object-Oriented User-Interface Management System," *Computer Graphics*, pp. 259-268, August 1986.

[Sim88a] L Simoes, "IMAGES - An approach to an Object Oriented UIMS," in *Proc. of the Autumn 1988 EUUG Conf.*, pp. 143-157, Portugal, October 1988.

[Sim87a] L P Simoes and J A Marques, "IMAGES - An Object Oriented UIMS," in *Human-Computer Interaction - INTERACT'87*, pp. 751-756, North-Holland, 1987.

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

[Sze88a] P A Szekely and B A Myers, "A User Interface Toolkit Based on Graphical Objects and Constraints," in *OOPSLA'88 Conf. Proc.*, pp. 36-45, San Diego, September 1988.

The Design of a PC-based NFS Client from an MS-DOS Programmer's Perspective

Ian Chapman

Siemens plc
Systems Development Group
65-73 Crockhamwell Road
Woodley, Reading
Berkshire, RG5 3JP
England
+44 734 443 042
ian@siesoft.co.uk

ABSTRACT

Siemens Distributed File System (DFS) is a client-only NFS implementation for IBM PCs and compatibles. The project was undertaken because a similar product did not exist to run with Excelan Ethernet hardware. The Project Team was composed primarily of MS-DOS programmers, which led to a number of interesting differences in the design of DFS compared to the other PC-based NFS clients on the market. This paper gives an overview of Siemens DFS, highlighting these differences.

1. Introduction

Siemens licensed NFS from Sun several years ago, and now uses it extensively in conjunction with its Sinix (UNIX) and BS2000 (proprietary) operating systems.

Perhaps, inevitably, the need arose to mount UNIX filesystems from a PC running MS-DOS and to allow a PC to access remote print services. The obvious solution to this requirement was Sun's PC-NFS product. However, Siemens has traditionally used Excelan ethernet hardware for all its microcomputer range, (both Multibus and AT-bus architectures) and has a large installed customer base of this hardware running a number of software products which rely on the advanced features of the Excelan hardware. The Excelan EXOS 205 card used on the PC has an Intel 80186 processor and up to 512 Kbytes of RAM on board. The TCP/IP stack is downloaded onto the board, hence freeing a large amount of PC memory. Unfortunately PC-NFS will only run with "dumb" ethernet adapters and will not run in conjunction with "intelligent" Excelan hardware. Licensing the source of PC-NFS was investigated, but proved impossible due to licensing problems with the Locus redirector used in it. The market was large enough to justify the development of a new product similar to PC-NFS but running exclusively with the Excelan hardware. The project was called Distributed File System (DFS).

The interesting point of this development was that the team was composed mainly of MS-DOS programmers with, initially at least, a scant knowledge of UNIX. This may be contrasted with the team which developed PC-NFS, who mainly came from a UNIX background. This led to a number of interesting differences in the design approach compared with PC-NFS. This paper describes the main differences, and the design decisions taken which led to them.

2. DFS Overview

There were two distinct parts to the development effort of DFS. The fundamental part was the kernel, essentially an MS-DOS redirector. This is responsible for intercepting various interrupts and deciding if a system call is intended for a local drive or a remote network drive. In the former case the call is passed directly to MS-DOS. In the latter case the call is turned into an NFS RPC call and passed across the ethernet to the server. The reply data is turned into an MS-DOS compatible format and returned to the calling function. If there is an error the appropriate error code is returned, or a critical error is emulated. The kernel is also responsible for handling remote print requests.

The rest of the project was concerned with developing the associated utilities. There are two classes of utilities. Some are required in order to use the system, for example to log on to the server and mount drives. Others are provided for convenience, for example to convert the format of text files, or to allow a remote shell to be run.

The kernel was written by two experienced MS-DOS programmers, and the utilities were written by two somewhat less experienced programmers from a UNIX applications background.

2.1. Design Requirements

At the commencement of the project a number of requirements were immediately obvious. The major design requirements are presented below.

- Match or improve size and performance of PC-NFS.

  Since the project was aimed at producing an alternative to PC-NFS the memory requirement and performance must at least equal that of PC-NFS.

- Have an optional dialog-box style interface for the utilities.

  It was decided from the outset to have an optional windows-style interface for the utilities, complete with on-line help. This was particularly aimed at the naive user.

- Fully internationalise all text.

  Since DFS was intended primarily for the European market it was designed to allow for different language variants. MS-DOS stores a code internally which is derived from the *country* parameter in the *config.sys* file. This value is used to determine the appropriate language automatically.

- Allow the redirector to be unloaded from memory and subsequently reloaded.

  In order to allow large programs to run, the redirector was written as a terminate and stay resident (TSR) program rather than a device driver to allow unloading from memory to occur at any time. This would then render any network drives inactive.

- Allow existing physical drives to be overlaid by virtual drives.

  DFS allows a drive to be mounted on top of an existing MS-DOS drive. There is a large market for diskless PCs, and to address this the ability to overlay existing physical drives with a virtual drive was deemed important.

- Emulate MS-DOS 3.10 advisory file locking.

  There has been provision in MS-DOS for file and record locking since version 3.10. This is emulated by DFS. The Siemens lock manager on Sinix is slightly different to the Sun lock manager, and so compatibility was maintained with both.

  DFS implements the file locking in a manner consistent with MS-DOS and does not allow an application to lock a region more than once. The UNIX lock managers allow a region to be locked any number of times.

- Allow LPT1-3 to be redirected over the network.

  Use of print services on a server is obviously very important. Conceptually there is little difference between mounting a drive and redirecting a printer and, in fact, when a printer is redirected PC-NFS uses a drive letter for the spool directory. It was decided to keep printer redirection totally separate from file system redirection in DFS since this would be much more consistent with MS-DOS practice.

- Make kernel fully configurable.

  As many kernel-related parameters as possible were made user configurable, via a text configuration file. This was in order to tailor the system to a user's needs and minimise the storage requirements. It results in a saving of up to about 16 Kbytes of memory, which is significant in the MS-DOS environment. The following parameters are configurable:

- Number of drives, servers, printers
- Path length
- Length of computer name, mount path and unmapped file name
- Number of open files, mapped files and locks
- Default umask
- Retries and timeout
- Printer option length
- I/O buffer size
- Default printer options
- Default authentication server

• Retain compatibility with Sun *pcnfsd.*

Sun developed the daemon *pcnfsd* to facilitate authentication and remote printing. DFS uses the same facilities.

• Do not allow the user id *nobody.*

PC-NFS defaults to the user *nobody* (uid and gid both -2) if there is no authentication carried out. It was decided not to have this feature, since it compromises security. Therefore no DFS actions are possible until the log on utility has been successfully executed.

3. Implementation

Apart from the MS-DOS redirector, the kernel also has NFS, RPC, XDR and sockets support. Many of the utilities require similar support, and additionally the mount protocol is used by several utilities. It would have been possible to implement this functionality from the specifications, but much of it was available from third party sources. Just before the commencement of the project the Sun Public Domain release 3.9 of RPC/XDR became available, and so it was decided to port this to MS-DOS. Excelan produce a BSD 4.1c compatible socket library which communicates with their hardware. The socket calls in the RPC/XDR code were modified from BSD 4.3 to BSD 4.1c to align with this, and a library was produced along the lines of the PC-NFS Toolkit to be used with the utilities. This port was fairly simple, although some problems were encountered moving code intended for a 32 bit machine to a 16 bit machine.

A major effort was involved in reverse engineering MS-DOS to discover the details of the various undocumented system calls, and also to emulate some of the more esoteric "features" of MS-DOS. The redirector was tested extensively with the major MS-DOS applications, with particular emphasis on the most badly behaved.

The redirector was written from scratch, as far as possible in C, and it ended up about 70% C, 30% assembler.

The RPC/XDR support in the kernel comprises of a heavily modified version of the utilities library described above. Very strict conditions are imposed on a TSR program; for example, it may not allocate memory and care must be taken when calling MS-DOS. Also, only a small part of the functionality was required within the kernel, and since memory was at a premium unused parts were removed. It remained entirely written in C.

The Sun *rpcgen* utility was used to generate the NFS and MOUNT protocol code. This was very straightforward, although two errors were found in the mount protocol as supplied.

In order to allow authentication and printing on the server, Sun developed a daemon known as *pcnfsd* providing these services. The source for it was released into the public domain. This was modified by Siemens and renamed *rpc.dfsd* and is now shipped with all Sinix systems. The printer support has been enhanced, but it remains upwardly compatible with *pcnfsd*, and DFS will work with *pcnfsd*, albeit with slightly less functionality.

3.1. Design Compromises

Although MS-DOS and UNIX have some quite marked similarities, for example in their directory structure, there are clearly a large number of fundamental differences. This led to compromises having to be made in the design of DFS. Where a design compromise had to be made the philosophy was adopted that MS-DOS conventions should be followed as closely as possible. This was to ensure that the system behaved as an MS-DOS user would expect rather then as a UNIX user would necessarily expect. This may be contrasted with PC-NFS which, in general, follows UNIX (SunOS).

The most notable of these compromises are described below.

- DFS only allows file systems to be mounted which have both read and search (execute) permissions.

  PC-NFS allows a file system to be mounted which just has search (execute) permission or read permission, or both. A *dir* (MS-DOS equivalent of UNIX *ls*) command will list all subdirectories of the mount point. However, if a directory has only read permission *cd* to it will not work. If a directory has only search permission *cd* will work, but *dir* will not list any files. In both cases new files may not be created.

  To avoid this problem DFS only allows a mount if the user has both read and search permissions. Also, any directories without both of these attributes will be hidden on a *dir* listing.

- If no filesystem is mounted on a drive it is inactive.

  PC-NFS acts as if the *lastdrive* parameter in *config.sys* is set to *V* (i.e. 22 drives are available) to determine the maximum number of virtual drives. If an allowed virtual drive is not mounted onto an NFS filesystem, a user may still change to the drive. However, when an attempt is made to access the drive a critical error will occur. This behaviour is analogous to attempting to access a floppy disk with the drive door open. There is a knock-on effect in the MS-Windows environment, where all 22 drives are displayed on the MS-DOS Executive screen.

  DFS behaves in a manner consistent with MS-DOS. If nothing has been mounted the virtual drive is inactive, and an "Invalid drive specification" message is given if an attempt is made to switch to it. This gives better results with MS-Windows where only the active drives are listed at the top of the MS-DOS Executive screen.

- Treat mount and unmount as the MS-DOS *subst* command.

  The MS-DOS command *subst* allows a virtual drive to be created (or deleted when the */d* option is used) by associating a pathname with a drive letter.

  Conceptually, mounting a virtual drive is the same as executing the MS-DOS *subst* command, and unmounting is the same as *subst /d.*

  PC-NFS allows the user to unmount the current drive, resulting in the drive no longer being valid.

  DFS does not allow a drive to be unmounted if it is the current drive or if it has open files on it. This is consistent with the *subst /d* command.

- DFS allows an application to lock a region of a file once only.

  PC-NFS allows an application to lock a region of a file more than once and allows part of a locked region to be unlocked. This is consistent with the SunOS network lock manager.

  DFS allows an application to lock a region of a file once only, and it must unlock the whole of the locked region not just a part of it. This is consistent with the file and record locking support in MS-DOS 3.10 and above.

- DFS will not allow a file to be removed with *rmdir.*

  NFS allows a file to be removed with the *rmdir* RPC call, which is arguably a bug in the present NFS server implementation. PC-NFS also allows this to happen, although it is inconsistent with MS-DOS. This is potentially very dangerous, since a non-empty directory can be deleted, effectively deleting all files and subdirectories below it. DFS traps this and disables it.

- DFS supports open/close of a network printer.

  There is a potential problem when printing is redirected. The print spooler does not always know when the spool file is complete, and so should be sent to the printer.

  Many applications allow output to be saved to a file rather than sent to a printer. If the output is saved to a file with the same name as the printer device, for example *lpt2*, printing will be started whenever the application closes the file. DFS behaves in this way, exactly as MS-DOS does.

  PC-NFS does not allow this. It treats the file *lpt2* exactly the same as the device *lpt2:*. Printing is only started when an application is left, after a timeout or when a hot key is pressed. DFS behaves in this way only if output is being sent to a device.

3.2. Filename Mapping

One of the major differences between MS-DOS and UNIX is in the area of filenaming.

MS-DOS has a very rigid format of between 1 and 8 characters for the filename, optionally followed by a "." and an extension of 3 or fewer characters. It is case insensitive and many non-alphanumeric characters are illegal in filenames, for example "/" and ";".

UNIX allows a very free format for filenames, with all characters except the "/" pathname separator being legal, and a maximum length of 255 characters in some flavours of UNIX.

There are two ways to resolve this problem. Either any UNIX filenames which are not also legal in MS-DOS could be ignored, or they could be mapped onto unique but legal MS-DOS filenames. Clearly the latter case is to be preferred in order to avoid compromising the usefulness of the system.

The approach used was to retain as much as possible of the UNIX filename, and use a tilde (˜) followed by a legal MS-DOS filename character to complete the mapped filename. If there is more than one "." in the filename, all but the last are stripped out. Where possible the first three characters and the last three characters before the remaining "." are preserved, after removing any illegal characters. The mapping characters are then inserted in the middle of the filename, unless there are less than six characters, when the mapping is put at the end. Finally, if the file has an extension, the first three characters after the "." are preserved as the extension, or less of the extension is not three characters. All UNIX lowercase characters are converted to uppercase before any mapping is carried out, with the implication that a UNIX filename with any uppercase characters will always be mapped.

The number of filenames which may be mapped is limited to 52, which is the number of legal filename characters in MS-DOS. A value lower than this may be specified when DFS is installed to avoid the data storage space required. Mapped filenames are held in a circular buffer, and so once more than the maximum number of files have been mapped, the mapping character is reused.

This algorithm is similar in spirit but very different in detail to that employed by PC-NFS. In general it makes the mapped filenames much easier to identify.

Some examples are given below to illustrate the above points, with the UNIX filename, the DFS mapped name and the PC-NFS mapped name:

UNIX	DFS	PC-NFS
`abc.123`	`ABC.123`	`ABC.123`
`ABC.123`	`ABC~A.123`	`ABC~~~AA.123`
`alongname`	`ALO~BAME`	`ALONG~BA`
`a.b.c.123`	`ABC~C.123`	`A~B~C~CA.123`
`abcd.1234`	`ABCD~D.123`	`ABCD~~DA`

A utility exists to allow the mapping to be resolved easily.

4. Conclusion

The DFS project has resulted in a high quality product. It contains approximately 90,000 lines of code, and the total bug count was under 100, of which only 7 were found after internal testing was completed. The total development time was around 2 elapsed years, or 8 man-years, including manual production.

All the design goals were met, although on fast 80386-based PCs the throughput is lower than PC-NFS since the Excelan ethernet hardware produces a performance bottleneck. Performance is comparable to a slow hard disk. Maximum throughput is 1200 Kbit/sec for reading, and 428 Kbit/sec for writing.

The memory requirement of DFS is between approximately 64-80 Kbytes, depending on the configuration chosen, which is better than was envisaged.

DFS has been tested and verified with various NFS servers, and in particular with Siemens, Sun and Apollo UNIX machines. It has also been tested against a VAX-based VMS NFS implementation originating from Excelan.

DFS is certainly not without its limitations, and two in particular may be highlighted. Currently DFS utilities do not include Yellow Page support for hostnames, and a local *hosts* file must be maintained. A future release is likely to include Yellow Page support so that the network data base may be used, easing administration problems. Also, being tied to Excelan's ethernet hardware is not an ideal situation, and the possibility of changing the interface to a packet level driver to allow more hardware to be supported is being investigated.

Archiving of Documents on WORM Disks – the NFS Approach

Helmut Kalb
Snoopy Schmitz

iXOS Software GmbH
Bretonischer Ring 12
8011 Grasbrunn/Munich
West Germany

ABSTRACT

We would like to present some work we have done in a contract for SIEMENS, Germany. The project was to allow UNIX users to store NCI documents on optical disks. Due to legal reasons WORM disks were chosen because the immutability of the medium makes them ideal media for archiving. Our solution is a client-server implementation that has the following features

On the server side:

- WORM-disks, jukeboxes and printers are connected via SCSI-2 interfaces
- UNIX and NFS transparent file system
- no changes in kernel
- autonomous information on the WORM-disks
- unique identifiers for all objects
- transactions
- additional SQL interface to the archive
- administration of on-line and off-line archives

and on the client side:

- Fax group IV compression for NCI documents
- display via X-Windows and MOTIF
- special client programs for mass scanning support
- several different client programs for the archive administration

1. The Problem

Archives are problematic beasts to map into the philosophy of rapidly changing UNIX file systems. They tend to grow and never change (except by further growth). The documents have to be stored in an NCI format to allow the storage of handwritten materials so as to allow the comparison of signatures etc.

The archival medium has to offer the same level of performance as e.g. micro-fiches in terms of immutability, longevity and image quality. One has to bear in mind that a retrieved document has to be a valid legal document.

A user-specified form of document retrieval should be possible. The queries should not be fixed in any form.

Due to the longevity of archives the criteria according to which documents are sorted or grouped may change several times over the lifetime of the archive.

Archives tend to grow and will therefore reach a point where it is impossible to keep all documents on-line, however the retrieval function should be flexible enough to allow the administration etc. of off-line documents.

Several users have to be able to access the archives simultaneously.

Sometimes the archive media is used to distribute the information to several different geographic locations (CD-ROMs are often used like this).

At the start of the project (in May 1988) we looked at several already existing solutions. However most fell down on the retrieval problem or they were MS-DOS based solutions which would not integrate noiselessly into a UNIX environment. We therefore decided to do it our way.

2. Basic Functionality

The heart of our solution is the WORM File System (WFS). Due to the WORM feature of the disks its not easy to mimick a UNIX file system. The UNIX transparency is handled by the illusion that the disks only have a decreasing capacity during their lifetime. If for example files get renamed this necessitates a control block on the WORM disk. The same holds for (logical) file deletion: a WORM disk is therefore considered full if there is just enough space left on the disk to write the control blocks for the deletion of all files on the disk (a volume is one side of the WORM disk – both sides can be used).

Another important feature is the NFS transpacency simply to allow easy integration into the UNIX world. Note also that the use of PC-NFS would also allow easy access via MS-DOS. The client machine mounts the WFS-Server which listens to the usual NFS requests. This server implements the WORM file system – it mimicks the UNIX semantics. It also administers the different volumes kept in single drives or jukeboxes.

Every file system needs i-nodes or similar structures to locate files and maintain consistent access. WFS i-nodes are written onto the disk and are also cahed on magneitc disk as relations in two database tables. We use a fully relational and distributed database (INFORMIX-NET in fact). C-ISAM would suffice for the needs of the WFS-Server however the use of a full SQL database allows for the required additional retrieval functions.

In case of a loss of the information of the database (due to disk or more commonly software failures) the whole database can be reconstructed with a special program called "import". Note that also the user defined tuples get recorded on the disk and are thus also reconstructable.

In order to identify WORM File Systems, directories and files each object is given a unique 64-bit identifier formed out of the date and some random components. This is unique for each server and sufficiently unique for a large number of servers.

The WFS Server is a collection of user processes. It is not implemented in kernel. This gain in modularity was worth the performance penalty. The redirection of the NFS-calls to the WFS server necessitated a small change in the mount daemon (/usr/etc/rpc.mountd). However it looks like these changes are in the process of migrating back into standard UNIX (Berkeley).

The WFS server supports single WORM drives and jukeboxes. Several of these can be supported (typically 4 Jukeboxes of 5 1/4'' disks and a total on-line capacity of 64 GB).

Note that most UNIX systems have a limit of about twenty mounted file systems. Thus it is clearly not possible to mount every volume of a jukebox (which means 40 volumes (or 20 disks)). Therefore we have implemented a scheme where an instance of the WFS server gets mounted.

Each volume that is available then exists as a subdirectory underneath the mountpoint, which is this process. This way we can support several disks and jukeboxes without ever exceeding the UNIX mount limit. Also the casual user of the *ls* command sees every available disk as a UNIX directory which is quite nice. After all these disks are available on or at least near-line. There is no reason they should be mounted specifically in order to be visible.

3. User Defined Tuples for Document Retrieval

The retrieval functionality can be supplied independently to the WFS but this does not guarantee the consistency between WORM objects and retrieval references. This is why we allow the user to extend the database used for the i-nodes of the WFS by new tables which contain the desired tuples. Such tables contain the user defined fields and the unique ID of the object referenced. Therefore consistency checks are very simple.

This concept allows two views of the objects on the WORM disk:

1 a set of file systems with UNIX semantics

2 a relational database with references to external objects (and SQL interface).

A write access into the database has to be announced to the WFS, in order for the tuples to also be recorded on the WORM disk. This was done with an interface on top of ESQL-C – the functionality is the same but the calls have slightly different names. The standard WFS functionality and the retrieval is handled by "vanilla" ESQL-C.

4. Administration of On-Line and Off-Line Archives

The WFS server handles all access to the files. Since the i-node information is kept in the database most NFS requests can be handled without the disk having to be available on-line (for example READ_DIR, GETATTR etc.). As long as nobody wants to read or write the disk (i.e. do any actual I/O) the disk can well be held off-line.

This means that the size of the database will soon be the limiting factor. Should this happen we recommend to remove the information of the off-line media except for a base minimum (i.e. the user defined tables). In case of a request the required volume can then be re-imported in a few minutes.

The "import" program not only imports whole disks but also allows incremental import to allow the quick update of the database in case of only small changes on the disk (for example if the disk was extended at a different site or if the database could be partially recovered). This greatly speeds the reconstruction of the database.

5. Use.of Caches and Several Server Processes

The WFS Server as well as the Informix database as well as any other server processes run and reside on the server machine.

The simultaneous access of several clients can thus become a bottleneck especially if the requests necessitate lots of disk exchanges in the jukebox. The client software therefore fires off read-ahead requests in order to cache several pages of a document for quicker browsing. The WFS server also has caches in memory.

In the case of writing into files, the throughput to the WORM disk is a lot less than reading (about half). Therefore the WFS server caches files on writing on magnetic disk. Additionally this allows files to be extensible and modifiable before they get committed to the WORM disk. NFS is a stateless protocol hence it is difficult for the WFS server to know when a file is closed because it then has to be written to the WORM disk.

A quirk of the UNIX system comes to the rescue – most programs (even tar, cp, cpio etc.) close a file and then cause an update of the files creation time – this causes an NFS SETATTR. This SETATTR is interpreted by the WFS server to close the file and flush the magnetic disk to the WORM disk. Needless to say the archive clients always ensure that they execute a SETATTR after they have finished the I/O into the file. However it is nice to know that most standard UNIX utilities do so as well and can thus freely use the WFS.

NFS requests are not handled sequentially by the WFS server but in parallel. Several processes share these tasks – one handles database requests and the others do the I/O (one process per jukebox or drive). These processes communicate via shared memory, message queues and synchronise via semaphores.

The main process is the RPC receiver. At startup it forks several sub processes: one for every single drive or jukebox. These processes handle all the I/O and the volume changing etc.

Another set of processes will handle all requests for the database. Note that such requests may come from the user (i.e. out of the kernel) or from one of the other processes which may need to enquire the WFS i-nodes.

The RPC receiver handles all incoming NFS calls and will keep a logbook. Some operations like disc exchanges in the jukebox may take several seconds (about 10). Therefore it is likely that the client will retransmit its request several times depending on its timeout interval. The receiver knows which requests are currently being processed and will ignore retries due to timeouts. Bear in mind that some operations may require a new import of an off-line volume – this may take up to ten minutes.

The statelessness of NFS is fairly nice but it may not be sensible for a user to have to wait that long especially since the statelessness of NFS does not allow a user to distinguish between the crash of the server and the importing of a disk.

6. Fax Group 4 Compression for NCI Documents

The huge capacity of the WORM disks makes them ideal for the storage of NCI documents. A scanned A4 page at 300 dpi takes about 1 MB. Compression by fax 4 reduces the disk use to less than 50 KB (typically a little less more like 30-40 KB). Documents get stored in compressed form and are transmitted over the net in such a fashion. The bitmaps get decompressed locally at the client machine. The idea of fax compression would also allow document output on (cheap) 400 dpi fax machines.

7. Display via X-Windows and Motif

The display of documents and all retrieval software was implemented using MOTIF. Work was begun using XUI and the port of MOTIF 1.O has only been finished recently.

8. Mass Scanning

Most organizations have large paper archives which have to be scanned. This demands a fast input channel into the archive system.

Documents get scanned by a double sided scanner on an MS-DOS based machine and get transmitted to the UNIX archive machine via PC-NFS.

Here the images are picked up by an iconisation daemon which first compreses every page according to Fax Group IV and also zooms the page down to produce a small icon of the page.

This icon is used by an interactive display program which allows the user to give attributes to the document (a document is any number of pages like a letter or a personnel file etc.). The order of icons presented is consistent with the order of the papers in the pile of scanned documents. Therefore the user who has to give the attributes (e.g. social security number) can easily relocate the paper original in the pile using the small icons and accurately define the attribute.

The above software then produces a command file in the directory of the document. The existance of this file wakes up another daemon – the archive daemon. This gleans the attributes from the commands file and checks if it can find a document in the database using the documents name. If it is found, the database is used to calculate a UNIX pathname to place the document onto the right volume into the right place (this is an extension operation).

If the document is not found it is considered to belong to a new collection of documents and a fresh disk has to be selected. The database contains a field containing the median document size for this installation (which gets regularly updated) and is used by the archive daemon to calculate the maximum allowed number of documents on a disk.

The archive daemon selects a disk which has the smallest delta between existing documents to this document limit. This guarantees that disks fill one after another and also ensures a certain locality of reference for new documents which minimises disk changes.

9. SCSI Device Support

The connection to the devices (scanner, WORM drives, jukebox, printers) is handled by SCSI-2. We have written other servers which handle CD-ROMs and rewritable optical disks on the SCSI-Bus. The target (SIEMENS) systems have a good and available SCSI-2 driver. Thus we had to make no kernel changes.

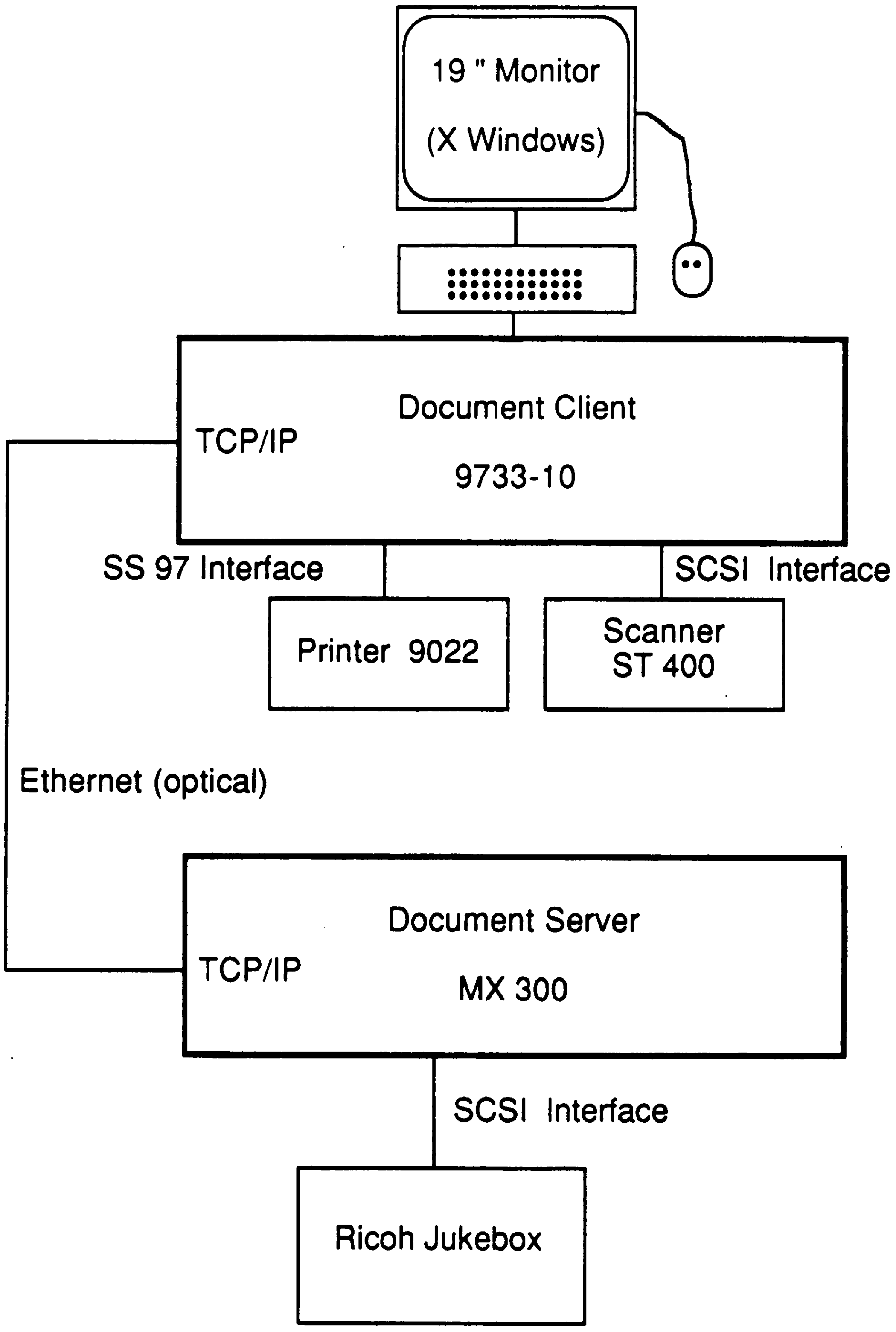
10. Configurations

Typical archive configurations contain the archive server as well as several scan and display work stations. They are all connected via TCP/IP and NFS over Ethernet media.

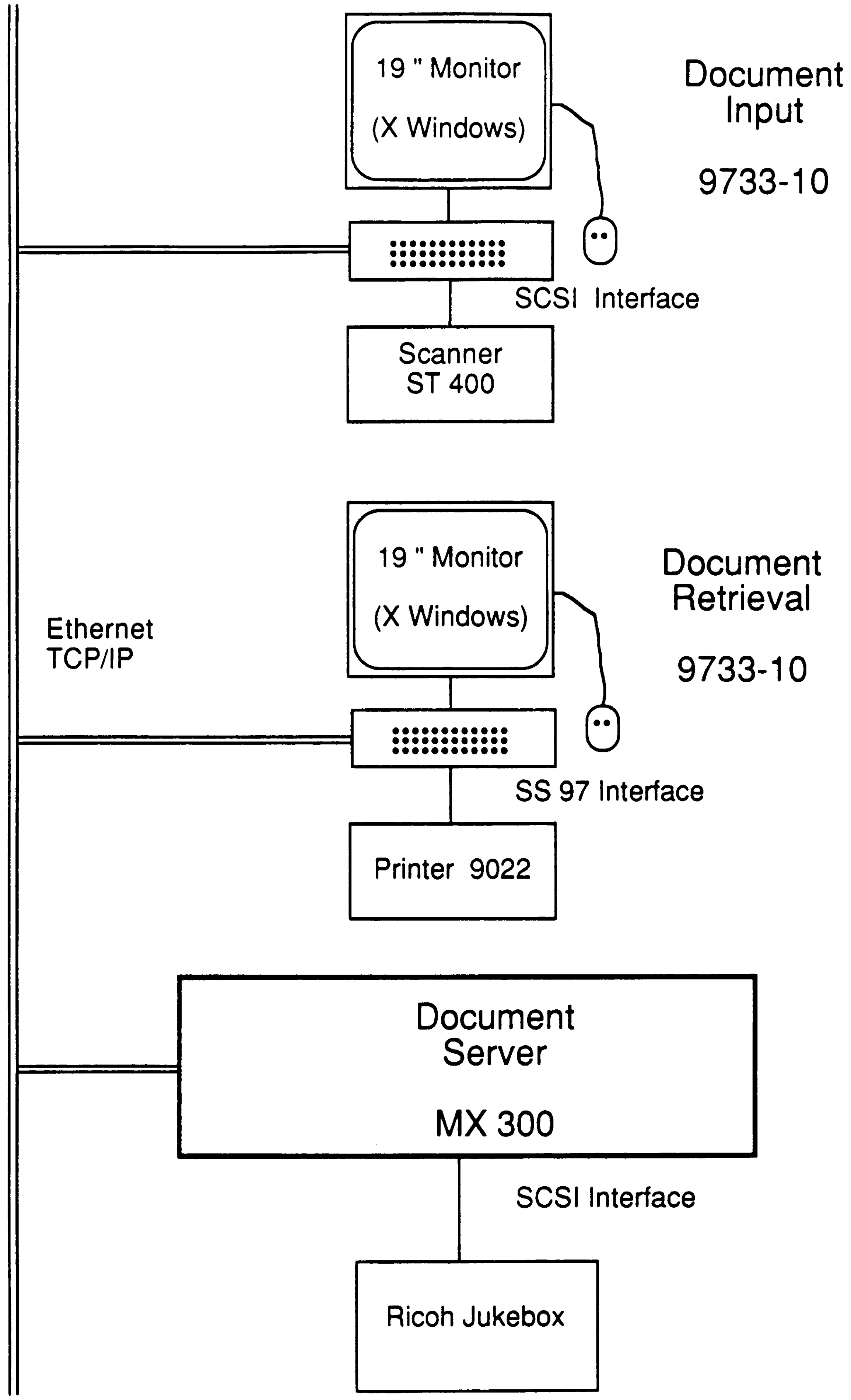
11. Further Work

The project is going to be installed at the end customer site for first tests while you are listening to this talk. However SIEMENS has decided to extend the archive systems functionality by features such as OCR to aid the attribution of documents and mixing of scanned images and ASCII documents. Other devices (i.e. different SCSI scanners) will also be supported in the future.

Example Configuration Labour Office



Configuration



Breaking Through the NFS Performance Barrier

Joseph Moran
Russel Sandberg
Don Coleman
Jonathan Kepecs
Bob Lyon
jkepecs@legato.com

Legato Systems, Inc.
260 Sheridan Ave.
Palo Alto, CA. 94306
U.S.A.

ABSTRACT

The Sun Network File System (NFS) has become popular because it allows users to transparently access files across heterogeneous networks. NFS supports a spectrum of network topologies, from small, simple and homogeneous, to large, complex, multi-vendor networks. Given the diversity and complexity of NFS environments, isolating performance problems can be difficult. This paper identifies common NFS performance problems, and recommends specific practical solutions. In particular, we evaluate the impact of reliable write caching on NFS performance.

Introduction

NFS [San85] performance problems can be broken down into three basic areas: client, network, and server bottlenecks. We will explore each of these areas and show why the server, and in particular, the server's I/O subsystem, is usually the primary cause of poor NFS performance. We demonstrate that reliable write caching can have a major impact on NFS performance improvement by eliminating slow, synchronous disk I/O.

Network Bottlenecks

Analysis of the network used to communicate between the client and server may turn up bottlenecks in addition to the inherent latency of the network. Three important factors to evaluate include network load, network topology, and packet loss.

Network Load

If the network is over-utilized, clients will experience longer delays waiting for a free slot to send requests in. However, it is a mistake to assume that an Ethernet can only handle a small fixed load because of increasing collision rates. Boggs et. al. [Bog88] have explored this issue in detail and demonstrate that in many cases, a heavily loaded Ethernet can deliver its full bandwidth. Even with 24 hosts continuously sending maximum-length packets, only about 3% of the delay in sending is due to collisions. The remainder of the delay results from the Ethernet's fairness policies which effectively cause hosts to "wait their turn" to send a request. Although the authors point out that average transmission delay increases linearly with the number of hosts and does not increase dramatically at a particular load threshold, it is still significant. With three hosts generating continuous maximum-length packets, average transmission delays of about 3-4ms are experienced. Since the average NFS response time under a low to medium load is around 30 milliseconds, it is apparent that a 10% (i.e., noticeable) delay would be experienced if three NFS clients or servers were continuously sending read or write data. Normal NFS clients do not put this much load on the network. To saturate an Ethernet, a client would have to continuously send around 800 1536-byte packets/sec. We have observed that average NFS request rates tend to be around 1.5 NFS ops/sec/client, with less than 4 packets/op. This is an average rate of less than 6 packets/second. In bursts, however, a single client can effectively saturate the Ethernet (e.g., by issuing a stream of read requests that are satisfied from server memory). In our experience, when the Ethernet is continuously loaded at a level of more than 40%, or when there are sustained bursts in excess of a 60% load, noticeable delays are seen. Thus, if you observe such Ethernet loads (Sun's *traffic* can help to measure Ethernet load), it would be useful to try to identify those clients generating the heaviest NFS load. *nfsstat* can be used to measure NFS load. Since NFS read operations can account for 7 packets per request (as opposed to 2 for most other NFS operations), it is important to take the type of load into account. If a small number of clients are found to be responsible for generating large read loads, buying these particular clients a disk may be a cost-effective solution if the problem of distributing files to these disks appropriately can be solved.

Network Topology

Another factor contributing to excessive delay is network topology. If clients are located many hops away from the servers that they use heavily, their requests will experience long delays. Restructuring the network topology to better distribute load can solve the problem.

Retransmissions

Excessive retransmissions can cause poor performance because the client must wait a long time for the server to respond before it can retransmit a request. Causes of excessive retransmissions include: overloaded servers which drop packets due to insufficient buffering; inadequate Ethernet transceivers which cause packets to be dropped under bursty conditions; physical network errors (such as noisy coaxial cable); and software bugs (such as broadcast storms caused by incorrect broadcast addresses). The *nfsstat* utility can be used to measure the NFS retransmission rate. We can calculate when the rate of retransmissions becomes unacceptable. Average NFS response time to an 8 KB client read request under a low to medium load is around 30 milliseconds. Most clients retransmit after 0.7 seconds. If we assume that the user can feel a 10% reduction in performance then a 3 millisecond reduction is the tolerable limit. This gives an acceptable NFS retransmission rate of 0.42%:

$$\frac{.003\ sec/call}{0.7\ sec/retransmission} = 0.0042\ retransmissions/call$$

Since an 8 KB NFS read request requires 7 packets (1 request, 6 fragmented replies), the error rate of the network must be less than 0.06%:

$$\frac{0.42\%}{7} = 0.06\%$$

The acceptable retransmission rate will vary with request size and frequency, but it is clear that even a small retransmission rate can have a big impact on performance.

The causes of a high retransmission rate are often easy to fix. The first place to look is to see if an individual machine is experiencing more than a few network errors. The *netstat* utility can be used to measure the network error rate. If the problem seems pervasive, analysis of the cabling technology being used may isolate the difficulty. The operating system itself may be incorrectly tuned for optimum network performance. With SunOS 4.0, we found high retransmission rates due to dropped packets. The cause turned out to be that too few Ethernet buffers were allocated to the Ethernet driver. This in turn resulted in a high percentage of Ethernet input errors when packets arrived with no place to go. To fix this problem, we wrote programs to monitor the Ethernet driver for the problem, and to patch the kernel to increase the number of buffers (see Appendix).

Client Bottlenecks

Adding disk or memory to the client can improve performance in two ways: by improving access time and by reducing overall load on the server and network. A client can avoid NFS performance problems for files that are not shared (e.g. swap and temporary files) by using local disks for these files. As disks become smaller, quieter and cheaper, this is becoming an acceptable alternative. Once these disks are used to hold valuable data however, the problems of administering them (e.g., backup and restore) and sharing the data they contain, become more of a problem. In addition, the cost of configuring in the UNIX file system (ufs) code on a client adds about 120 KB to a Sun-3 kernel, which reduces the amount of memory available for other purposes. On diskless clients, more memory can make a big improvement in swapping performance by allowing the client to swap less often. By adding local resources, the demands on the server and the network may be reduced, yielding better system throughput.

While it is easy to improve client performance by adding memory or disk, there are no simple rules of thumb to guide you when determining how cost-effective these improvements are. However, it is easy to see that resources added to the server are more cost effective (because a single instance of a resource can be shared by many) and are easier to administer. Thus, the best place to start when trying to improve performance is with the server.

Server Bottlenecks

The server Ethernet interface may be implemented with old technology (such as 16-bit data paths) which causes decreased throughput. The server CPU could be a bottleneck, although in an environment with a reasonably powerful server (>= 2 MIP machines), this is probably not the case. Processors of this class or above are usually able to keep up with the rate of NFS requests that the Ethernet can deliver. Slower processors (< 1 MIP machines) are not able to keep up with the Ethernet, and even moderate network loads on a slow server could swamp its CPU. Many users who experience NFS performance problems with Sun-3's are surprised that upgrading to a Sun-4 gives no better NFS performance. This is because CPU performance was not the problem. Sun's *perfmeter* utility may be used to measure CPU utilization.

On most NFS servers the limiting factor is the speed of the disk. Most high-speed disks today can sustain from 30 to 40 disk operations per second. Most of the time spent waiting for a disk operation is in seeking or rotational delay. Using a faster disk or disk controller, and spreading the load over multiple disks may help, but this usually only provides an incremental improvement. The best way to improve disk performance is to reduce the number of disk operations.

NFS Server Performance in UNIX

UNIX uses a buffer cache to avoid disk operations whenever possible. The buffer cache is very effective in reducing the time that clients wait for slow disk I/O. It also makes disk I/O more efficient by allowing staging and scheduling of disk operations. A performance gain is made by allowing the disk device driver to schedule several requests at a time to take advantage of the position of the disk arm. Also, the total amount of disk I/O is reduced, since repeat requests may be found in the cache.

The buffer cache on NFS servers is usually very efficient for reads. Hit rates greater than 80% are normal, and hit rates as high as 98% are not uncommon. If the buffer cache hit rate on your server is low, adding more memory to increase the size of the cache should improve NFS read performance. With most current UNIX implementations, buffer cache sizes can be increased by changing system parameters. In SunOS 4.0, adding memory directly increases the buffer cache size since all of memory is effectively a cache.

NFS Bottlenecks

The nature of NFS itself causes performance bottlenecks at the server. To see how these bottlenecks occur, we must first discuss how NFS works.

NFS uses a simple stateless protocol. This protocol requires that each client request be complete and self-contained, and that the server completely process each request before sending an acknowledgement back to the client. If the server crashes, or an acknowledgement is lost, the client will retransmit its request to the server. As a consequence of this policy, the server cannot acknowledge the client's request until data is safely written to non-volatile storage. In this way, the client machine knows exactly how much modified data has been safely stored by the server. This means that the server cannot store modified data in volatile storage (storage that can be lost when the server crashes) to avoid disk writes, because if it did store and acknowledge it and subsequently crash, the acknowledgement would be a lie. The client would assume that its data was safely stored, and would be in for a rude surprise when it is eventually discovered that its data was not available. In fact, the client may not discover that anything went wrong until long after the crash occurs (e.g., if a page that was swapped out is swapped in with different contents). Since a single server stores data for many clients, this surprise could make life painful for an entire user community. By always writing modifications directly through to the disk, NFS ensures that server crashes do not cause data to be lost and crash recovery is trivial.

In addition to writes, other NFS operations cannot be cached to avoid disk writes. Operations that modify the file system in any way: file creation, file removal, attribute modification, etc. add significantly to the amount of filesystem data that the server must write synchronously to disk before responding. For example, when the client creates a new file, the server must update the data and inode blocks for the directory, and the inode block for the file.

This very desirable property of crash-survivability causes performance problems because of the fact that many NFS operations must be synchronously committed to disk. First, these operations take place at disk speeds, not the memory speeds available to cachable operations. Second, since these operations are processed serially, there is no opportunity to optimize the scheduling of the disk arm or the rotation of the disk. Finally, since modifications to the UNIX cache are written through synchronously to disk, there is no opportunity to decrease write disk traffic with cache hits.

If you have already added memory to your server to increase the size of the buffer cache and the server is still too slow, one possible solution is to purchase another server and split the load between the two servers. Not only does this have a large direct cost, there is a significant administrative cost associated with supporting this additional server. We will next examine an alternative solution which gives analogous performance without requiring a new server and without added administrative overhead.

Write Caching to Boost NFS Performance

Unless the server is only supplying read-only access to files (i.e., all modified files, including client swap files, are maintained locally), it is required to synchronously commit some NFS operations to its disks. Since disks are several orders of magnitude slower than memory, this is a tremendous burden. If disk writes could be safely cached in non-volatile memory, significant performance gains could be made.

There are four basic reasons why reliable write caching can boost NFS server performance.

1. Faster Transfer Time

An incoming NFS write request can be secured safely in non-volatile memory and then acknowledged to the client. Because the time to transfer data to memory is far less than the time to transfer data to disk, the turnaround time for NFS update request handling is reduced.

2. Repeat Cache Hits

A single NFS write operation causes two or three writes to the disk. This is because each server write must update not only the data block but the inode and (often) an indirect block as well. Since the same inode is updated for each data block in the file, many disk writes can be saved by rewriting the cached inode. Data blocks may also be found in the cache, although the frequency of these cache hits is significantly less than the hits on the inodes. If data blocks are reclaimed (e.g., as a result of removing a file) before they are flushed, it is possible that a file can be created, written and removed with no disk I/O involved at all. By lessening disk traffic, overall system performance is enhanced.

3. Improved Disk Scheduling

Since the non-volatile memory will continue to retain data until the disk acknowledges that the data has been written, data may be flushed asynchronously from the non-volatile memory. This allows multiple blocks of data to be scheduled together to take advantage of the location of the disk arm. As disk seek times and rotational delays are significant (tens of milliseconds), this represents a major economy.

4. Importance of Update Operations in NFS Traffic

Because UNIX read caching is already very effective, NFS operations that modify file data account for a disproportionately large amount of actual disk traffic. In an environment where NFS operations that modify data comprise about 20% of the operation mix, over 60% of the requests for disk I/O are due to these operations. This is measured by code in the device driver for our write cache which is instrumented to provide statistics on disk access.

As a result of these factors, write caching can greatly increase server performance. In addition to accelerating NFS servers, a write cache can speed up any application requiring synchronous writes to disk. This includes local synchronous operations such as file and directory creation and removal. Many database or transaction systems require local synchronous writes of files and can potentially show significant performance improvements with write caching.

Write Cache Implementation

Prestoserve is Legato's filesystem write cache. It consists of non-volatile memory and a device driver. Non-volatile memory is required since data must not be lost when power fails or the system crashes. We use low-power, highly reliable static RAM, and a triply-redundant set of lithium batteries which can keep data valid for over two years without external power. The device driver intercepts requests for the disk by inserting its own routines into the *bdevsw* and *cdevsw* tables where the original device entries were. The original entries are maintained in a table used by the device driver. In addition, the *reboot*(2) system call is intercepted, so that the driver can flush its buffers when the system is shut down cleanly. Because the write cache device driver looks like a normal disk device driver to the rest of the kernel, all UNIX semantics are maintained. In our current implementation, all disk controllers use the same write cache.

The administrator can select which filesystem will use the write cache (by default, all writable filesystems are selected). When a synchronous write request is issued to an accelerated disk, it is intercepted by the device driver which stores the data in non-volatile memory instead of on disk. Thus, synchronous writes occur at memory speeds. As the cache fills up, it asynchronously flushes cached data to the disk in groups large enough to allow the disk drivers to optimize the order of the writes.

From the point of view of the operating system, the write cache simply makes existing disk drivers faster. Since it is implemented as a device driver and maintains its own data structures in stable storage, the cache is relatively invulnerable to system crashes which are caused by software bugs, administrator error, power failures or hardware failures.

Reliability Considerations

Because it interacts with the server's disk data, it is imperative that all error situations be properly handled by the write cache device driver. The basic axiom followed in the design of the write cache is to preserve data integrity at all times. Some situations that must be handled correctly include:

- orderly shutdown
- system panics
- power failures
- removable disk packs
- disk failures
- attempts to use a cache with dirty data on a different system

To ensure that these situations are properly managed, we take the following precautions. Should the cache ever be disabled or removed for any reason, all data is passed directly through to the disk. Should it prove impossible to flush a block to disk, the block remains in the cache until the disk can be repaired. When the system is shut down normally (e.g., via *shutdown*, *reboot*, or *halt*), all cached data is flushed to disk. This is done so that normal board swapping activities have no potential for causing data to be lost. The write cache should be viewed just like a disk: it is possible to lose data if you really try. By pulling the plug on a running system, removing the board, and turning off the batteries, you can lose data. This is analogous to opening the top of a disk drive and de-magnetizing the surface of the disk. To prevent a board containing valid data from being inadvertently moved into a different machine, the cache maintains the host id of the system it is installed on in non-volatile memory. If a board with valid data is moved to a foreign host, the administrator will be asked upon reboot whether the data in the board should really be discarded or not. When a removable disk pack is removed, it must first be unmounted. As a result, the device close routine is invoked. Our device close routine is interposed between the real device close routine and the rest of the kernel, so we ensure that its invocation causes write cache contents for that device to be flushed. Thus, the contents of the removable disk pack are consistent.

Performance Results

Any filesystem not mounted read-only will benefit from write caching. The implementation required much experimentation to determine the correct cache size and flushing algorithms to employ. The 1MB cache can hold 1-2 seconds of Ethernet traffic at maximum speed, adequate time to flush the entire cache to disk if needed. Empirical results also suggest that the cost/benefit of a larger cache is unwarranted. However, when faster networks (such as FDDI) are employed, a larger cache can be of value. A modified form of Least Recently Used (LRU) replacement is used to efficiently flush the cache and keep enough free buffers available to handle bursty traffic. Care has been taken to ensure that the cache is also used effectively to assist the UNIX buffer cache when appropriate for asynchronous requests. For example, read requests not satisfied by the UNIX buffer cache may be satisfied from the write cache, and blocks repeatedly updated asynchronously (e.g., superblock, cylinder groups) may be written to the write cache instead of to disk.

Our investigations show that users have an acute awareness of poor NFS performance, although some would prefer that response time (the time to process an individual request) be improved while others express a preference for throughput (the number of clients a server can support). To objectively measure response time and throughput, we have devised a benchmark called *nhfsstone* which places an artificial load on an NFS server based on NFS operation mixes obtained from working systems.

The *nfsstat* utility provided with the Sun NFS reference port can be used to determine the mix of NFS operations in a particular environment, and this mix is used to drive the *nhfsstone* benchmark. To chart response time, the turnaround time for handling a given load is measured. Throughput is the inverse: how many NFS operations per second can be handled at a given level of response time. It is important to use a benchmark that simulates a realistic load on the server to avoid getting a distorted view of performance. For example, measuring the time to copy a large file over NFS on a server with write caching shows at least a 300% improvement. This is somewhat misleading, since file copying does not dominate the workload of a typical system. However, it does illustrate that some applications may be accelerated more than others.

Nhfsstone Results

To evaluate performance, we used *nhfsstone* to generate a reproducible load on a server. The mix of NFS operations used (note that 19% of these operations modify data) follows in figure 1. Operations representing less than 0.5% of the mix are not included.

% in Mix	*NFS Operation*
13	getattr
1	setattr
34	lookup
8	readlink
22	read
15	write
2	create
1	remove
3	readdir
1	fsstat

Figure 1: *Typical Diskless NFS Operation Mix*

Results vary depending upon the configuration of client, server and disk being used. Since *nhfsstone* was designed to run on a single client, it is important to use a client powerful enough to generate significant loads. For the results given here we used a Sun-3/470 client machine and a Sun-4/260 server running SunOS 4.0.3. The server had 24MB memory, a Xylogics 7053 controller, and a CDC 9720-850 disk. *nhfsstone* divided its load between two disk partitions to simulate normal use (e.g., a */home* and a */export* partition). The results with this mix at varying loads are shown in figure 2.

We observe that with write caching, response time stays acceptable, even as the load climbs over 100 NFS operations per second. Without write caching the server cannot process loads exceeding 70 NFS operations per second. We have also noticed that with write caching, CPU utilization at high loads begins to approach 100%. Servers without write caching begin thrashing before utilizing 100% of the CPU. Thus, a faster CPU will not make much difference in improving NFS performance until the I/O bottleneck at the disk is reduced. *nhfsstone* currently measures a single client loading a server. With write caching the server can comfortably handle more load than one client can generate, so we are unable to report on server behavior at loads much beyond 120 NFS operations/second.

The numbers we have reported reflect an NFS operation mix in which about 20% of the operations modify data. This corresponds to a fairly typical diskless NFS environment. If a response time of 40 ms/operation is acceptable, then a server without write caching can support about 40 operations per second. The same server with write caching can support about 110 operations per second. This translates into nearly a 200% increase in the number of clients that can be supported at this level of responsiveness. If local disks are used for temporary files and swapping, the percentage of NFS operations that modify data typically drops to about 10%.

Load(ops/sec)	*Non-cached Response* Time (ms/op)	*Cached Response* Time (ms/op)
10	11	9
20	19	10
30	27	11
40	39	15
50	54	17
60	71	19
70	93	22
80	infinite	26
90	infinite	32
100	infinite	35
110	infinite	40
120	infinite	46

Figure 2: *Results at Varying NFS Operation Loads*

In PC NFS, clients typically write directly to a file in 512-byte chunks. This translates into an 8 KB write requiring not 1 8 KB synchronous filesystem write but 16 512-byte synchronous filesystem writes on the server. As each of these writes requires an inode (and possibly an indirect block as well) to be written synchronously, at least 32 and as many as 48 synchronous disk write operations will be required. With write caching, an average of one asynchronous disk write per 8 KB of file will be generated. Thus, PC NFS benefits from write caching can be impressive.

To really understand how an individual site will benefit from write caching, each site will have to measure its own mix of NFS operations. The higher the percentage of operations that modify data, the more beneficial write caching will be. Also, the higher the load on the server, the more noticeable the improvement will be. To get a feel for how different mixes of NFS operations affect the throughput increase from write caching, we present some ranges we have measured in figure 3. We use "dataless" to indicate a client that uses a local disk for temporary files, root partition and swapping, and NFS for all other file access.

Clients	*Modify Mix*	*Throughput Increase with* Write Caching
All dataless	5%	10-40%
Mostly dataless	10%	30-80%
Diskless/dataless	15%	60-120%
All Diskless	20%	80-200%
PC/NFS	10%	200+%

Figure 3: *Effect of Different Mixes on Throughput*

Other Solutions to NFS Performance

There are several other methods that can be employed to achieve NFS performance benefits. Many of these methods have drawbacks or limitations that make them unacceptable for some users.

RAM disks are simply large amounts of memory (which may have battery back-up) with an interface that emulates a disk interface (SCSI for example). Unlike a RAM disk, a write cache serves as a front end to normal disks, so the benefits of fast memory are shared among all the disks on the server. A RAM disk has a finite capacity, and the expense of providing large amounts of storage is prohibitive.

Some vendors change the kernel NFS code to not do synchronous updates for NFS requests that modify filesystem data. However, this *dangerous mode* of NFS operation makes clients susceptible to mysterious losses of large amounts of data in the event of a server crash. It is also a violation of the NFS protocol specification. Some vendors use this method without warning their customers; it is always a good idea to check.

A new and better UNIX filesystem [Bra89] could improve filesystem write performance. Significant improvements from advanced techniques such as disk striping and the use of disk arrays [Kat88] may also accelerate filesystem write performance. These solutions can be expensive, and require major changes to the UNIX file system.

Attention to correct implementation and underlying protocol issues can yield significant NFS performance improvements (see [Now89], and [Jus89]). We expect to see these enhancements make their way into vendor's implementations of NFS in the near future. Other NFS modifications that address cache consistency, such as [Sri89] can reduce the frequency of update traffic (for example, setting the sticky bit on swap files tells the server that inode modifications need not be written synchronously as long as the size of the file remains constant and no new blocks are allocated). However, none of these enhancements completely eliminate the need for synchronous writes and the performance penalty of writing at disk speed. Because a server with a write cache can respond quickly to update requests, clients can issue new requests more quickly, generating more load on the server. The write cache is able to handle this increased load by immediately buffering incoming requests safely and flushing to disk at its leisure.

Summary

While many factors can contribute to poor NFS performance, server disk I/O is often the leading contributor. Write caching can improve NFS server performance by allowing the server to add memory to cache update requests safely, much as existing technology allows the addition of memory to improve the efficiency of read requests. By making caching available to operations that modify files, we can put back the performance NFS took away from UNIX.

Appendix

Legato has made publicly available some unsupported software (in source form) which is intended to help measure and improve NFS performance. To obtain this software, send electronic mail to *request@Legato.COM* or to *{sun,uunet}!legato!request* with a subject line: *send unsupported <product>*. Please include your return address by including it in the body of your message with a *path* command: *path <return address>*. A subject line of *index unsupported* will produce a list of all currently available software which includes:

nhfsstone	– NFS load generating and performance measuring benchmark
etherck	– Ethernet monitoring tool and kernel patch to increase available network buffers
netck	– Heuristic program to locate general network performance problems

References

[Bog88] David R. Boggs, Jeffrey C. Mogul, and Christopher A. Kent, "Measured Capacity of an Ethernet: Myths and Reality," in *SIGCOMM '88 Symposium on Communications Architectures and Protocols*, August, 1988.

[Bra89] Andrew Braunstein, Mark Reiley, and John Wilkes, "Improving the efficiency of UNIX file buffer caches," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December, 1989.

[Jus89] Chet Juszczak, "Improving the Performance and Correctness of an NFS Server," in *1989 Winter USENIX Technical Conference, San Diego*, 1989.

[Kat88] R.H. Katz, G.A. Gibson, and D.A. Patterson, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *ACM SIGMOD 88*, June 1988.

[Now89] Bill Nowicki, "Transport Issues in the Network File System," in *Computer Communication Review, Vol. 19 #2*, April, 1989.

[San85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, "Design and Implementation of the Sun Network Filesystem," in *USENIX Summer Conference Proceedings, Portland Oregon*, 1985.

[Sri89] V. Srinivasan and Jeffery C. Mogul, "Spritely NFS: Experiments with Cache-Consistency Protocols," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December, 1989.

HIT-Multicode – Implementation of a multilingual Word Processing System

Thomas Merz

InterFace Connection GmbH
Weißenburger Str. 43
8000 München 80
West Germany

ABSTRACT

Current internationalisation concepts are suffering from a major disadvantage: they are limited to one particular codeset. This codeset may be designed according to the needs of a specific language, even based on a non-Latin alphabet such as Greek or Arabic. However, the requirement for mixing of different languages and scripts which frequently arises in modern communications, cannot be met by such systems. The only way to cope with this problem efficiently is to use multicode techniques, i.e. software which abandons the restriction of a single codeset.

Subject of this paper is the implementation of HIT-Multicode, the multi-lingual extension of a standard UNIX word processor. It allows arbitrary intermixing of character-based scripts and therefore creation of documents using languages based on the Latin, Greek, Cyrillic or Arabic alphabet.

Both, the techniques required for handling several codesets and the extensions to the user interface, are described. To start with, we will take a look at several scripts and the evolution of corresponding codesets. Also included is a technical introduction to the standard HIT system, which provides the basis for the multicode implementation.

1. The next Step in Internationalisation

Word processors have always been the first to introduce new methods of internationalisation, since this is the area in the computing environment where the users native language becomes centre of interest. Poor Mr Dürre or Mr Franc¸ois or Mr Dubcek would rather like to write their names correctly than having their machines giving boot time messages in German, French or Czech. Modern word processors deal with character sets that include all necessary special characters for Latin-based scripts. Special versions are available for Greek, Cyrillic or Arabic, each combined with ASCII. But arbitrary intermixing of any of these languages among each other and all the Latin special characters is not possible. Whereas this is not a severe limitation for operating systems, it is for word processors and therefore highly desirable to offer means that might be called the next step in internationalisation: multicode support. The presented methods allow manipulation of documents using a mixture of several character-based languages and scripts, the most important ones being Latin, Greek, Cyrillic and Arabic.

The only group of writing systems beyond the scope of this approach are the syllable or word orientated scripts of the Far East.

2. Character-based Scripts and international Codesets

To us, the Latin alphabet, consisting of the letters A-Z in upper and lower case, is the most familiar one. Nevertheless it will only satisfy the phonetic and linguistic needs of most european languages once several special characters are added. These special characters are essential in the written representation of these languages. Consider København, Lübeck, la Révolution franc¸aise, Jiri or señor García written in plain ASCII! Many of these additional characters are constructed by combining an ordinary character with a diacritic mark or accent, while others (e.g. the German ß) are altogether new. This is reflected in the construction of ISO 6937, which contains all characters necessary for latin-based scripts. Simple characters are given an 8-bit code, whereas combined characters are represented by two bytes, first the diacritic mark and then the basic character. This character set however is difficult to handle, since the length of a string on the screen bears no relationship to the number of bytes in its representation. Therefore ISO 8859 [ISO84a] provides regional subsets in a proper 8-bit representation. ISO 8859/1, for example, provides all characters for the languages of northern and western Europe.

There are two more european scripts, viz Greek and Cyrillic. Both are character-based and use some additional diacritic marks. Whereas the Greek script is only used for the Greek language, the Cyrillic script is used for several eastern languages such as Russian, Bulgarian and Serbo-Croat. Both scripts can be coded in an 8-bit character set: ISO 8859/7 contains ASCII and Greek, ISO 8859/5 ASCII and Cyrillic.

The semitic scripts, the most important of which is Arabic, are also character based. They are written from right to left. Arabic incorporates some additional idiosyncrasies. These shall only be mentioned briefly here (more details may be found e.g. in [Bey88a] or [Bec84a].) The flowing and cursive nature of the Arabic script is achieved by connecting the letters to their neighbours according to certain rules. This implies that most characters may appear in four different forms, depending on the context, namely the position of the character in the word: at the beginning, in the middle, at the end or isolated. In addition, the use of ligatures is very common. The determination of the correct letter shape is referred to as context analysis. The vowels in Arabic are not part of the basic alphabet. They appear as "accents" above or below the consonants and are often left out. Whereas Arabic text is written from right to left, insertions of latin text or numbers are written from left to right. Such insertions, for example foreign names, appear very often. The Arabic character set, together with ASCII, is contained in the 8-bit code of ISO 8859/6.

Some remarks on Asian scripts should indicate the limits of the methods described in the next chapters. Chinese, for example, is written with thousands of ideographic Hanzi characters, each representing a single word. Moreover, ideograms may be combined, thus achieving a new meaning. Coding of these ideograms can only be done with double-byte character sets. In Japan, two syllable-based scripts named Hiragana and Katagana are used. Since these are not sufficient, chinese ideographic characters are added. The existence of three concurrent scripts makes Japanese the most difficult writing system of the world.

3. An Introduction to the standard HIT System

HIT is a word processing system which is available on many UNIX-Systems and provides all the functions necessary to modern communication with the comfort one may expect from a mature software product. Features and functions, ranging from keyboard macros to POSTSCRIPT support, from file management to database access, from "undo" facility to programmable text processing, cannot and need not be discussed here.

Instead, a technical introduction to the monocode version of HIT as well as an explanation of the basic methods of internationalisation will be given in the following. This is necessary for an understanding of the extensions employed in the multicode version.

Although there is also a version of HIT running on the Collage window system, most HIT installations are running on alphanumeric terminals connected with serial lines. For this reason, and because it is the basis for the multicode implementation, we restrict ourselves here to the alpha-version, as it is called.

3.1. HIT Character Set and HCHARs

The standard HIT character set is based on ISO 6937 with minor extensions, e.g. semigraphic characters. The overall number of accented and nonaccented letters and other characters is 346. (The problem of mapping these characters to a limited keyboard is dealt with in the next sections). According to ISO 6937, ordinary letters are stored on disk as 8-bit characters, whereas accented letters are represented by a two-byte combination of a non-spacing accent and the letter to be accented.

A HIT text file may contain much more information than just a plain character sequence:

- information on structure and format of the text (user tab stops, margins, indentation, justification etc.),
- additional printing instructions (typeface to be used, line spacing etc.),
- form-related field attributes (numerical, justified etc.),
- text attributes (bold, italics, underlined etc.) and hyphenation hints.

The first two kinds are line orientated, whereas character attributes are handled in start-stop-semantics (bold on - bold off), since this more closely resembles the application of text attributes to whole words and phrases. This method, as clearly as it leads to disk space preservation, also leads to difficulties where internal handling of characters is concerned. These are due to the heterogeneous nature of the storage format, i.e. a combination of attributes, single and double-byte characters.

For this reason HIT files are translated in what we call HCHAR-Format when read into memory. HCHARs, in contrast to ordinary chars, consist of 32 bits. The first two bytes (called the charpart) are used to locate the character in a fictitious 9-bit character set while the remaining two bytes (called the attrpart) hold the attributes related to the charpart. The charpart is also used to hold additional information, necessary due to the complex rules of hyphenation found in some languages, esp. German and Dutch.

3.2. The Hardware Interface

HIT is available in several country-dependent configurations, on a variety of machines, and supports an even greater variety of peripherals. In order to master this chaos easily and efficiently, HIT uses special tables describing the different hardware interfaces. These tables are plain text files (i.e. HIT-documents) and may therefore be modified by the distributor, the system administrator or even a knowledgeable user. To increase performance, these are translated into binary format through the use of special compilers.

There is a HIT-specific termcap as an extension of the well-known UNIX-concept. The original termcap is not adequate for a system offering nearly 350 characters, not mentioning screen attributes and additional control functions, e.g. switching between different character sets of a terminal. Termcap interpretation and character output is done by a set of library functions called the Screenmanager, which also provide support for the usage of separate screen areas. This Screenmanager does some sort of opimization to minimize the number of transmitted characters. This is achieved by activating the suitable delete/insert functions of the terminal and stripping of characters which are already on the correct position on the screen.

Keyboard mapping is done via the keycap, which contains codes or code sequences for function keys and ordinary characters. Special features facilitate input of non-standard characters: some keys may be defined non-spacing, implying that they have to be followed by another key to produce a character. This resembles the treatment of accented letters in ISO 6937, but the method is not limited to a particular subset of the character set. Other characters may be entered by combining single keystrokes with the combi-key. This method is used for defining mnemonic combinations for characters from foreign languages not available on the keyboard in use. For example, you could type "? <combi> ?" to produce the Spanish '¿' on non-Spanish keyboards or "A <combi> E" to produce the Danish character 'Æ'.

The next kind of table is the printcap for a description of printer facilities. Here we have several functions in addition to character codes or sequences to be output: text attributes as in termcap, line spacings, different fonts, paper tray changes and so on. When the document has to be printed, an output filter interprets the printcap and translates the descriptive format rules into adequate output for the printer.

3.3. Means of Internationalisation

To ensure easy translation and adaptation to local customs, HIT has anticipated functions that have now been standardized by X/OPEN [XOP89a] in the concept of NLS.

Messages and help texts, like the hardware tables, are prepared as HIT files and then compiled to binary format. Tables and messages are selected via command line options or environment variables. Thus it is possible to run several language versions with the same installation. Keyboards are configured according to national customs (e.g. QWERTY or QWERTZU) using keycap. Alien characters can be produced with the combi-key (which corresponds to the compose-key in NLS environments). So every user has access to the whole character set whilst retaining his local keyboard assignment.

According to this concept, the HIT system consists of two parts: the basic unit containing all executable and system files and the second unit containing all language- and country-dependent files like messages, help-files, keycaps and so on.

4. HIT-Multicode

4.1. Design Goals of HIT-Multicode

The main principle may be described in a few words: HIT-Multicode should offer the same standard word processing functions for different character-based scripts, with possibility to intermix all languages easily. Although there are no definitive multicode standards, the codesets used should be aligned to international norms.

However, a practicable and efficient realisation imposes some additional requirements to be met. Implementing a complex word processor from scratch is heavy duty: the whole HIT system is constituted by 250.000 lines of code! Therefore, one of the major goals in the design of the multilingual version has been to change the existing HIT structures very carefully to allow the reuse of standard editing and processing functions without great effort. Some examples of these functions are searching and replacing of text, paragraph or page formatting and mailmerging.

4.2. 4.2 Treatment of Multiple Character Sets

As has been pointed out in [Bec84], storage of multilingual documents is most efficiently done by using separate character sets and switching between them with a change script byte. This indicates a change of the character set and must be followed by the number of the character set to which the next characters belong. Since HIT is line-orientated, the language code is repeated at the beginning of each line. This redundancy must be accepted in order to ensure correct language treatment for random line access: each line may be processed separetely as it carries all necessary information. The character sets are treated independently so we can even intermix the heterogeneous Latin character set of ISO 6937 with homogeneous sets for Greek, Cyrillic and Arabic. 8-bit character sets for these languages are defined in ISO 8859. These codesets have been slightly modified for our purposes: Control characters used by HIT have been inserted, so that the first 32 code positions incorporate the same control function for each set. Moreover, the ASCII section of these sets has been removed except for common characters such as punctuation marks, numbers and brackets. Multiple coding of these characters and the control characters might look like unnecessary redundancy, whereas it is much more efficient than switching code sets for each comma or bold-attribute.

For characters from the Latin alphabet we switch to the complete Latin character set. The Latin character set of the standard HIT version as well as the additional Greek, Cyrillic and Arabic character sets of HIT-Multicode are shown in the appendix. Note that the notion of both Hindi and Arabic numbers in the Arabic character set is not ambiguous, but aimed at allowing a choice of one of these representations. This is done through appropriate termcap and printcap entries and will not affect the internal representation.

For incore processing it is desirable to have the character set information joined to each character inseparately. Fortunately, there is some space left in the HCHAR representation of a character, as the 16 bits of the charpart hold only a 9-bit character set in the standard Latin implementation. When reading a line of text from file into memory, the language information is included in the charpart of the HCHAR structure. Thus, each character is assigned a unique code in the internal text format. This ensures that those functions that are not concerned about the contents of a HCHAR will still operate without changes.

In both the external and internal formats, characters are stored in phonetical order. This means that the storage sequence is equivalent to the spoken and not to the written order of the text, which is very important for easy processing of bidirectional text, e.g. a mixture of right-to-left Arabic and left-to-right English. The correct output sequence must be determined by the screen and printer drivers.

4.3. Extended Hardware Interface

The most natural way of extending the HIT hardware interface for multicode support is to provide a separate hardware table for each language. Moreover, this method has the advantage of upward compatibility of both tables and related compilers.

Some differences in the treatment of the tables shall be mentioned. The termcap contains terminal- as well as character-specific information. As the terminal info is the same for all languages, it suffices to read it into memory once and to ignore it in all following language-dependent termcaps. Several termcaps are used at the same time, since characters of different sets could appear anywhere in the document or message. When a character is to be displayed, the Screenmanager decides on the termcap and then reads the desired terminal code from the selected one. Of course, all termcaps are held in memory simultaneously.

In contrast to this, keycaps are used one at a time. They are exchanged at the moment the user requests another input language. The use of printcaps for the output filter is similar to that of termcaps. They are used in parallel and some additional support, e.g. for font changes in the different languages, has been built into the printcap-interpreting tool filter.

All tables describing the same piece of hardware are given the same basic name, differing only in a language-dependent suffix. This ensures easy selection via command line options or environment variables without confusing the user with dozens of names.

4.4. The User Interface

There is one obvious point where the user interface is affected directly by multicode activities: this is the change of the input language. HIT-Multicode offers two possibilities: a pull-down window may be called which offers all supported languages. This allows arbitrary intermixing of languages. Fast switching of languages for bilingual text is possible with one keystroke. The last used language is remembered and is used for the next fast language switch. This is very convenient for writing insertions in another language and then returning to the previous one. As an aid for the user in handling nearly 600 characters, a help screen is available showing the current keyboard assignments. The current language is indicated by an appropriate symbol in the system line.

A rather subtle point in the user interface not so apparent may be explained by an example: Consider a user working on a system configured with English messages. After writing a passage of Greek text, he might want to quit the editor, therefore having to answer the system message "Do you want to save your document (y/n) ?" in English and having to change to English keyboard mode explicitly before answering the question. This situation is remedied by the automatic language switch. Whenever a system message has to be confirmed or responses to be entered, the current language is temporarily changed to the appropriate message language and afterwards changed back to the original text language.

4.5. Additional Support for the Arabic Script

As has been pointed out in chapter 2, handling of the Arabic script requires some additional processing concerning terminal and printer output, primarily context analysis. There are several possibilities to do this: either application programs contain special functions for this task, or the operating system postprocesses all output. The first method has the disadvantage of having to change every application program with screen output, the second requires significant changes in the kernel. Both share the disadvantage of burdening the host. We prefer a third solution, which places the context analysis into the firmware of the output device, thus releaving the host and allowing standard firmware to operate without changes (provided that an 8-bit character set is supported). Also the implementation of new software is simplified. Despite these advantages, there are some cases where it is not sufficient to do the context analysis in the terminal or printer firmware, especially in word processing applications: cursor positioning according to the physical layout of the text instead of spoken order has to be under control of the application; the correct formatting of proportional text and mixed Arabic-Latin parts requires that the context analysis is already done. As modern firmware is written in the C language, it is obvious to use the context analysis functions in a common library to ease development and assure identical behaviour. We are in negotiations with a major terminal developper concerning this subject. Awaiting this common library for context analysis, no additional support for the Arabic language is implemented at the moment except screen handling of bidirectional text.

5. Conclusion

Multicoding techniques are the key in the current state of internationalisation. In the field of word processing, some problems remain to be solved, for example hyphenation and spell-checking tools for all languages and multilingual text. Then the last step in internationalisation can be taken: full integration of the Asian scripts.

References

[ISO84a] "8-Bit single-byte coded graphic character sets.," *ISO 8859.*

[XOP89a] *X/OPEN X/Open Portability Guide.*

[Bec84a] Joseph D. Becker, "Multilingual Word Processing," *Scientific American*, pp. 96-107., July 1984.

[Bey88a] Pascal Beyls, "The Arabisation of UNIX.," *EUUG Autumn 1988 Conference Proceedings*, pp. 253-264.

6. Appendix

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
00	HEOS	HCC	▨	0	@	P	`	p	|	▨	NBSP	•	HHYPH	—	Ω	K
01	HEOF	HEZ	!	1	A	Q	a	q	└	▨	¡	±	`	^{1}	Æ	æ
02	HFE	HF5	"	2	B	R	b	r	┴	▨	¢	^{2}	´	®	Đ	đ
03	HUS	HBR	#	3	C	S	c	s	┘	▨	£	^{3}	^	©	ª	ð
04	HHO	▨	$	4	D	T	d	t	├	▨	$	×	~	™	Ħ	ħ
05	HTI	▨	%	5	E	U	e	u	┼	▨	¥	µ	¯	♪	▨	ı
06	HKU	▨	&	6	F	V	f	v	┤	▨	#	¶	˘	¬	Ĳ	ĳ
07	HTR	▨	'	7	G	W	g	w	┌	▨	§	·	˙	¦	Ŀ	ŀ
08	HGE	▨	(	8	H	X	h	x	┬	▨	¤	÷	¨	▨	Ł	ł
09	HTAB	▨	)	9	I	Y	i	y	┐	▨	‘	’	▨	▨	Ø	ø
10	HLF	▨	*	:	J	Z	j	z	─	▨	“	”	•	▨	Œ	œ
11	HNU	HESC	+	;	K	[	k	{	▨	▨	«	»	¸	▨	º	ß
12	HRB	▨	,	<	L	\	l	|	▨	▨	←	1/4	_	1/8	Þ	þ
13	HCR	▨	-	=	M	]	m	}	▨	▨	↑	1/2	˝	3/8	Ŧ	ŧ
14	HDT	▨	.	>	N	^	n	~	▨	▨	→	3/4	˛	5/8	Ŋ	ŋ
15	HIT	HEX RET	/	?	O	_	o	HMETA ESC	▨	▨	↓	¿	ˇ	7/8	ʼn	▨

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
00	À	É	Ĩ	Ô	Û	à	é	ĩ	ô	û	▨	▨	▨	▨	▨	▨
01	Á	Ê	Ī	Õ	Ũ	á	è	ī	õ	ũ	▨	▨	▨	▨	▨	▨
02	Â	Ē	İ	Ō	Ū	â	ē	▨	ō	ū	▨	▨	▨	▨	▨	▨
03	Ã	Ė	Ï	Ö	Ŭ	ã	ė	ï	ö	ŭ	▨	▨	▨	▨	▨	▨
04	Ā	Ë	Į	Ő	Ü	ā	ë	į	ő	ü	▨	▨	▨	▨	▨	▨
05	Ă	Ę	Ĵ	Ŕ	U̇	ă	ę	ĵ	ŕ	u̇	▨	▨	▨	▨	▨	▨
06	Ä	E	Ķ	Ŗ	Ű	ä	e	ķ	ŗ	ű	▨	▨	▨	▨	▨	▨
07	Ȧ	▨	Ĺ	R	Ų	ȧ	ǵ	ĺ	r	ų	▨	▨	▨	▨	▨	▨
08	Ą	G̀	Ļ	Ś	Ẁ	ą	g̀	ļ	ś	ẁ	▨	▨	▨	▨	▨	▨
09	Ć	Ğ	L	Ṡ	Ý	ć	ğ	l	ṡ	ý	▨	▨	▨	▨	▨	▨
10	C̀	Ġ	Ń	Ş	Ỳ	c̀	ġ	ń	ş	ỳ	▨	▨	▨	▨	▨	▨
11	Ċ	Ģ	Ñ	S	Ẏ	ċ	▨	ñ	s	ÿ	▨	▨	▨	▨	▨	▨
12	Ç	H̀	Ņ	Ţ	Ź	ç	h̀	ņ	ţ	ź	▨	▨	▨	▨	▨	▨
13	C	İ	N	T	Ż	c	i	n	t	ż	▨	▨	▨	▨	▨	▨
14	D	Í	Ò	Ù	Z	d	í	ò	ù	z	▨	▨	▨	▨	▨	▨
15	Ì	Î	Ó	Ú	▨	è	î	ó	ú	▨	▨	▨	▨	▨	▨	▨

Latin character set of the HIT-system

				b	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b	b	b	b		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0	0	0	0	00	HEOS	HCC	SP	0	@	▨	▨	▨	|	▨	NBSP	°	ΐ	Π	ΰ	π
0	0	0	1	01	HEOF	HEZ	!	1	▨	▨	▨	▨	└	▨	ʻ	±	Α	Ρ	α	ρ
0	0	1	0	02	HFE	HF5	"	2	▨	▨	▨	▨	┴	▨	ʼ	²	Β	▨	β	ς
0	0	1	1	03	HUS	HBR	#	3	▨	▨	▨	▨	┘	▨	£	³	Γ	Σ	γ	σ
0	1	0	0	04	HHO	▨	$	4	▨	▨	▨	▨	├	▨	▨	΄	Δ	Τ	δ	τ
0	1	0	1	05	HTI	▨	%	5	▨	▨	▨	▨	┼	▨	▨	΅	Ε	Υ	ε	υ
0	1	1	0	06	HKU	▨	&	6	▨	▨	▨	▨	┤	▨	¦	Ά	Ζ	Φ	ζ	φ
0	1	1	1	07	HTR	▨	'	7	▨	▨	▨	▨	┌	▨	§	·	Η	Χ	η	χ
1	0	0	0	08	HGE	▨	(	8	▨	▨	▨	▨	┬	▨	¨	Έ	Θ	Ψ	θ	ψ
1	0	0	1	09	HTAB	▨	)	9	▨	▨	▨	▨	┐	▨	©	Ή	Ι	Ω	ι	ω
1	0	1	0	10	HLF	▨	*	:	▨	▨	▨	▨	─	▨	▨	Ί	Κ	Ϊ	κ	ϊ
1	0	1	1	11	HNU	HESC	+	;	▨	[	▨	{	▨	▨	«	»	Λ	Ϋ	λ	ϋ
1	1	0	0	12	HRB	▨	,	<	▨	\	▨	|	▨	▨	¬	Ό	Μ	ά	μ	ό
1	1	0	1	13	HCR	▨	-	=	▨	]	▨	}	▨	▨	HHYPH	½	Ν	έ	ν	ύ
1	1	1	0	14	HDT	▨	.	>	▨	^	▨	~	▨	▨	▨	Ύ	Ξ	ή	ξ	ώ
1	1	1	1	15	HIT	HEX RET	/	?	▨	_	▨	HMETA ESC	▨	▨	—	Ώ	Ο	ί	ο	▨

Greek character set of HIT-Multicode

				b.	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b.	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b.	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b.	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b.	b.	b.	b.		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0	0	0	0	00	HEOS	HCC	SP	0	@	▨	▨	▨	│	▨	NBSP	А	Р	а	р	N°
0	0	0	1	01	HEOF	HEZ	!	1	▨	▨	▨	▨	└	▨	Ё	Б	С	б	с	ё
0	0	1	0	02	HFE	HFS	''	2	▨	▨	▨	▨	┴	▨	Ђ	В	Т	в	т	ђ
0	0	1	1	03	HUS	HBR	#	3	▨	▨	▨	▨	┘	▨	Ѓ	Г	У	г	у	ѓ
0	1	0	0	04	HHO	▨	$	4	▨	▨	▨	▨	╟	▨	Є	Д	Ф	д	ф	є
0	1	0	1	05	HTI	▨	%	5	▨	▨	▨	▨	┼	▨	Ѕ	Е	Х	е	х	ѕ
0	1	1	0	06	HKU	▨	&	6	▨	▨	▨	▨	┤	▨	І	Ж	Ц	ж	ц	і
0	1	1	1	07	HTR	▨	'	7	▨	▨	▨	▨	┌	▨	Ї	З	Ч	з	ч	ї
1	0	0	0	08	HGE	▨	(	8	▨	▨	▨	▨	┬	▨	Ј	И	Ш	и	ш	ј
1	0	0	1	09	HTAB	▨	)	9	▨	▨	▨	▨	┐	▨	Љ	Й	Щ	й	щ	љ
1	0	1	0	10	HLF	▨	*	:	▨	▨	▨	▨	─	▨	Њ	К	Ъ	к	ъ	њ
1	0	1	1	11	HNU	HESC	+	;	▨	[	▨	{	▨	▨	Ћ	Л	Ы	л	ы	ћ
1	1	0	0	12	HRB	▨	,	<	▨	\	▨	|	▨	▨	Ќ	М	Ь	м	ь	ќ
1	1	0	1	13	HCR	▨	-	=	▨	]	▨	}	▨	▨	HHYPH	Н	Э	н	э	§
1	1	1	0	14	HDT	▨	.	>	▨	^	▨	~	▨	▨	Ў	О	Ю	о	ю	ў
1	1	1	1	15	HIT	HEX RET	/	?	▨	_	▨	HMETA ESC	▨	▨	Џ	П	Я	п	я	џ

Cyrillic character set of HIT-Multicode

				b	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b	b	b	b		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0	0	0	0	00	HEOS	HCC	SP	0 ٠	@	░	`	░	|:	░	NBSP	▨	▨	ذ	ـ	ِ
0	0	0	1	01	HEOF	HEZ	!	1 ١	░	░	░	░	┕	░	▨	▨	ء	ر	ف	ّ
0	0	1	0	02	HFE	HF5	"	2 ٢	░	░	░	░	┴	░	▨	▨	آ	ز	ق	ْ
0	0	1	1	03	HUS	HBR	#	3 ٣	░	░	░	░	┘	░	▨	▨	أ	س	ك	▨
0	1	0	0	04	HHO	░	$	4 ٤	░	░	░	░	├	░	¤	▨	ؤ	ش	ل	▨
0	1	0	1	05	HTI	░	%	5 ٥	░	░	░	░	┼	░	▨	▨	إ	ص	م	▨
0	1	1	0	06	HKU	░	&	6 ٦	░	░	░	░	┤	░	▨	▨	ئ	ض	ن	▨
0	1	1	1	07	HTR	░	'	7 ٧	░	░	░	░	┌	░	▨	▨	ا	ط	ه	▨
1	0	0	0	08	HGE	░	(	8 ٨	░	░	░	░	┬	░	▨	▨	ب	ظ	و	▨
1	0	0	1	09	HTAB	░	)	9 ٩	░	░	░	░	┐	░	▨	▨	ة	ع	ى	▨
1	0	1	0	10	HLF	░	*	:	░	░	░	░	─	░	▨	▨	ت	غ	ي	▨
1	0	1	1	11	HNU	HESC	+	;	░	[	░	{	░	░	▨	؛	ث	▨	ً	▨
1	1	0	0	12	HRB	░	,	<	░	\	░	|	░	░	،	▨	ج	▨	ٌ	▨
1	1	0	1	13	HCR	░	-	=	░	]	░	}	░	░	░	▨	ح	▨	ٍ	▨
1	1	1	0	14	HDT	░	.	>	░	^	░	~	░	░	▨	▨	خ	▨	َ	▨
1	1	1	1	15	HIT	HEX RET	/	?	░	_	░	HMETA ESC	░	░	▨	؟	د	▨	ُ	▨

Arabic character set of HIT-Multicode

An X-windows Teletext Service for UNIX

George M. Taylor
Jim Reid

Department of Computer Science,
Strathclyde University,
Glasgow,
Scotland,
G1 1XH.
gtaylor@cs.strath.ac.uk
jim@cs.strath.ac.uk

ABSTRACT

Teletext information in the UK has been available on all four of the national television channels for some time now. The aim of this project is to provide the Computer Science Department with a network-wide, UNIX-based service for viewing the broadcast teletext information. This will use the X window system to present the information on a bit-mapped workstation display. An alternative display for dumb terminals will also be developed as part of this project.

In this paper we describe a final year student project to design and implement such a service. The work entails the design and construction of a decoder and interfacing it to a UNIX system, development of system software – a device driver, daemons and client/server processes – and a user interface. The paper presents an overview of the project and the problems encountered in its development.

1. Introduction

The development of this teletext service can be split into three sections. The first concerned the design of an interface to decode the teletext information from the broadcast television signal and present this information to a UNIX system. The second stage was to develop a simple client/server model to present teletext pages to application processes that could do something meaningful with them. Finally, the raw teletext information had to be interpreted by an application which would then display the data.

2. Overview of Teletext

Teletext is implemented as an additional information service included with a broadcast television signal, using a portion of "spare" bandwidth at the end of each video frame. This extra information can be extracted from the television signal by a special decoder and displayed on a television screen. In the UK, the four national television channels offer teletext services, consisting mainly of news, sports reports and details of the day's television schedules.

The teletext data is divided into pages, each page containing up to 2 Kbytes of information. Each frame (or page) of teletext can be further divided into 960 bytes of display (24 lines of 40 characters), using a special character set comprising alphanumerics, simple graphics characters and control codes for selecting colour and so on. The remaining bytes are used to hold a CRC checksum, details of any relevant international character sets and teletext control information.

3. Project Design and Implementation

The project comprises two parts: the design and construction of the hardware and the design and implementation of the software. A detailed description of these parts now follows. Decisions made when designing the hardware influenced the software design and *vice versa.* These are discussed in the sections below.

3.1. Hardware Development

A variety of factors had to be considered during the design stage of the project. Most of these concerned the selection of hardware, although this in turn had implications for the system software. At all stages of the project, simplicity was the prime consideration. Complex or elaborate hardware designs had to be discounted for reasons of both cost and time. A sophisticated board could have been built and debugged, but this would have exhausted the time available to complete the project and would have cost far more than the available project budget (£100-200).

This meant that a Multibus-1 board had to be used. Boards were readily available and could be easily installed in the most suitable departmental computers – a Sun-2/120 and Sequent Balance B8000.† The former is a Multibus based machine and the latter has two Multibus cardcages and adapters. An added attraction was that the operating systems on both these computers supported the *mmap()* system call. In addition, another Multibus-1 based machine – a Wicat 150 – was available for hardware testing and device driver development.

It was initially intended that the Multibus board would contain the teletext receiver and decoder chips, using the widely used I^2C interface. Although attractive, this approach presented too many problems and was abandoned. Firstly, there was some doubt if there would be sufficient space on the board for this circuitry, a local processor and its RAM and PROM, plus the necessary components needed to interface to the Multibus. It was also thought that a board like this would have high power requirements which could make cooling a problem. Another consideration was RFI screening. There was a possibility that electromagnetic interference from the host computer could corrupt the TV signal.

After some discussion, it was finally decided to use a decoder based on the Philips E.C.C.T. (European Computer Controlled Teletext) chipset, a simple and cheap device that is commonly used with the B.B.C. microcomputer. It had the advantage of offering a trivial B.B.C. serial port connection to provide an I^2C interface to the decoder. An added attraction was that the decoder could be kept separate from the Multibus board. The board would only require logic for this simple serial interface to connect to the decoder. Another benefit of separating the decoder from the Multibus board was that it simplified the development and testing of the firmware. The decoder could be connected to a B.B.C. microcomputer and tested while the Multibus board was under construction.

One disadvantage of this approach was the low rate at which teletext pages could be acquired – initially just 1 per second. Upgrading the decoder to an 8Kbyte RAM meant that the decoder could grab 4 pages per second. The clear implication of this was that pages would have to be stored somewhere if an acceptible response time and display rate was to be achieved.

Since there are approximately 600 1 Kbyte teletext pages on all four T.V. channels, storing every page would require a total of about 2.5 Mbytes of memory. For obvious reasons, it was decided that these pages would be cached by the UNIX host rather than the Multibus board§.

Another simplifying decision that was made was not to give the board an interrupt capability. The board would therefore not require any interrupt logic, making it easier to install in the host. The device driver could then function without an interrupt handler. This also avoided undue kernel processing – potentially for each teletext frame received at a rate of 25 frames per second (the broadcast standard for PAL television in the U.K.) – and eliminated additional interrupt latency which could have been a potentially significant overhead.

The board was built and consists of four functional sections: the Multibus address logic, an Intel 8051 microcontroller, a dual-port RAM and the Philips I^2C serial interface to the decoder. The dual-port RAM allows the board to present a page of teletext to the host system while the microcontroller fills the other page of RAM with data from the decoder. Once a page is copied from the decoder, the microcontroller simply flips pages. The new page is visible to the UNIX host while the old one can be replaced by another page from the decoder.

† The other Departmental machines were rejected for a number of reasons. Some used complex and/or proprietary buses – e.g. VMEbus, UNIBUS and HP-IB. Others like a Sequent Symmetry could never be taken out of service for device driver development.

§ The imminent arrival of satellite television channels with even more teletext pages meant that a potentially enormous number of pages could need to be cached – certainly more than could be stored on a single Multibus board.

To the UNIX system, the board appears as 2 Kbytes of RAM in the Multibus address space, the RAM containing a complete teletext frame. By arranging this RAM to be aligned on a page boundary, it was hoped to use the *mmap()* system call to allow a user process to access the device directly. In addition to reducing the kernel overheads of copying data, this would also save writing anything more than the most minimal device driver.

It was also decided not to give the Multibus board a set of control/status registers like a conventional Multibus device. This would have added unnecessary complexity to the board. It was simple enough to arrange for the microcontroller to interrupt when the host wrote to the dual-port RAM. It could then inspect a known location in the RAM for any commands (tune to a given frequency, etc) issued by the device driver. Since the Multibus board was essentially a read-only device, this was a very simple and effective (albeit somewhat crude) way of programming the hardware. In retrospect, this proved to be an unwise decision, since it caused unnecessary complexity to the microcontroller firmware.

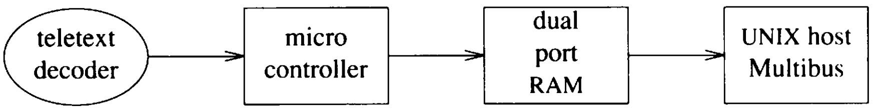


Figure 1: *Schematic Diagram of Teletext Interface*

3.2. System Software

The initial intention was to use a single process which would receive requests for specific pages, fetch those pages from the hardware and then return them to the requesting process. There was a major flaw with this approach.

Teletext pages are transmitted in an order determined by the broadcaster. To all intents and purposes, this order is semi-random although commonly used pages (e.g. index pages) get transmitted more frequently. This meant that it could a long time – perhaps 10 to 30 seconds or so – for a request for a rarely transmitted page to be satisfied. In short, there is no way of knowing in advance how long it will take for a particular page to be transmitted (if at all). Not only is this delay unsatisfactory, it could cause the server to be unresponsive to other requests for long periods of time. This problem would be compounded by the poor page capture rate of the decoder.

The only way to get round this problem was for the host system to cache teletext pages. It was thought that a single daemon could request pages from the hardware, gradually building up a store of pages, and then service client requests from its copies. Sadly, this too had to be abandoned. Such a daemon would eventually hold 2 or 3 Megabytes worth of teletext pages. If this was held in virtual memory, it raised the prospect of a large process waking up at frequent intervals – many times a second – to service incoming requests and to fetch pages from the hardware. It could cause excessive paging by the virtual memory system and/or waste large amounts of host memory. Eventually, it was decided to store the teletext pages in files, one per T.V. channel, and use the teletext page number as an index to the file.

3.2.1. Device Driver and Microcontroller Firmware

The microcontroller firmware is straightforward. It accepts commands from the device driver and controls the operation of the teletext decoder, handling the tasks of controlling the tuning frequency and page selection. As data arrives from the decoder, the microcontroller stores it in a page of the dual-port RAM. Once the page is received, the microcontroller recalculates the checksum and sets some control and status information in the user-defined part of the teletext frame. This allows the daemon to know if a page is corrupt (saving the host from calculating the CRC). When the page is no longer needed, the microcontroller flips the dual port RAM. A new page is now visible to the Multibus address space of the UNIX host.

In the case of a fault in the decoder, the microcontroller sets up the dual port RAM with an explanatory message in text – "I^2C Bus not responding" for example – and an error code that can be passed back to the daemon. Typically, these will happen if the decoder is powered off while the device is in use.

The device driver issues 4 commands to the hardware: request a new page, request a particular page, tune the teletext decoder and reset the hardware. These commands are largely self-explanatory. The first simply asks the microcontroller to fetch any new page that has been captured by the decoder. This is simply a semaphore to the microcontroller to tell it that the device driver is no longer interested in the current page accessible on the Multibus. The second asks the microcontroller to set up the decoder to capture a specific teletext page. It is possible to specify individual pages – "capture page 200" – or wildcard page selection – "capture any page between 100 and 199".

The tuning command allows for coarse and fine tuning. This permits the daemon to tune to a particular frequency – 662.65 MHz for B.B.C. 1 – and to make minor adjustments (50 KHz increments) in light of prevailing reception conditions. It is intended that the daemon will use this command to select a particular teletext service and to adjust the tuning frequency to minimise the proportion of corrupted teletext pages.

The reset command is used at start-up to re-initialise the hardware and to put the microcontroller in a known state.

The device driver also polls the hardware 4 times a second, waking up the daemon if anything interesting has happened – for instance, a new page appearing. This allows the daemon to gather teletext pages on the fly while it waits for a particular page to be transmitted.

One problem with the firmware is its interaction with the device driver. When a command is issued and the Multibus writes to the dual port RAM, the microcontroller is interrupted. It can read the known location in the RAM that has been written to and interpret the data found there as a command. However, it is possible for the microcontroller to do this before the device driver has completed writing the command. (i.e. the controller could interpret old teletext data as a command from the driver.)

Two steps were taken to minimise this problem. The first solution was simple. Since teletext data uses 7-bit characters, driver commands would just set the eighth bit, allowing the microcontroller to easily distinguish a command from old teletext data. However, a corrupted teletext page could contain characters with the eighth bit set and thus confuse the microcontroller. The next solution was to arrange for the microcontroller to delay accessing the RAM for a few Multibus bus cycles. Hopefully, this would give the UNIX host time to completely write the command to the RAM before the microcontroller attempted to read it. If a conventional memory mapped control/status register had been used, this problem would have been avoided.

3.2.2. Teletext Daemon

The daemon has the task of operating the hardware and storing the collected pages in files. At start up, it programs the hardware to tune to a TV signal and then builds up a store of all the teletext pages on that TV channel, copying the pages to a file. The procedure gets repeated for all four channels.

Unlike the microcontroller firmware, the daemon knows about how the TV channels organise their teletext service. This allows for flexibility as and when the TV stations re-organise their teletext services.† For instance, B.B.C. offer pages 100 to 699 while Scottish TV only provide pages 100 to 299. Unless the microcontroller was aware of this, it could ask the decoder to capture a page number that will never be transmitted, hanging the hardware. To prevent this happening, the device driver verifies page number requests from the daemon before passing them to the hardware.

Once the daemon has collected all the pages from all the channels (taking 5 to 10 minutes at present), the daemon then goes into an infinite loop. It then retunes the decoder and requests pages, updating the copies on file as appropriate. The daemon keeps track of the most frequently changing pages and requests them more often so that the teletext database on the UNIX host is kept as up to date as possible.

3.2.3. Client and Server Programs

At the time of writing, specifications for these programs are being drawn up. A simple UDP based protocol will be used to request and send teletext pages. The server will decode an incoming request, fetch the appropriate page(s) from the relevant file and transfer the page to the client. Client programs may choose to recalculate the teletext checksums on each page to ensure the reliability of the received data.

† It is far easier to recompile the daemon than to remove the microcontroller and reprogram it.

3.3. User Interface

Client programs requesting particular teletext pages are passed the complete frame, including control information, checksums and any graphics characters. This permits clients to have full control over the data they receive and how it is displayed. For instance, the checksum can be recomputed and compared to ensure that the datagram arrived intact. Control information such as colour and/or flashing display segments can be interpolated and displayed às well as the available hardware will allow. Judicious use of reduced intensity and/or reverse-video produces reasonable quality displays on a terminals that have a minimal graphics capability such as the Lear-Seigler ADM 11. A simple curses-style interface is provided to permit users to select individual teletext pages or to scan through the pages at an acceptible rate.

At the time of writing, the X-windows interface is under development. This will provide a menu-style interface for the selection of pages.

One interesting point here is date conversion. Each teletext page includes a string indicating the exact time when the page was transmitted. Since the current system caches every teletext pages, the stored date can be much older than that when the page is displayed. The display software therefore has to note the current time and ensure that this is the date which is shown when the teletext page is eventually displayed.

4. Further Problems

One area that has so far been neglected is the issue of copyright. Apparently, the broadcast teletext pages are the copyright of the broadcasting organisations. It is not known if the service described above will be a legal or illegal use of teletext data. This will require further investigation before the facility can be brought into service for the benefit of the local user community.

5. Future Developments

The current board is primitive and offers an adequate but slow performance. A new board should be construoted that is capable of a far higher page capture rate, ideally 25 pages per second on all channels simultaneously. If practical, the new board should also contain a substantial amount of RAM to cache teletext pages. It would also be better for the board to use a conventional memory mapped control/status register instead of writing commands to the device's mapped memory that holds teletext data. A device driver that fully supported *mmap()* would be desirable.

The daemon should be capable of more intelligence in selecting tuning frequencies that offer the best reception of teletext pages. It would also be worthwhile for the server to cache the most heavily requested teletext pages to improve response time.

6. Acknowledgements

Special thanks are due to Philips TMC, of Bishopbriggs near Glasgow who have generously sponsored this project. Their assistance has included supplying components, data sheets and test equipment and access to technical staff and hardware development facilities that would otherwise have been unavailable. The authors particularly express their gratitude to Mr. Suli Yacoob, Head of Software at Philips TMC, Bishopbriggs for his enthusiastic support and encouragement.

7. Bibliography

"Computer Controlled Teletext User Manual", *Philips Components*, 1984.

"Characteristics of Teletext Systems", *Vol. 11, Report 957 C.C.I.R. XV Plenary Session*, Geneva 1982.

"I^{2}C Specification", *Philips Components*, 1985.

Porting Banking Software from VMS to UNIX

Rudolf Koster

InterFace Computer GmbH
Garmischer Str. 4
D-8000 München 2

ABSTRACT

This paper describes the process of porting and partially re-implementing one of the first UNIX software packages of substantial size for the banking sector. The *Deutsche Terminbörse* began operation in January 1990 and consists of networked minicomputers running VMS or AIX, IBM's version of UNIX. During the port to AIX, special care had to be taken to strictly retain functionality and to guarantee fairness to traders.

1. Introduction

In January 1990, Germany's first fully computerized option exchange began operation: the *Deutsche Terminbörse* (DTB). DTB is an enterprise of the major german banks. Members can buy options that enable them to buy or sell a certain number of objects within a certain period of time and according to a determined rate. During the first phase, options on 14 different shares are traded. Each member bank operates at least one so called *User Device*. The User Device is a minicomputer that shows current rates and accepts orders to buy and sell. It is connected to the nation-wide network by so called *Communication Servers* (CS). The CS multiplex connections from User Devices to a central host computer in Frankfurt, where trading takes place.

The network consists of dedicated lines with a bandwidth of 64 KBit/s. The protocol between the Host and the Communication Servers is Digital Equipment's DECnet, and between User Devices and CSs it is TCP/IP. Host and communication servers are Digital Equipment VAXes (the host being actually a VAX cluster) running VMS. In 1987/88, Arthur Andersen Consulting developed the Swiss computerized options and futures exchange, called SOFFEX. This system was entirely based on DEC hardware and software, including the User Devices. DTB commissioned Arthur Andersen to adapt the SOFFEX system to the german exchange, thereby making available IBM hardware as a new platform for the User Device.

After a feasibility study, it was decided to use the IBM 6150, also known as IBM PC RT, as the User Device. This RISC machine is runs AIX 2.2, IBM's current version of UNIX. This is noteworthy because of UNIX's notoriously weak position in the commercial marketplace, and especially in the banking sector, where UNIX generally is not viewed as a mature commercial product. It also shows IBM's commitment to UNIX, which has been much debated recently.

The former User Device software for VMS was written in Cobol and Pascal, and used a lot of VMS specialties, e.g. a non-standard dialect of Pascal. Therefore only most of the Cobol part was retained; the rest was re-implemented in C.

2. User Device Architecture

The User Device is located at the banks and connects to the Frankfurt host via a Communication Server. TCP/IP was chosen as the communication protocol to the CS, simply because it was the only protocol available both on VMS (by a third party product) and AIX (built into the kernel).

The original design provides an ASCII terminal for each trader, the screen of which is divided into several sections to supply the necessary information. The current rates are shown in one section of the screen and are updated continuously, while in a different section the user places his or her orders. To manage this relatively complex screen layout, the VMS version used a screen handling package called *FMS* (Forms Management System). Under AIX, it was implemented using the familiar curses package, which has been extended by IBM to support colour, among other things. To enhance functionality it was decided to replace the ASCII terminals by IBM PS2 Personal Computers running the Server software of the X Window System either on AIX or PC-DOS. They are connected to the User Device over a IBM Token Ring Network and allow each trader to have multiple trading sessions in different windows on the screen.

One set of processes in the User Device handles communication with the Communication Server. Messages vary in length, but are limited to a maximum of 2048 bytes. A simple handshake protocol is employed, so that every message sent to the host gets an acknowledgement. Each message sent is kept in the User Device, until an acknowledgement is received. Every User Device knows the addresses of a fixed number of Communication Servers (usually five). For security reasons, the addresses are patched into the executable of one of the communication processes. If the User Device detects a failure of the CS it is currently connected to, the next CS in its list is chosen to provide communication with the host. Once the new connection is established, all the stored but not yet acknowledged messages in the User Device are retransmitted, assuming that the collapsed CS was not able to transfer all of them correctly. The host handles duplicate messages by simply ignoring them. The communication processes were originally written in Pascal under VMS and have been newly implemented in C under AIX.

Another set of processes in the User Device exists per trader and implements user input and screen output, among other things. Originally a big Cobol program, large parts had to be re-implemented using C and the curses package. To strictly retain the functionality of the VMS version, we had to break up the trading process into three separate ones in order to be able to emulate VMS asynchronous processing. This introduced a substantial load in terms of number of active processes on the User Device. With the introduction of the X Window System and the possibility of having multiple windows on every PS2, the User Device got further loaded with a window manager process per PS2 and a terminal emulator process per open window.

These two sets of processes — one for communication and one for trading — are tied together using most of the UNIX mechanisms for interprocess communication and synchronization. Fortunately, AIX is a very complete mixture of AT&T System V.2 and BSD4.3, so we were able to use the best of both worlds.

3. Inter Process Communication

Because modern versions of UNIX provide many different IPC mechanisms, we had to choose which one to use. According to the benchmarks we took, message queues turned out to be the fastest, but our maximum message size was unfortunately too large for them to handle (2048 bytes). Named pipes had the drawback of small buffer size, so we decided to use sockets. This had the additional benefit of being uniform, since we had to use *stream* sockets for network communication anyway. For local communication, we decided to use *datagram* sockets for several reasons:

- Resemblance to VMS mailboxes
- No overhead for connection establishment when a trader logs in
- Message boundaries are preserved
- Extensible to network environment

With the introduction of PS2s as display stations, the last point became the most important. Because we use sockets to transfer messages from the trading processes to the communication processes, it is easy to move all the trading processes to PS2s running AIX. This takes most of the load off the User Device and results in a fully distributed system that better utilizes the local processing power of the PS2s. However, this has not been implemented in the current version.

All the details of socket-related system calls have been hidden in a set of library functions that resemble the corresponding VMS mailbox services. This was done to aid programmers with little experience in UNIX in transferring the VMS code to AIX. Furthermore, this additional layer of software makes it easier to completely substitute the communication mechanism used; this could be desirable or necessary, if in a future version of AIX a different and possibly more efficient IPC mechanism is introduced.

Whenever two or more processes must have access to the same in-core data, we use shared memory combined with a semaphore. This is done in three different places:

- Both the process reading from the network and the process writing to the network must update the table of saved messages for retransmission
- Several processes must access the table of logged in traders
- All logged in traders must have access to certain basic trading information

A process that wants to access data in shared memory first consults a semaphore to determine if the data is currently in use by a different process. If it is, the process is blocked until the using process frees the semaphore. Additionally, semaphores are used to compensate for a shortcoming of signals: even if you use the Berkeley version of signals (which is more enhanced than System V), you might lose signals if they come fast, since only one signal is stacked. In our implementation of the User Device, one process uses a signal to notify another process if a trader logs off. When, during simulation, several traders were logging off at once, some signals got lost, and — as a result — internal tables were corrupted. So we introduced a Semaphore that is consulted both by the process sending the signal and the signal handler routine. Only if the signal handler has taken the appropriate action, it frees the semaphore, thereby allowing the notifying process to send another signal.

4. Network Communication

As mentioned above, an IBM User Device and a Communication Server communicate over TCP/IP. From within programs, the socket interface on the IBM 6150 and an AST (asynchronous system trap) system call interface to PASCAL on the DEC VAX Communication Server is used. The type of socket we really needed, called *sequenced packet*, was not available on both machines. We had to transmit variable sized messages over the network but didn't want to implement our own sequencing, flow control and retransmission protocol. So we decided to use Stream Sockets. With Stream Sockets (mapping to an underlying TCP connection) you get reliable, in-order and unduplicated delivery of data. Unfortunately, you really get a *stream* of data, meaning no message boundaries are preserved. The process on the reading side of the connection sees whatever happens to be there at the moment it reads. It could be a few bytes of the message, the whole message, or more than one message. You can easily prevent reading more than one message by giving the system call an upper limit of number of characters to read. But the case where you would read only a few bytes is a problem. The solution is to read in a loop until you read enough bytes, and then to assemble the message. Instead, you could *peek* at the message until it is available as a whole, and then read. A waste of CPU time, in both cases.

Hopefully, future versions of UNIX will contain the Sequenced Packet socket type, allowing for reliable and sequenced data transfer with message boundaries preserved.

Another problem we encountered with TCP/IP is that a breakdown of a remote machine is not detected if the connection is idle. As mentioned above, our concept of failing assumes that in the case of a CS breakdown, a different CS is contacted immediately. Unfortunately, a *read* or *write* system call has to occur to detect such a breakdown. The *keepalive timer* offered by the sockets library does not help much in this respect, because it is far too large and is hardcoded into the particular socket implementation.

5. Support of the X Window System

When trading on a traditional character terminal, a trader can view one screen full of trading information. If the trader wants to change certain trading parameters, he or she has to leave the current application and invoke a different one. This is time consuming. With the introduction of PS2 computers as X display stations, the trader can have multiple overlapping windows on the screen, each having different trading parameters. By clicking at a window with the mouse, this window gets the keyboard input focus, while output continues to appear in all the windows. Thus, the trader is supplied with additional information and the ability to change quickly between environments.

The User Device, together with the PS2s, is connected to an IBM Token Ring LAN, the bandwidth of which can be 4 MBit/s or 16 MBits/s. For every PS2 connected, a window manager is running on the 6150 to allow for creating, moving and destroying of windows. Every window opened consists of the standard X Window terminal emulator, running the trading application. The keyboard translation mechanism had to be used extensively, to allow for the desired mapping of function keys to user functions. All the application processes as well as the X clients run on the User Device. This setup allows cheap PC-DOS machines with only the X server running to be used as display stations. Should the need arise, it would be easy to distribute the trading applications plus all the X clients to PS2s running AIX. In this way, it would be possible to connect a much higher number of display stations to a single User Device, because the User Device's role would be reduced to a relay station for host messages, instead of additionally running all the applications as it now does.

To provide a limited form of protection against machine breakdown, several User Devices can be configured to reside at one bank. In this case, each trader on a PS2 chooses the User Device over which to trade. If the User Device breaks down, the user can choose a different one from a window manager menu.

6. Development Environment

The development environment consisted of several IBM 6150s connected by Token Ring and running Distributed Services, IBM's version of an integrated file system. We attempted to create the same logical layout of the development tree as existed on the VMS development machine. Each program resided in a certain directory, where it was tested locally by the programmer. After finishing this local test, a person called the *migration manager* moved the program to a test directory and made an integration test with the rest of the system. If the program didn't pass this test, it was returned to where it came from. If it did, it was moved to the (final) execution directory. In addition, each programmer could have a private copy of global header files. This means that different copies of source and header files could be present at various places in the source tree. For a compile, a specified file had to be searched for in certain places and in a certain order. This is a task that the UNIX *make* utility can't handle very well. A set of shell-scripts was written to gather the required files in a common place and then call make.

AIX provides convenient user interfaces, like menu shells and windowing systems, but these can be used fully only on the system console which has a mouse and graphics capabilities. AIX contains a nicely enhanced version of the *dbx* debugger. It provides windowing support for debugging multi-process applications (only on the console) and was of invaluable importance to our project. Additionally, we used a third party product that provided windowing support for ASCII-Terminals which proved to be very useful during testing.

7. Optimization

The trading programs had been written to a large part in Cobol, which resulted in very large processes (5-6 MByte). We did not use the dynamic linking facility offered by the compiler. Although this would have reduced the code size, run time optimization was our primary goal and it would have been inhibited by this facility. The machines were configured with the maximum amount of main memory possible (16 MByte) and a huge swap space. Fortunately, AIX employs the copy-on-write scheme for a process' data pages. This means that during a *fork* system call, not all the data pages of a process are copied. Instead, individual pages are copied when a data value is altered. Therefore we could afford to fork every trading process twice, in order to emulate asynchronous processing, without incurring much overhead.

Startup time of the Cobol processes was very poor at first. It turned out, that the Cobol compiler allocated most of the declared variables dynamically from the heap, a very expensive operation in UNIX. This problem could be solved by pre-allocating the whole chunk of required memory immediately at the beginning of the (C) main program. This way the following blocks to be allocated are taken from the user's address space and no context switch is necessary. Further performance could be gained by placing the heavily used file systems on a different disk than the swap space. In addition, these filesystems were located near the middle of the physical disks. A big help in determining the right values for a lot of system parameters is the *sar* (system activity report) program. It gives you information about usage of the buffer cache, process table size, paging activity, and just about any other tunable kernel parameter.

8. Limitations of UNIX

UNIX is known as a very good system for software development. In the project described, we encountered a few limitations that complicated matters.

- The project team was large (around 30 people), and all of the programmers were relatively new to UNIX. The biggest problems they had to face were the terseness of the user interface (on ASCII terminals) and the limited help system.
- *Make* was the only software engineering tool we used. It proved to be not flexible enough for the kind of "staged" source tree we wanted. Therefore we ended up with a plethora of complicated and slow shell scripts that automated the compilation process.
- UNIX has no true asynchronous processing, so we had to simulate it with multiple processes communicating with signals. This complicates program structure and makes debugging harder.

- The signal mechanism, although enhanced in BSD versions, is still one of UNIX's weakest points. Only one signal gets stacked. To prevent the possible loss of signals, semaphores and other synchronization mechanism have to be used.
- Although not a limitation of UNIX, TCP/IP has a few weak points that caused problems. Stream connections do not preserve record boundaries and breakdowns of remote machines are not detected on idle connections.

9. Conclusion

Our general approach to porting a large VMS-based software system to UNIX proved to be quite reasonable. Instead of trying to provide the needed services by libraries and retaining the original code to the largest extent possible, we decided to rewrite the central and time critical portions of code in the C language. This lead to a stable system with high performance that can easily be adapted to different network environments.

Is Unix resistant to computer viruses?

Pascal Beyls

BULL
1, rue de Provence
Echirolles
FRANCE
beyls@ec.bull.fr

ABSTRACT

Epidemics always catch people's imagination. It was the case with the plague during the Middle Ages. More recently, computer viruses have appeared, and some of them, like Friday the 13th have attracted considerable media attention. The means of contamination (in cauda venenum), their reproduction and their spread show real similarities with biological viruses. In the same way, they are a scourge for all users.

The distribution of applications in binary format and intensive use of networks speed up the spread of these viruses, aided even further by the ignorance, naivety and incredulity of data processing managers. Talk about viruses and the danger they represent causes amusement, derisive smiles, condescension and even compassion. But it does not just happen to other people.

To fight such an adversary, we have to get to know it. This is why it can be useful to define the virus, describe the inoculation method, explain its workings, its spread and even its mutations.

In this field, the authors show great imagination. As well as simple destruction of files, the viruses poison† users' lives: ping-pong balls on the screen, little gnome closing all the windows on the screen, indelible message on printers, etc.

The user is not entirely defenceless. Each virus can be identified and neutralised using programs known as vaccines and disinfectants. But, as is the rule during epidemics, prevention remains better than a cure. This is particularly true as it is always difficult to get rid of viruses.

Both micros and mainframes are ideal environments for virus proliferation. Nevertheless, there are few examples mentioned where UNIX systems were contaminated. Is UNIX resistant? Do its mechanisms provide it with defences? Or is it simply because it has not been possible to isolate one?

The aim of this paper is to make a brief survey of the situation.

1. Our starting point

During summer 1989, one of our customers using a Unix system, was the victim of a computer piracy incident. As is often the case in France, the hacking was simply carried out by means of a Minitel, a very widely available system in France. A system which enables connections via a PAD, but does not enable the pirate to be identified. This incident did not involve any sophisticated techniques but was merely confined to exploiting some bugs in the system. This computer piracy was immediately stopped by a secure version of Unix. Nevertheless, we were expected by the customer to carry out further investigation on the subject of viruses in Unix systems. Up until our investigation we had not even posed the questions. This paper is a summary of several months work on the subject. It is not expected to give a definitive answer to the question. But there are more advanced studies referenced in the bibliography that cover the subject further. The purpose is to give a general idea concerning the resistance of Unix to viral attacks.

† The word virus comes from the Latin for poison.

2. A word about the jargon

We first must define the terminology of computer viruses. A Trojan Horse is a spoof programs masquerading as a legitimate program, but containing some illegal malicious component. A trojan horse has an immediate action, does not propagate itself and is not self-replicating. A logic bomb is a program containing a function which will be triggered when a particular event occurs. It can be a specific date (such as we saw with the famous Friday the 13th virus), or the presence of a file. A logic bomb is hard to detect because the hosting program continues to run normally.

Well-written viruses contain logic bombs. That gives enough time for the virus to spread widely and it makes it much more difficult to detect the originator. A worm is a self contained program similar to a virus. But worms are not parasites, they do not live off themselves other programs. On the contrary, a worm contains some mechanisms which permit it to move from one computer to another. Networks constitute their favourite playgrounds. Famous worms are the IBM Christmas Tree and Internet Worm. In November 1988, the Internet worm infiltrated a network of about 6,000 computers. It took several days to clean up the network. The best definition of a virus remains the one which was given by Fred Cohen: a computer program that can infect other programs by modifying them to include a possibly evolved copy of itself.

3. A little bit of history

In 1974, the first replicating code was produced (Xerox). Ten years after, Fred Cohen presented the first paper on viruses. This is an important starting date. In 1986, the first virus infection started: it was the Pakistan virus, alias Brain, originated by 2 brothers from Pakistan. In 1988, there were industry responses: new companies were created to help users to get rid of viruses, give advice and offer anti-virus programs, even disinfect whole sites. In the USA, the CERT (Computer Emergency Response Team from the Carnegie Mellon University) plays this role. There were 19 such companies in January 1989 worldwide. In November 1988, the Internet Worm affected about 6,000 computers within a week end. In October 1989, a virus Friday the 13th was given much coverage in the press.

Also, the NIST (National Institute for Standards and Technology) is already aware of the problem and published an excellent guide. In the days following the famous Friday the 13th virus, the NIST, the NCSC (National Computer Security Center responsible to the DoD) as well as the SEI recommended several steps to cut the possible risk of viruses. A study was carried out and sent to the different system manager in large US Corporations.

We must appreciate that more than 300 computer viruses now exist, they are now something that we must live with.

4. Getting started with computer viruses

For people wanting to know exactly what is going on with computer viruses, it is possible to read the netnews *comp.virus* which is very informative. As an example, during November 89, more than 220 news items were posted. That means you can read about 10 articles each working day. The news group comp.virus is a non-digested usenet mail forum, also there is VIRUS-L which is a moderated and digested forum providing relevant contributions, information on accessing anti-virus, document, and back-up archives. Moreover, now and then, lists of anti-viral archive sites are provided which could be of considerable help to an infested site.

Another source of information is reading the computer press. The topic is often described and useful advice is given. In all cases, the most interesting thing is that only viruses running on micro computers are analysed. Seldom, do we read of viruses effecting mainframes. Particularly viruses on Unix are rare, we could say that the problem does not seem to exist under Unix. This is certainly what the netnews would lead us to believe. Looking back at the comp.virus for November 89, out of a total of 220 articles only 10 mentioned Unix, none of which related to actual cases of viruses under Unix. However, as we will see, that does not imply Unix is a special case.

5. The virus mechanism

The principle is always the same: a virus modifies the code of another program. Usually inserts itself at the end of the target program, leaving the bulk of the program code intact. The initial jump instruction in the program is modified into a jump to the virus itself. When the virus is terminated, another jump is made to the start of the original host program code. Malicious viruses sometimes overwrite the program, making it totally unusable.

We can summarise this as follows:

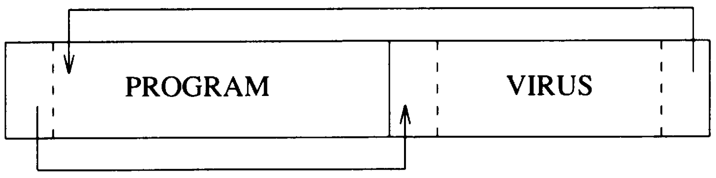


Figure 1: *The virus mechanism*

Most viruses spread their infection to other uninfected programs by reading them, checking they are not already infected (a specific mark is used by the virus) and rewriting them.

6. The spreading

During 1989, in Brazil, Modulo Consultants published a study of viruses covering some 550 micro-computers. It was revealed that most of the virus were distributed by the means of disks. The viruses detected were Brain, Israely, Ping Pong, sUMsDos, Alameda, Lehigh, Madonna, Cookie, Water fall, Mailson.

This report clearly shows the exponential progression of the infection:

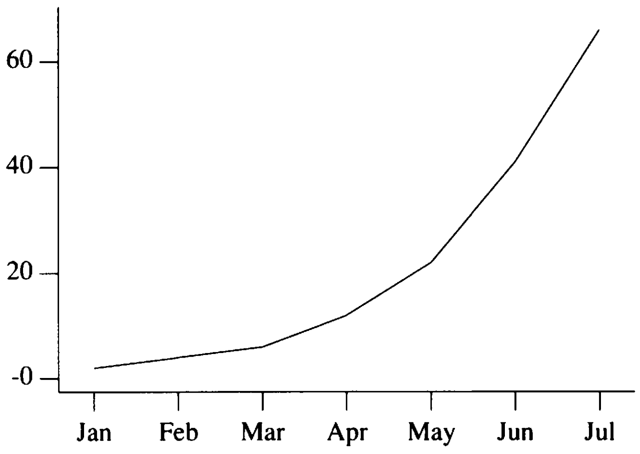


Figure 2: *Number of PCs infected on a total of 550*

In order to avoid a virus execution when calling a program we must check that it has not been modified, i.e. that its integrity is intact. With programs that manipulate sensitive data, it is possible to add an initial protection sequence such as:

```
main(argc, argv)
/*  Check the integrity of this program.  */
if (make_checksum(argv[0]) != CHECKSUM)
{
    printf("Ooops !  Program modified - Must be analysed"!0);
 /* better to add unlink(argv[0]); */
    exit(2);
}
```

Other methods exist that allow software integrity to be kept intact. Notably the relatively recent introduction of smart cards provide an unforgeable means of checking the origin and integrity of software. This is an intelligent means of sealing and checking information. This exists, the smart card contains a small micro-chip, which can take keys and decrypt/verify programs.

7. Writing a virus under Unix

What are the problems in writing a virus that can propagate itself through binary executable files under Unix? This is what we set out to discover. To do this we wrote our own virus, for strictly the purpose of conducting a study on the subject. Obviously we took every possible precaution, in both the writing and the subsequent testing. The danger exists and has occurred in similar experiments (under MS-DOS) that the virus will be propagated out of the control of the experimentor.

In a well controlled environment it is possible to contain the virus (cut all communications links, completely, reinitialise the disks afterwards etc.)

Writing a program which replicates itself is not too difficult. The following is a one line program†:

```
char *f="char *f=%c%c%c; main()
{printf(f,042,f,042,012);}%c";main(){printf(f,042,f,042,012);}
```

Tom Duff gave an excellent example of a virus written in Shell language. This is the example:

```
#!/bin/sh
( for i in * /bin/* /usr/bin/* /u*/*/bin/*
 do if sed 1q $i | grep '^#![ ]*/bin/sh'
 then if grep '^# mark$' $i
 then :
 else trap "rm -f /tmp/x$$" 0 1 2 13 15
 sed 1q $i >/tmp/x$$
 sed '1d
 /^# mark$/q' $0 >>/tmp/x$$
 sed 1d $i >>/tmp/x$$
 cp /tmp/x$$ $i
 fi
 fi
 done
 if ls -l /tmp/x$$ | grep root
 then rm /tmp/gift
 cp /bin/sh /tmp/gift
 chmod 4777 /tmp/gift
 echo gift | mail me@athome
 fi
 rm /tmp/x$$
) >/dev/null 2>/dev/null &
# mark
```

This little piece of code illustrates the dangers perfectly, note how difficult it is to detect that this is a virus, and how simple it is to write. For all types of executable file it is possible to introduce a virus, it is possible to write programs in shell, awk, M4 programs, even in troff macros.

However, we decided to concentrate on the usual executable files: the binary programs.

The problem of writing a virus consists of adding a piece of autonomous executable code into an existing binary executable file. We must create a mechanism so that the virus can be self-replicating, without this it can not propagate itself. The autonomy of the virus code is produced by the use of assembler library calls for open, read, write etc.

† For editing purposes, two lines were needed.

The virus must be able to execute anywhere in memory (position independent code). Therefore, it isn't possible for virus code to use the data or bss sections, that is all local data must be held on the stack. Calls to the assembler library routines can be made by the use of an indirection table which can be kept up to date dynamically and copied into the target file.

The virus must be loaded into memory in order to be executed. It can be held in the executable file after the original data section. The size of this section is held in the executable file header and must be patched in the target file in order that the system will load the virus code.

The virus code, now loaded will be executed when the program starts to run. In our design the virus attempts to propagate itself throughout all COFF files of type 526. It infects three files and then cleans the bss code.

We included a malicious section in our virus, in order that we could see that it was being propagated. This consisted of outputting some thirty line-feeds, which told the user that the virus was present in the program he had just loaded. Once executed the virus returns control to the beginning of the original program.

So what are the conclusions that can be drawn from our experiment. There are a number:

1. The additional execution time is insignificant. A virus is practically undetectable when replicating itself. If the malicious part is a logic bomb, there is no chance of detecting it.
2. The code is some 190 lines of C and some 60 lines of assembler, in conventional programming style. In our example, out of 190 lines of C, 10 of them were the malicious component.

   A virus with a simple malicious component is not a large piece of code. When introduced into the source of another much larger program, it becomes practically undetectable.
3. The time to write the virus was only 1 week for an experienced programr. It requires a very good technical competence, and a good knowledge of C and binary process file structures (COFF). This is not something that the average UNIX user is going to write.

7.1. The Malicious Component

The example described above contained only a relatively harmless malicious component: 30 line feeds were output to the screen before executing the infected program. But even so this is annoying if all programs have been infected in this way.

On the other hand to write good malicious components requires a fertile imagination and the examples from the PC world are numerous and sufficiently imaginative that we could say that they are works of genius. It is regrettable that the authors remain incognito. (It is true that many viruses have only the malicious effects of clearing sections of disks and deleting file). More ingenious, indeed more spectacular are those that disturb the display (Rain fall etc.). A demented user can be sufficiently irritated into distroying his own data when the victim of such an attack.

We have carried out two malicious experiments. The first consisted of systematically sending to the printer, via the lp command, 26 lines of a PostScript program. These lines (given in appendix 1) are a program that causes all subsequently printed pages to have the word DRAFT as a footer.

The only way to stop this, is to turn the power off to the printer, which is hardly a solution. On a printer with this problem the printer is rendered useless.

A second experiment was more amusing. On an X Window system, a small gnome appeared all of a sudden, at the bottom of the display and started to walk off with all the windows, leaving only one window, a blank screen, and a disabled keyboard. The effect is startling and leaves the user totally perplexed! It is worth something to see the unfortunate victim watch this little gnome steal the screens! On the other hand, this is much less funny when he appears regularly at the start of every day. You may think that such a virus represents many X Window program lines. But this is not the case, the power of the Window Manager library is such that it is very easy.

The examples of viruses are numerous and the resulting effects many. There is much scope for diversity, and we must expect many more improbable and imaginative viruses to come.

The biggest danger is logic bombs, that is when code is triggered on a built in date/time, or a built in condition. This type of infection has plenty of time to spread. The clean-up from this type of infection is much more difficult. A virus can be designed such that harmless virus A awaits the arrival of harmeless virus B. But their combination causes a dangerous virus mutation. Another question that one must ask is what happen in the case of an incest?

The only thing that can be said about any virus, is that it is undesirable. The initial discovery is sometimes amusing but after that it is only a nuisance. There is no such think as a Good Virus.

7.2. Propagation

The propagation mechanism, via replication, is one essential component of a virus. But it is perhaps not very well known one.

Once the virus has been detected, it is normally impossible to say from where the virus originated. The most that can be said is that program X is infected. Even this is only possible when the virus started to work immediately. But when the virus carries a logic bomb it become more difficult. A virus that only shows itself according to the season or only from within certain programs, makes the extent of the infection impossible to determine. We should note that the names given to viruses (Lehigh, Madonna, etc) are given by those who discover the viruses (following in the foot steps of astronomers) and not by the authors (who remain hidden in the shadows), always assuming that the discoverer is not also the author.

The propagation is difficult to predetermine. We have already measured the spread of an epidemic on a contaminated system. The transmission from one system to another is equally arbitrary. There are essentially two possible means for contamination to spread from system to system, the exchange of media containing an infected program and the network.

In both cases the user is unaware that the program that he wants to use contains a virus. There is no general means to diagnose a virus. Examing the binary is not really feasible. Even looking at the source is not really easy in a program of any complexity. For example consider trying to find 200-300 lines of virus code in an apparently inoffensive program of 5,000 lines such as Kermit.

The only thing that can be done is to use virus detection programs. But this is not so easy, as new viruses appear every day.

The propagation can be slowed down by the security mechanisms of the system. It is this very reason that PCs are a prime target. With no real protection mechanisms, any program can write into the system tables or the interrupt vectors there is no operating system or hardware protection mechanisms.

On principle in Unix, it is impossible for a normal user (i.e. a virus program) to become super-user. That considerably lessens the risk of a virus spreading far. But we have observed that many systems are not altogether well administered or set up and viruses can take advantages of all the security holes.

The human factor is of great importance. As an example, you send an infected program via e-mail, with a covering note: this benchmark will measure performances of your NFS or your X Window system. As usual in the benchmark environment, you take care of the following advice stating that a benchmark should be run in super-user mode (!). It's in the bag. How many system administrators will be suspicious?

Theoretically, we can say that the propagation of a virus relies upon the human factors rather than flaws in Unix.

8. The Bad News

In the future, the lack of viruses under Unix may change due to two major reasons:

8.1. Binary Software Distribution

Binary program exchange is no longer peculiar to the PC world, until recently Unix only supported source level distribution, because of the numerous different hardware architectures. But the situation has now changed with the introduction of ABIs (Application Binary Interface), for numerous processors. The OSF work on ANDF (Architecture Neutral Distribution Format) proposes the use of an intermediate format. In the future, Unix program distribution will no longer be in source form, and this will lead to an increased risk of viruses under Unix.

8.2. Distributed systems

The move towards distributed processing (formerly only at the file level, now at the computing level) will be an open door for the proliferation of viruses.

In addition, there is a lack of security features in many of these extensions such as NFS. X Windows, especially does not provide any serious security controls. Open computing risks to open the door to Unix viruses. This must be changed in the future.

9. The Good News

Although the evolution of distributed processing increases the risk of viruses spreading. The good news is that security features are now becoming a standard component of Unix, and secure versions of Unix are much less vulnerable from viral attack. For example the OSF/1 to be released September 1990 will offer US DoD B1 security features, towards the end of the year AT&T will also offer security features for the V.4 version of Unix.

How do secure versions of Unix lessen the risk of attack from viruses. Increased access control, such as is offered by Access Control Lists and Mandatory Access Control, lessens the risk of viruses being introduced or propagating through a system. Also many versions of secure Unix have built in configuration control features, which controls which executable binaries may be present on the system, and prevents them from unauthorised tampering.

10. Our conclusions

We found that it is quite easy to write a virus under Unix. Moreover, it constitutes an excellent exercise to increase an understanding of this problem. The difficulty of replication is the same as for the PC world.

However, Unix has natural defences, which give it a better immunity, due to its protection mechanisms and access control, this prevents viruses from spreading rapidly and reduces the danger from the malicious component.

Viruses spread slowly under Unix with a greater risk of detection compared to PCs. In addition, an epidemic can exist only by using security holes, such as bad installation, lack of protections, misuses of super-user mode.

End users, being careful with security, using only reliable software bought from reputable software dealers, can rest easy.

This paper does not aim to give all the advice, but one piece of advice is: people wanting protection from viruses must read the bibliography. We can especially recommend the book issued by NIST. It costs only $5 and it provides a comprehensive set of advice. Another valuable document is the proceedings of a two day conference held in Milano (Italy) 7-8 February 90: I Virus del calcolatore†.

11. Acknowledgements

Without the support, advice, and consultation given by Gerald Krummeck, Stephen Hinde, and Philippe Garrigues who wrote the Unix virus, this paper would not have existed. We would like to express our gratitude to all the anonymous people who have helped us, we mean all the writers of the comp.virus newsgroup.

Appendix 1 — A PostScript malicious program

```
%Prelude to show a draft string on every page.
(PRELIMINARY)
/DRAFTDIC 10 dict def
DRAFTDIC begin
/DRAFTSTRING exch def
/bd /Helvetica-Bold findfont def
/od bd maxlength 1 add dict def
bd {exch dup /FID ne {exch od 3 1 roll put} {pop pop} ifelse} forall
od /FontName /Outline0 put od /PaintType 2 put od /StrokeWidth 0 put
/Outline0 od definefont pop
/DRAFT { gsave
initmatrix
/Outline0 findfont setfont
DRAFTSTRING dup stringwidth pop 8.875 exch div dup 72 mul dup scale
52.3 rotate 2.5 exch div -.35 translate
0 0 moveto show
grestore } def
/oldshow /showpage load def
/oldcopy /copypage load def
```

† aThe papers are in English.

```
end
/showpage { DRAFTDICT begin DRAFT oldshow er.d } def
/copypage { DRAFTDICT begin DRAFT oldcopy er.d } def
% End of draft prelude

% START OF NORMAL LINES
/Times-Roman findfont 14 scalefont setfont
200 200 moveto
show showpage
```

Bibliography

Burger R.: Virus la maladie des ordinateurs
Editions Micro Application, 1989 (French)

Cohen, F.: Computer Viruses.
PhD Thesis, University of Southern California, 1985 (English)

Deloitte, Haskins, and Sells: Computer Viruses.
Proceedings of an Invitational Symposium, 1989 (English)

Dierstein, R.: Computerviren.
KES-Zeitschrift fr Kommunikations- und EDV- Sicherheit 6, 1985 (German)

Duff T.: Experience with viruses on Unix systems
Computing Systems, Vol 2 No 2, 1989 (English)

Dvorak, J.: Virus Wars: A Serious Waring.
PC Magazine; Feb 29, 1988 (English)

Gleiβner, W., Grimm, R., Herda, S., Isselhorst, H.: Manipulationen in Rechnernund Netzen
Addison-Wesley, 1989 (German)

Highland, H. J.: Computer Viruses and Sudden Death.
Computers and security, 6, 1987 (English)

Highland, H. J.: An Overview of 18 Virus Protection Products.
Computers and security, 7, 1988 (English)

McAfee J.: The virus cure.
Datamation, Feb. 1989 (English)

Muβtopf, G.: Trojanische Pferde, Viren und Wrmer.
Per-Comp-Verlag, 1989 (German)

NIST: Computer Viruses and Related Threats: A Management Guide.
NIST, 1989 (English)

Parker, T.: Public domain software review: Trojans revisited, CROBOTS and ATC.
Computer Language, 1987 (English)

Proceedings of the 7th National Computer Security Conference.
Gaithersburg, MD, 1984 (English)

Proceedings on I Virus del Calculatore.
Milano 7-8 Feb. 1990 (English)

Spafford, E. H.: The Internet Worm Program: An Analysis.
Purdue Technical Report CSD-Tr-823, 1988 (English)

Ugon M.: La carte microcalculateur, un antivirus informatique.
Telecom, 1989 (French)

White S. and Chess D.: Coping with Computer Viruses and Related Problems.
IBM Research Report RC 14405 (#64367), 1989 (English)

The Administerability of UNIX Systems

Norbert Ondra

Department of Computer Science
University of Klagenfurt
Austria

ABSTRACT

Today, UNIX enjoys a wide-spread use all over the university world, both in research and education areas. The main reasons for this are the high degree of availability, portability, compatibility and its tailorability towards the individual needs of specific application areas. One important issue, however, is often left out of consideration when talking about the virtues of UNIX, namely administerability.

While this issue might be a minor concern in small experimental environments, it becomes most relevant in larger and more complex network environments, in particular when reliable production operability must be safely and easily provided. Unfortunately, UNIX systems show substantial shortcomings in both enabling, facilitating and supporting system administration, thus handicapping the goals of system availability, security and integrity to be achieved. As will be shown in typical examples, the deficiencies cover the whole range from conceptual issues through insufficient tools up to incompatibilities between UNIX systems, making system administration a tough and time-consuming job.

Hence, bad administerability compromises the requirements of production environments, e.g. in commercial or business areas, where those main goals must be easily attainable with negligible effort. As a consequence, it will turn out to be necessary to significantly improve the administerability of UNIX systems through enhanced concepts currently missing even at kernel level and through providing a standard toolset for system administration routine.

In the opinion of the author, this is a serious challenge for the UNIX manufacturers' community to come up with really administerable UNICes which comply with the still evolving standards and all the requirements of system administration in complex, networked (production) environments, where administerability will become a major decision making factor.

1. Introduction

Today, UNIX has made its way into many scientific, educational and technical areas at universities and research organizations all over the world and is now beginning to spread beyond such fields, e.g. into areas like business administration. There are plenty of good reasons why that should be the case. One immediately comes to think of UNIX's major benefits in availability, portability, nowadays compatibility, tailorability, adaptability, and many more -bilities. Among those properties, there is an outstanding one, likely to be neglected many times, but most important and crucial for the proper operation of any (UNIX) system and, to an even higher degree, for whole networks of such systems. This is *system administerability.*

While adminsterability might not matter very much in small experimental environments where you can (and have to) flexibly tailor the systems to your own needs by improvisation rather than routine, the question becomes really urgent in complex, networked environments where it is crucial that looking after proper operability and availability of all the systems with keeping track of who is allowed to do what on which machines be carried out in an easy, reliable and effective way. That is even more true with application areas where (UNIX) systems are being used for production work (business or not), which by definition have high demands towards ease and reliability in performing system administration to achieve security and integrity of systems.

One can now ask the question how administerable UNIX is. I shall try to sketch a satisfactory answer in the rest of this article concentrating on the specific requirements of our university research and CS education environment. Most of the practical experience has been derived from problems (and related solutions) in the daily routine of administering a variety of UNIX systems at our university.

Chapter 2 briefly describes typical goals and tasks of system administration in general and under UNIX in particular. In the main part of this article, chapter 3, I shall concentrate upon missing or insufficient concepts at kernel level, making system administration a tough job requiring enormous effort, discipline and continuous hands-on the system. All of the shortcomings became apparent through actual problems in the daily administration routine. Workarounds and/or real solutions to the problems are suggested.

The usability of UNIX tools for system administration is discussed in chapter 4. Finally, chapter 5 brings some considerations about system administration compatibility troubles arising from different versions of UNIX, especially the two main streams System V and BSD(4.2). Concluding with chapter 6, I shall summarize the results and give an answer to the question of administerability. Consequently, I shall derive some requirements for improving the administerability of UNIX.

2. System Administration Objectives and Tasks

In general, system administration, serves the overall purpose of making the resources of a computer system or whole networks of systems available to a variety of users with heterogeneous profiles of requests. Typically, that must always be done with another major objective in mind, namely establishing and maintaining the security and integrity of systems. By means of adequate functionality as provided by the underlying operating system, individual users or whole groups of users are granted certain privileges to access and manipulate system resources, with the security of the whole system being preserved, including each user's data privacy. Various tasks must be performed by system administration in order to realize its goals.

In the following, most considerations reflect the specifics of our environment consisting of a network of multi-vendor UNIX systems most of which are used for three principally differing purposes: teaching CS students, research work and department administration tasks.

2.1. Environmental conditions

Both a System V *compatible* and BSD(4.2) derivates are present (see [HPU88a, ULT86a] and [SUN86a]), and all communicate to each other via an Ethernet network running the TCP/IP protocols and the usual networking software based on those. In addition, NFS is being used for providing virtual network discs. Several categories of users – with their largely differing degree of education, CS knowledge (and, hence, demands) – must be simultaneously served on most of the machines: scientific staff, secretarial personnel, project groups, students (diplomas and a considerable number of labs). Only some workstations are solely dedicated to research work of department staff members.

2.2. System administration objectives

Most of the following objectives are relevant in any multi-user multi-tasking environment, in particular in the one under consideration. The prime goals of system administration can be summarized as follows:

2.2.1. Availability of adequate working/programming environments

According to the specific needs, experience and educational background of the individual users, several different kinds of working environments must be provided. We basically distinguish between

- full environments with access to all resources that are needed,
- restricted environments with specific constraints on both the accessibility of (software) resources and the set of available commands, and
- application dependent environments each confined to running one particular application only.

Any user environment needs to access and utilize the right operating system services and utilities and the right software resources (like compilers, desk top publishing, mailing, etc.), so *selective accessibility* must be given to individual users by exploiting the concepts and mechanisms of the underlying OS. Quite the same holds for other resources, like (remote network) printer spoolers, additional (remote network) disk drives, tape units, etc.

2.2.2. System security and integrity

Any part of the OS itself (kernel, configuration files, commands, utilities, devices, etc.) and the application softwares must be protected [Pet88a] against both accidental and malevolent manipulation. This is called system data security. Likewise, data security (integrity and privacy) of the individual users' environments must be established and maintained by enforcing appropriate access control mechanisms on each user by the OS. In both cases, establishing and maintaining the corresponding protection domains (UID's/GID's) [Pet88] and defining the right access capabilities on resources belong to the main duties of system administration.

Since both the user environment area and the UNIX OS itself with all the software reside on the same medium, the UNIX file system, obeying the same uniform scheme for *protection* and *granting access*, the two conflicting objectives of selective resource accessibility while providing utmost security cannot be viewed separately but rather have to be treated simultaneously, in a complementary fashion.

Physical data security is considered to be important but not first priority in the given environment. Only a few applications (secretary DTP and the lecture/exam/dates-schedule data base) are regarded as production, and the corresponding machines are backed-up on a periodical basis.

2.2.3. System availability and performance

For a computer system to be really usable, it should show some minimum degree of availability. User requests must be served within acceptable time, the overall operability and performance must not suffer from single users' excessive resource consumption, notably in terms of cpu time and disc space. In order to provide a certain availability of resources, constant monitoring of users' activities and resource utilization is unavoidable, if only to at least notice when system resources are running short so that appropriate action can then be taken. In certain situations, even OS kernel customization might be necessary, such as tuning parameters to increase system performance.

2.2.4. Support of routine work

In the daily routine of working on a computer system there are always some frequently recurring tasks which are not sufficiently supported by the OS (or not supported at all). In this situation, system administration should provide solutions for facilitating those routine tasks, automating processes as far as possible. This may include parts of otherwise system administration tasks being delegated to ordinary users' responsibility.

2.3. System administration tasks under UNIX

In performing a certain system administration task one always must regard several goals simultaneously, e.g. selectively providing system resources **and** maintaining system security. The enumeration below contains the most important tasks actually occuring in the given university environment.

- user and working environment administration: installing (or removing) users and granting (or withdrawing) privileges in terms of who is allowed to access or manipulate which (software) resources and data. Group memberships and the "group access list" mechanism are heavily used to provide selective access to software and other resources. Typically, any installed user gets equipped with an adequate shell environment. System administration provides reasonable default shell setups in startup files like ".login", ".profile" etc. and appropriate commands for restricted environments.
- **administering software:** consists of installing software on top of a UNIX system and customizing it to the given system's environment. Examples are programming language processors, data base applications, text processors like "LaTEX", systems for electronic submission of assignments, etc. Appropriate access rights must be defined on the directories and files which hold the software such that users can access the right softwares. With the objective of system security in mind, the crucial task [Fox98a] is choosing the access rights on directories and files as restrictive as possible in order to safeguard the software or critical parts against inspection or manipulation by users.
- **operating system administration:** comprises all tasks of customizing and configuring the operation of UNIX itself according to functional and performance requirements. Typical activities are the tuning of kernel parameters (such as the system's open file limit), setting up tty lines, mounting additional discs, providing enough swap space, and, most important, automating routine activities that usually take place at system boot time and shutdown time. That includes startup and shutdown of whole subsystems like data bases and, more generally, any system with server daemons in the background. Furthermore, periodical maintenance of physical data security must be provided by

backing up file stores (and eventually restoring them).

- **providing utilities to facilitate routine tasks:** sometimes routine tasks carried out by system administration or ordinary users are not properly supported by tools. A typical example is performing system shutdown through a dedicated user *sysstop*, say, in order to enable the controlled usage of *reboot* to authorized *shutdown*-users. Another routine activity not really supported in the standard toolset is to *killall* processes of a particular user when interaction on his or her terminal got stuck (due to terminal setup problems, for example).
- **monitoring and controlling resource utilization:** serves the goal of system availability. By constantly monitoring user and system resource utilization and activities an unexpected exhaustion of resources, especially disc space, shall be avoided. Likewise, dynamically rescheduling time-consuming processes by controlling the processes' priorities contributes to more fairness in cpu usage. For both disc space monitoring and process activities supervision minimal solutions exist in BSD UNIX systems, none exist in standard System V UNICes.
- **administering networking services:** is just mentioned for completeness. Networking facilities, like file transfer, remote login, etc. must be administered mainly by providing and subsequently enabling or disabling the corresponding services between any two UNIX hosts in the network.

2.4. The role of system administerability

System administerability can be conceived as the ease, safety and reliability that OS mechanisms, concepts and utilities can be applied with in order to carry out system administration tasks to achieve the goals outlined in 2.2. While this feature might be an issue of minor concern in small experimental environments that can be easily supervised and can be administered in an improvising fashion, it soon gains utmost relevance in more complex (networking) environments where it is no longer obvious who is able and allowed to do what on which machine of the network. Especially in (non-technical) production environments a sophisticated and efficient system administration routine of providing access to computer resources while preserving systems' security is an indispensable requirement.

Since administerability directly impacts the quality of actual system administration activities taking place on behalf of the competing goals of resource accessibility and system security, it must not be neglected even in university environments. Bad administerability normally leads to administration carried out (if at all) sporadically and in an improvisational manner, which in turn leads to insecure and unreliable systems that nobody any longer understands, *administrators* included. This is definitely undesirable.

We now can investigate the question how well UNIX suits the requirements of system administerability, that is how easy it is to safely and reliably carry out the tasks described in 2.3. In particular, kernel mechanisms and concepts and existing software tools will be examined.

3. Deficiencies in Unix Concepts

This section elaborates on conceptual deficiencies of UNIX systems as implemented in the kernel. Some are due to the absence of specific concepts the kernel is lacking altogether, some concern existing concepts that do not suffice for the system security goals to be attained. I will give typical examples as can be found in the daily system administration routine and work out the conceptual shortcomings and pitfalls. In all cases, workarounds (if possible), existing solutions or proposals for solutions are presented.

3.1. Deletability of files

Take, for example, some multi-user DTP software on a standard UNIX System V, consisting of various executables and a lot of data files neatly arranged in subdirectories. In the particular case, the software cannot even be write-protected against manipulation because of the following: upon invocation, first some user status file is created in the base directory of the executables. Since the process executes under the invoker's user ID (the SUID mechanism cannot be used for functional reasons!), it is required that any DTP user be granted write-permission on the directory (for creating/removing those status files) which allows him to remove all the DTP binaries as well. Thus, a concept is needed to delete-protect individual files without disabling the deletion (and hence creation) of entries altogether.

3.1.1. The demand for a delete permission

To have *delete permission* on a file means being allowed to remove it. Delete permission primarily is a property of the file, not its directory as in standard UNIX. That concept (which is not new at all, see the VMS operating system, [VMS90a] permits to grant or withdraw another file access right just like the *read, write* or *execute* permission UNIX already knows. For solving the above problem, we could then just delete-protect the binaries while creation and deletion of other entries in the directory would still be possible. It is important to realize that the above problem is unsolvable within the standard protection scheme of UNIX (unless major changes to the DTP software take place).

Besides, the above example is no singular problem, for it is perfectly legitimate to have a directory, say, where different student users can electronically hand in their programmes without being able to remove each others' submissions. Again, the submitted files must be delete-protected while the directory itself allows the creation (and hence deletion) of entries. Generally speaking, the delete permission concept is necessary for helping administration to establish system security.

3.1.2. Introducing a delete permission

The only true solution is to add the concept of a delete permission to the existing UNIX permission scheme, which goes beyond existing UNIX standards like [XVS90a], [POS90a] or [SVI86a], all of which agreeing on the successful and simple *rwx* scheme.

As a second best solution that at least remains syntactically consistent with the *rwx* scheme, directory permissions and file permissions could be used *in combination* to obtain delete protection in the following way: for a file to be deletable it is required that the file have its write permission enabled and, like before, that its directory be writable. That is reasonable because destroying the contents of a file by rewriting it to zero length is of about the same quality as deleting it altogether, and deletion of a directory entry cannot just be a matter of the directory alone. Conversely, write-protected files are undeletable.

Both solutions, however, share a common disadvantage concerning current UNIX standards and implementations. Of course, the new concept must be reflected by extended kernel functionality to enforce delete permission at kernel level. So, portions of the UNIX kernel would have to be rewritten. The same holds for all system calls affected and most utilities and software built upon those calls. Finally, UNIX standards, like the [XBS90] of X/Open, see [XPG90a], would have to be amended accordingly.

3.2. Public readable system files

Several UNIX system files such as *passwd, group* or *logingroup* that contain important information about user and group identitifications, passwords, etc. are protected against modification through users, yet *public readable*. Any user of the system, once having gained access, can inspect the password file, say, learning whatever he wants about users, groups, login shells, home directories, non-existent passwords, and the like. Malicious users thus are implicitly *invited* to break the data privacy of other users and possibly corrupt their data integrity.

Therefore, public readable system files like *passwd* or *group* constitute a substantial security threat that must be overcome in some way.

3.2.1. Protecting the password file

Surely, the best solution possible is to abandon public readability altogether by making the file *invisible* to ordinary users.

In a first approach, we could just read-protect the password file from other users but the super-user. Regrettably, many ordinary UNIX commands like like *ls, chmod, chown, groups*, etc. explicitly inspect the password file for alphanumerical UID information (and the *group* file for related information on the group's name), so just making those files unreadable would not work. Actually, that is all that most commands need to learn from the password file because any requested file accesses in the UNIX file system are treated according to the requesting process's effective user and group ID's (or several such group memberships in case of group access lists) and the files' access permissions. So, the *passwd* information need not even exist by that time.

One way to overcome the problem is to factor out the crucial information of the password file (encrypted password, home directory and login shell) into a *hidden password file*, which is absolutely inaccessible to any user but the super-user. The harmless information (numerical UID and GID, user name), however, is retained in the original password file for compatibility with utility programmes wanting to extract alphanumerical UID's. The really sensitive information about password, login directory and login shell is evaluated once at login time by the *login* command which must execute in super-user mode most of the time anyway; so it could as well inspect the hidden password file in that mode. Other commands like *passwd* or *su* which inspect and/or alter crucial password file information are also run under super-user privileges (root-owned SUID-commands), which again permits them to access the hidden password file as needed.

Adopting the solution requires no changes in kernel functionality but massive changes in all subroutines, commands and in every software accessing and using those system files. Again, the UNIX standard ([XPG90] and others) get heavily affected because the suggested password file concept conflicts with the standards' stringent conventions concerning name and location of the one password file and the field structure of its entries.

There already exists a UNIX system [Com89a] that implements password file protection in a slightly different way. The hidden password file is just a read-protected copy of the original located in some secure system directory. The original password file is still public readable and contains all the usual information, save for the encrypted passwords which are just blanked out through asterisks. Any process requesting a password must reference the hidden file. That solution at least protects the password information, while home directory and login shell information are still globally accessible.

3.2.2. Protecting other system files

There are still some more security breaches with system files concerning the ARPA/Berkeley network software on TCP/IP. The file *hosts*, for example, must have general read permission for network services to be able to extract the internet address associated with a given host's logical name. The same holds for file *services* determining the network services available. Without going into details, I just want to remark that, in principle, public readibility of those two files on all of the network nodes reveals the connectivity within the network, hence facilitates potential break-ins from one node into another. That problem is not solvable, I believe, without massive changes to the network software, which is unlikely to happen.

3.3. The omnipotent role of the super-user

Since every UNIX system centers around one unique super-user *root* with all permissions to do anything feasible in UNIX at all, any kind of system administration principally tends to be performed under that user *root*. Unrestricted use of super-user privileges, however, is a potential threat for system security and integrity, because any requests are just carried out as far as possible, without any messages or warnings being issued. In addition, systems end up with being administered by too many people, all of whom acting as super-users, for convenience rather than necessity in many cases. Thus, it is only desirable and recommendable to use root privileges as rarely as possible, even for system administration purposes.

As a reasonable solution one might imagine a dedicated system administrator user equipped with a minimal set of restricted super-user permissions necessary to safely carry out most routine tasks, such as user administration (with adding, modifying and deleting of user information in the password/group files), adding and removing of entries in *checklist* (or for BSD systems in *fstab*), doing *fsck*, shutting down the system, performing system backups and many more. All of these tasks, however, require to be carried out under root privileges, in most current UNIX systems.

3.3.1. A restricted super-user workaround

The concept is fairly simple: just provide a set of root-owned binary commands sufficient to carry out most administration tasks and make them SUID-executable. Then install some administration user *sysadmin*, say, and make him a member of the group all those commands belong to according to their GID.

While that method works perfectly for most binary utility programmes as delivered with a UNIX system, it fails with utilities realized as shell scripts. Here, the SUID trick cannot be applied directly but requires binary SUID commands which in turn invoke the shell in order to have the scripts interpreted. An analogous problem exists with binaries which internally check for *real* user id of *root*, also handicapping the SUID mechanism. (In some BSD4.2 UNIX systems important commands like *reboot* are implemented this way.) Again, a binary invocation programme with the SUID bit set will help. This time, however, the real user ID of the process must change to that of *root* prior to executing the original command.

Unfortunately, all these workarounds do not automatically exist by themselves, when the UNIX system is delivered. No, they have to be written (by some poor fellow of system administrator) which is a tedious, time-consuming and troublesome task not facilitated by the operating system. Moreover, it leads to an immense number of similar, yet incompatible solutions provided by system gurus at many UNIX sites throughout the UNIX user community.

3.3.2. A concept for restricted super-users

The only sound solution to provide system administration under restricted super-user privileges is to conceptually make it part of UNIX. I shall try to propose a powerful yet simple scheme:

Consider a system administrator group *sysadmin*, say, which is unchangeably determined a priori as a solid part of the system kernel, like the root user concept. Any member of that group would then be granted all privileges needed to do restricted system administration (in particular all the kernel services needed for it). Default member of that group is user *sysadmin*. The respective capabilities which can be selectively assigned or withdrawn by only the super-user comprise, among others, the use of system calls like *mount*, *chown*, *reboot*, etc., use of commands otherwise reserved for the super-user only, and access/manipulation rights on protected system files. In particular, it no longer matters whether commands are realized as binary executables or as shell scripts. Only the super user *root* would be permitted to grant and renounce memberships to that sysadmin group. Consequently, such a concept could be extended to all groups (or even individual users) regarding arbitrary capabilities on resources.

The introduction of a privileged system administration group as part of the kernel makes it possible to easily perform system administration tasks under restricted super-user privileges. Hence, the administerability of UNIX is greatly improved.

There already exists a system [HPU88] where some minimal means of subsetting super-user privileges to *privileged groups* is provided. Those privileges, however, are confined to only a few kernel capabilities, such as the *chown* system call or the setting of real-time priorities for processes. See also 3.6.

3.3.3. Administering subsystems

To be fair one must admit that there do exist privileged users for administering particular subsystems in UNIX. For example, the printer spooler subsystem in System V UNIX, consisting of various spool directories, other files and related commands, is owned by a special user *lp*. Hence, that *lp* user can be used for administrating the printer spooler system: installing, adding and removing printers, queue-handling, etc. That is also reflected in a handful of commands.

3.4. Underprivileged SUID commands

Consider again the *lp* subsystem. The *lp* command used for submitting files to the printer spooler system is owned by *lp* and has the SUID bit set such that it can manipulate the *lp*-protected spooling areas on behalf of requestors. Therefore, when invoked by some user, the *lp*-command is run under UID *lp* and gets read access to files destined for being printed according to the normal permission scheme. That means that you either have to allow public read access to whatever you want to get printed, or have to use the normal state-of-the-*art* workaround to produce the data to be printed on standard output by using *cat*, say, and pipe it into the *lp* command.

Of course, both these methods are undesirable: the first one would seriously deteriorate the overall security and data privacy of the system, because, for convenience, users would tend to keep files public readable and directories public accessible. The second is just preposterous if propagated as the standard *method* to do print file spooling on the threshold of the 21st century.

3.4.1. The concept of multiple effective user ID's

The *lp*-command could easily access any print files of the requesting user if it were just run under the requestor's effective UID. Without the SUID mechanism such would be the case, by default. So, the *lp*-command just requires both user ID's to be effective for accomplishing its purpose. The situation is quite analogous to the *group access lists* concept which permits a user to be an effective member of other groups than the one which he belongs to by default (GID in password file!).

So, what is needed is to allow processes to have several effective user ID's at the same time, which represents a substantial extension to UNIX, conceptually. That new concept of multiple effective user ID's currently missing in UNIX would definitely improve the usability of commands such as *lp*, thus implicitly increasing system security and, ultimately, administerability.

3.5. Monitoring of disc storage resources

Have you ever stood in front of your system console terminal panicking, while watching a never ending stream of error messages all telling you that the *file system is full?* In such a situation, no ordinary user can write to the disc any more. Typically, things like saving the editor buffer with the latest data changes onto discs could and normally would fail. Loss of data and incomplete, hence potentially inconsistent data are the consequence.

Since, in System V UNIX, any user is principally permitted to allocate as much disc space as needed, within the limitations of available disc capacity, such an undesirable situation can come about even accidentally. It is a troublesome and tedious task to make a file system which has got full available again, so, for the sake of system security and availability, such a situation must be prohibited from occurring at any rate.

3.5.1. Limiting disc storage usage

Principally, in UNIX System V, there is no real way to limit the amount of disc space on a per user basis. Although the system call *ulimit* can be used to limit the size of files which are dynamically written by processes, there is no way to limit the amount of data a user statically holds at any given point of time on the disc as permanently allocated.

What would be needed is something like the VMS quota concept [VMS90]. Every user is granted a maximum number of disc blocks to be consumed, with the possibility of eventually exceeding that slightly, for a certain period of time. Any allocation beyond that quota is denied by the kernel. A similar yet less sophisticated quota concept is realized in BSD4.2 UNIX systems [ULT86], [SUN88]. When introducing disc quota monitoring, one can think of two principal ways of doing it: the first follows the VMS (or the BSD4.2) example by integrating some disc quota monitoring **into** the ordinary UNIX file system interface.

The second is a humble workaround that puts ex post monitoring on top of a running UNIX system. This alternative cannot prevent the file system from eventually getting full, but should nevertheless be realized in existing systems anyway. It serves to keep track of disc utilization by running some monitor daemon which periodically traverses the whole disc store computing the current disc usage per user. If users are reported to have exceeded some disc quota, appropriate actions can be taken by system administration.

In a mere research environment one might be able to comfortably live without disc usage monitoring (which would be delegated to the discipline of each individual user). But with more complex environments, system availability must be guaranteed, therefore file systems that get full all of a sudden cannot be afforded. So, in such environments some disc quota concept is an indispensable necessity. UNIX System V certainly will have to catch up in this respect.

3.6. Controlling process scheduling

All that is said in this section again refers to standard UNIX System V. As in the case of disc usage, the other critical resource, CPU time, must be shared among all the activities according to some fair scheduling scheme. Traditionally, UNIX scheduling is mainly oriented towards dialogue-intensive, interactive processes in the first place [Bac87a] which favours background tasks being granted interactive priority. The only way to lower the priority of a background process is to to submit it with a *nice* priority which is once set at submission time. As soon as the process begins its execution with normal priority, all one can do is to stop the process and restart it with lower priority. If, however, too much work has already been done or the results of execution would not be reproducible, nothing can be done but patiently awaiting the process's termination.

What one wishes in such a situation is some way of dynamically rescheduling (background) processes. This concept does not exist in standard UNIX System V, yet there is some mechanism for dynamically *renicing* process priorities at run time, in BSD(4.2) systems. There is yet another UNIX system permitting real-time priorities [HPU88] to be set on processes or altered even at run time.

As a minimal requirement, I believe, it should be possible to increase and decrease the priority of processes at run time which means the implicit facility of influencing process scheduling. Without that facility, any UNIX production environment is potentially subject to situations where high-priority background jobs prevent any other reasonable work. No system administrating can provide for reliable production work in that case. So, lack of dynamical setting of process priorities definitely decreases the administerability of (UNIX) systems.

3.7. Summary: Overcoming the conceptual deficiencies

Several shortcomings have been recognized as existing within the conceptual architecture of standard UNIX System V. Most of them impact the adminsterability of UNIX in a way such that sound solutions are definitely asked for should not system security badly suffer.

Without a delete permission concept, disc quota monitoring and dynamic rescheduling of processes it is virtually **impossible** to guarantee system security and availability. The visibility of system files and the existence of underprivileged commands like *lp* are a **potential threat** to system security. Finally, the missing concept of a restricted super-user leads to frequent abuse of root permissions where not really necessary, again a potential security risk. The ex post introduction of a restricted super-user concept is not easy to accomplish and can only be done with considerable effort, especially due to lack of appropriate provisions as made by UNIX manufacturers. Together, all the deficiencies significantly decrease system administerability.

For some of these missing concepts and related problems, system administration workarounds have been presented that could be implemented on top of existing UNIX systems. The majority of the problems described above, however, constitute real pitfalls for system security which can only be overcome by introducing new or enhanced concepts for some of which I have made concrete suggestions. All of the corresponding changes to UNIX System V require new kernel functionality, so various parts of the kernel and related system calls must be reimplemented. In some cases, new system calls become necessary. Moreover, all commands and softwares depending on the respective calls must be amended. The UNIX standards like those by X/Open (as described in [XPG90]), IEEE P1003 [POS90] or by AT&T [SVI86] would have to be adapted to account for the conceptual changes.

4. Tools for Unix System Administration

Apart from adequate concepts as provided by kernel functionality, system administration naturally lives on the availability of proper support through tools in terms of single utilities or whole system administration software packages. We will have a brief look at the usability of UNIX tools for system administration. First, the most important system administration routine tasks are sketched out, being candidates for support through adequate tools. For clarity, I will divide those tasks into three categories:

- Typical administration duties that are absolute routine work and preferably should be carried out by a restricted super-user are, among others, user and working environment administration, software and resource administration, system startup and shutdown administration and periodic maintenance like *fsck* and "backup/restore".

  For lack of a restricted super-user scheme in existing UNIX systems, all these tasks must be carried out under real root user ID.

- Frequently requested administration tasks that could be delegated to the individual users' responsibility by providing suitable tools are, for example, soft-resetting a particular printer system that has got stuck through a paper jam, killing all one's own processes (in case the terminal interaction has got stuck somehow) and individual startup and shutdown of specific software systems.

- True super-user activities that should definitely be performed under the root user only are, for example, software installation and updates (from installation medium), operating system configuring or kernel customization.

4.1. Availability of system administration tools

Tools as provided with the delivery of a UNIX system are, for example, user administration utilities like *reconfig* (or *addusers*), tools for periodic maintenance like *backup* (or *dump*) and *restore*, commands for checking file systems (*fsck*), system shutdown, creating and installing new devices, utilities to install and update software, like *update*, and many more.

Usually, to my knowledge (in all the examples I have seen), all of the tasks outlined above are supported by some UNIX tools, apart from minor exceptions maybe where tools must be provided by system administration, especially for the second category. Every UNIX vendor offers a set of tools normally consisting of ordinary (binary or shell script) utilities scattered all over the file systems' directories. The really unpleasant thing is, however, that every vendor provides **his own** slightly different set of slightly differing tools, which is explicable since there is no common agreement on which tools are to be provided in any UNIX system in the form of a **standard toolset.**

It is not that easy, of course, because some utilities will always remain manufacturer dependent, as is the case with most software installation and update procedures which are tightly connected to the specifics of the manufacturer's software distribution method/media and/or machine architecture. Most device and installation independent tasks, however, certainly could be integrated in some standard toolset even including the two UNIX *cultures*, BSD and System V. Why must we have *dump* and *backup*, for example?

4.2. Usability of existing tools

Due to a variety of factors, the usability of tools for system administration is largely reduced. In the following I shall work out the most important factors interfering with usability, as experienced in our university environment:

- Most tools are just ordinary UNIX commands (utility programmes) residing in various different locations throughout the whole root file system and others. Where which command really resides under which name mainly depends on the specific operating system release used.
- Being true UNIX commands, they are cryptic in use with many incomprehensible command options and the indecent habit of working silently unless there happens to exist a *verbose* mode one happens to have switched on.
- Usually, these commands are invoked from within an interactive shell on an alpha terminal, with all options and parameters as command line arguments. For that reason, most such administration tools suffer from the same lack in user interface ergonomy as most ordinary UNIX utilities. Really useful tools should be more user-friendly and menu-oriented if possible.

So far, all the complaints could of course be used against the normal UNIX programming environment as well. Criticizing the general UNIX user interface would be a major task in itself, going beyond the scope of this article. It should be welcomed, however, that the importance of a really user-friendly interface has been recognized for UNIX system administration (because the chance for UNIX novices to commit fatal administration errors gets drastically reduced).

Some of the tools above, like *update* or *reconfig* for example [HPU88], are somewhat menu-driven, mainly for sake of the complexity and sensitivity of the actions to be executed. Furthermore, there already exist examples for integrated, menu-oriented system administration managers [HPU89]. But let us continue with further factors decreasing the usability of tools:

- All tools (except for some subsystem administration tools, *lp*) are bound to the one real super-user, thus prohibiting their being used by restricted super-users. Moreover, the restricted super-user concept cannot easily be implemented ex post because too many tools are realized as shell scripts (with no possibility to directly apply the SUID mechanism) or internally check for a certain real UID.
- Finally there are great discrepancies between the tools provided in different UNIX systems, both between the two main stream UNICes and between different UNIX versions within a main stream.

  So, through incompatibilities among different UNIX versions, insufficient user interface ergonomy and the extreme emphasizing of the one root user the usability of tools for system administration is greatly reduced. This leads to mediocre if not bad administerability. Hence, improvement of tools is definitely asked for.

4.3. Improving usability of tools

In order to improve existing tools' quality and usability in the short run, vendors could follow the workaround as sketched in 3.3.2. and deliver the whole system with all administration commands neatly arranged in only a few dedicated directories.

In principle, the method works perfectly in multi-user run-level. Unfortunately, there are a few critical commands, like *fsck*, that should only be run in single-user mode. To overcome that difficulty, one could introduce a run-level that is solely reserved to the restricted super-user *sysadmin*. A special version of *shutdown* could accomplish all necessary activities as in the case of "single-user" shutdown, then performing the transition to that new run-level. The system file *inittab* must be adapted to account for the new run-level's semantics.

Another tool badly needed in adapting shell scripts for restricted super-user usage is a shell script compiler, in particular for making shell scripts *execute only* and, as before, to make shell scripts *SUID-executable*. Such a tool is missing in any UNIX known to me and is definitely no standard command either.

5. System Administration Compatibility

When performing system administration on a variety of different UNIX systems, one inevitably comes to desire some uniform way of doing it. While most concepts relevant to system administration are fairly identical at kernel level, the administration interfaces at user command level widely differ in terms of names, syntax and location of system files, utilities and whole subsystems. One major contribution to incompatibility at interface level is due to the two main stream UNICes, System V and BSD(4.2). Without going into detail too much, I just want to point out some obvious differences that seriously impact the uniformity in performing system administration tasks on the different systems which ultimately deteriorates administerability:

- **system files:** A typical example is the system file containing information about local and NFS file system volumes to be mounted upon system startup. In System V this table is called *checklist* located in the "/etc" directory, whereas in BSD4.2 it is called *fstab*. While field structures and semantics of entries are virtually the same with the System V representative [HPU88] and with one BSD4.2 derivate [SUN88], with spaces or tabs as field separators, there is a difference to the other BSD4.2 derivate [ULT86]. The latter just uses colons as field separators. A negligible (yet unnecessary) difference which makes it a tedious job to provide one shell script tool for handling those tables to be used on all three systems.
- **ordinary commands:** See, for example, the command *ps* used to extract information on currently active processes. Not only is the format of the output produced by *ps* somewhat different in System V and BSD4.2, but there are different command options, in name and in semantics. Both versions of *ps* provide some common base functionality, the rest differs. Again, it is difficult to build scripts based on *ps* that are uniformly applicable in all three systems.
- **subsystems:** Consider, for example, the printer spooler subsystem. In System V and BSD, subsystem structure (in terms of directory hierarchie for spooled files etc.), functionality, commands, administration utilities, etc. are completely different.

Several prerequisites have to be met should system administration really be carried out in a uniform, i.e. compatible way on different UNIX systems. First, all important system files (passwd, group, checklist, inittab, etc.) must have same names, same semantics and must reside in the same (directory) location on all the systems. Second, a handful of essential commands must exist on all machines under the same name, with same functionality and same interfaces in terms of options, parameters and return codes. Third, all important concepts, such as the "group access list" mechanism, must be implemented the same. Fourth, the most important subsystems, such as the printer spooler or accounting subsystem, must be identical at conceptual and interface level.

6. Conclusions

This article has pointed out the importance of system administration in general, on UNIX systems in particular. Several objectives of system administration have been stated, such as system security and availability. By taking a closer look at UNIX concepts and tools necessary to support system administration, we have investigated the administerability of UNIX systems.

6.1. How administerable is UNIX?

Several shortcomings have been detected at the conceptual level which prohibit administration from achieving part of its goals. Typical examples are the missing delete permission concept, the missing concept of disc quota monitoring in UNIX System V, the missing facility of dynamically influencing the priorities of background processes and the necessity to keep system files public readable. In all these cases, it is impossible to guarantee system security and/or availability, just for lack of concepts. I strongly believe that this is a serious deficiency of UNIX that might hamper its employment in application areas where safe and reliable operation are main concerns as opposed to system flexibility.

The second experience is that no convenient way of safely, easily and effectively doing system administration exists without constantly abusing super-user privileges. This is mainly due to the lack of adequate UNIX tools and the coarse granularity of the UNIX permission scheme which, together, make it a tough and troublesome task to establish restricted super-users for system administration ex post. These, however, would be necessary to safely perform sensitive routine tasks.

A third experience is that system adminsterability has not yet gained enough attention and importance in the UNIX community. This is mainly reflected by insufficient tool support (see section 4) and by the fact that standardization committees (X/Open Group and others) tend to freeze concepts not sufficient for administration purposes (delete permission, public password file!) by mainly concentrating on unifying the kernel interface. So, portability evidently is still the main direction of standardization efforts. What about a standard for system administration?

6.2. Improving UNIX system administerability

Several suggestions have been made for improving the administerability of UNIX at the conceptual level (section 3). Implementing the new concepts will result in new kernel functionality with implications for

- the kernel's implementation itself including new system calls and modification of existing ones,
- existing commands, some of which must be amended, and
- the UNIX standard(s) which must be adapted accordingly.

But, improvements must also take place on the administration interface level e.g. by defining a **standard** set of preferably menu-driven tools. A third improvement that could be realized in current UNIX anyway is shipment of systems with the minimum access permissions needed on system directories, files and utilities.

6.3. Future perspective

In the long run, system administerability will belong to the major criteria for the usability of an operating system, even for UNIX with all its other benefits. This is especially true for application areas where the ease and efficiency of achieving system security and availability are of prime importance. Some effort will be necessary both by UNIX manufacturers and standardization committees to come up with a version which keeps all of the benefits and incorporates a new one, administerability. That new keyword will greatly dominate the non-technical, commercial market place.

6.4. Afterthought

The author strongly wishes that System V UNICes and BSD systems at least become compatible at the system administration level.

References

[SVI86a] AT&T, *SVID.*

[Bac87a] Maurice J Bach, *The Design of the UNIX Operating System*, 1987.

[HPU88a] Hewlett-Packard Company, *HP-UX6.2*, 1988.

[Com89a] Hewlett-Packard Company, *HP-UX7.0*, 1989.

[ULT86a] Digital Equipment Corporation,, *ULTRIX*, 1986.

[VMS90a] Digital Equipment Corporation, *VMS.*

[Fox98a] Eric Foxley, *UNIX for Super-Users*, 1998.

[POS90a] IEEE P1003.1 Group, *POSIX.*

[SUN86a] Sun Microsystems Inc., *SUN-UNIX*, 1986.

[XPG90a] X/Open Group Members, *XPG2*.

[XVS90a] X/Open Group Members, *XVS*.

[Pet88a] James L. Peterson and Abraham Silberschatz, "Operating System Concepts," *Addison-Wesley Series in Computer Science*, 1988.

Authentication Using a Chip Card

Stanislaus Gefroerer
Ingo M. Hoffmann

Siemens AG
Otto-Hahn-Ring 6
8000 Muenchen 83
Germany

ABSTRACT

To protect information effectively against unauthorized access it is essential to have fail-safe access authorization identification. As is well known, password procedures have their limitations in this respect, especially when used in computer networks. Today, chip card technology provides a means of solving this problem. It enables chip card owners to identify themselves definitively as the rightful owner of the chip card, and thus to prove their identities. A similar proof can be conducted by the computer vis-a-vis the chip card owners. This technique is used for system access to SINIX systems† and is also provided as a program interface for ordinary application programs. The corresponding software product with the associated hardware is available under the name ASECO ‡.

1. Introduction

Information, although immaterial, is nevertheless valuable and therefore deserves to be protected.

Information is always associated with a carrier, often the owner of the information. As we all know, however, information in the possession of the owner is not always safe as there are various ways of persuading a person to speak, even against his or her will.

Other information carriers such as paper, diskettes, cassettes and disks can be stolen, together with the information they contain. Or they can be simply copied, which is even more annoying since this normally doesn't leave any traces. Thus, it is essential to protect information at every level of access. As we all know, this can be problematical, especially at lower access levels such as the level of physical access.

To simplify matters, we have assumed here that in all cases there is only one entity that monitors access to information. This control entity and the information it monitors have a tightly coupled link, e.g. similar to the one that exists between the components of an object in object-oriented programming [Gol83a].

The control entity knows the names of the users and has mechanisms and more or less secret information in order to check whether a person or entity requesting access to this information is authentic, i.e. whether it is in fact what it says it is.

Operating systems, and higher application systems as well, ordinarily use user identifications and passwords as a protective barrier for the information being monitored. This barrier, for whatever reasons, is constantly bypassed, even in especially high-security areas.

Today, one way of remedying this situation is offered by chip card technology, which, for example, makes it possible to prove beyond all doubt the identity of the person demanding access.

† SINIX is the Siemens version of UNIX.

‡ Advanced SEcurity COncept.

2. Authentication Concept on the Basis of Chip Card Technology

An Analogy

The following authentication concept is entirely comparable to conventional key systems of the sort customarily used for the security of buildings. Imagine, first of all, ordinary safety locks with a series of fitted keys which are numbered and, as we all know, can only be replaced or supplemented from a central authority.

In the discussion below, a "doorlock" is likewise a self-contained system and also requires its own keys which fit this lock only. Just as with real systems, a person possesses a key ring with many individual keys exactly fitting the locks for which he or she has access authorization.

Unlike real passive locks, however, when access is attempted the protection entity "doorlock" searches for a suitable key from the key ring and checks that key reliably. The check takes place once only, i.e. it cannot be spied on or repeated.

Moreover, the check is mutual, i.e. the owner of the key is assured that the key ring provides protection from being pulled into a dark room and kidnapped. This may not seem a particularly grave danger in one's own local building, but it can be dangerous in a distant building. It might happen that you are in the wrong building without knowing it and the door only looks like it is locked.

Neither the individual keys nor the key ring as a whole is copyable, but represents unique information. Even keys which fit the same lock but belong to different rings are different. The actual implementation of the keys is kept secret even from the owners who cannot manufacture their own keys themselves.

Finally, the key ring knows its owner. It refuses service to the person using it when that person does not know the secret of the key ring. This secret is not revealed by the key ring under any circumstances, and is even safe from being tapped when owners identifies themselves. And it goes without saying that owners can change their own secrets at any time.

There are no unexpected surprises if the key ring is lost. For one thing, malicious finders cannot do anything with it for they will be detected by the doorlock at the very first attempt. For another, the loser can have a new ring issued with the same authorization, though admittedly this can be somewhat tedious.

We could easily continue with our analogy, but this should be enough to illustrate the possibilities of today's chip card technology – of course "only" as it applies to immaterial systems.

Entities and Authentication Protocols

Taking access to a SINIX system as our example, we will first explain in the abstract the entities involved and their identification and authentication protocols. In the later sections we will then describe the entities in greater detail. Our remarks all refer to figure 1.

The login name helps the login process to identify the user. When non-chip card identifications (login name) are employed, the associated SINIX password is used in the customary manner and serves for user authentication. When chip card identifications are employed, however, the password is generally dispensed with.

In the case of chip card identification, the control entity of the chip card requests owners to prove their identity to the computer. Normally, to do this, the chip card requires owners to authenticate themselves by entering the PIN (Personal Identification Number).

Then the chip card and the control entity of the chip card communicate "end to end" in pairs on the target computer side. Here the target computer can be the user's own local SINIX computer or even a remote SINIX computer which is reachable via rlogin, rsh and other remote components (REMOS).

The chip card first identifies itself to the control entity of the chip card with its "name", the CID (Chip card Identification Number). The control entity of the chip card then authenticates the chip card by sending it the name of a key and an unpredictable random number and waiting for its encrypted value as a response. Here the same securely stored authentication key is used on both sides for encryption purposes. The chip card performs this service only if a necessary PIN check was run beforehand with a positive result.

Depending on which configuration of the SINIX chip card system ASECO is being used, the chip card then checks the control entity in the same way, again using the same authentication key.

If the chip card recognizes the control entity as inauthentic, it refuses to perform any services until further notice. If the control entity is located in a remote computer, the reliable local computer additionally monitors the authentication protocol for the remote control entity and, in this case, interrupts the attempted remote login.

If the control entity recognizes the chip card as authentic and if, due to the properties of the authentication key used, it is certain that a PIN has also been entered, then it causes the caller login to cease its own password interrogation. In all other cases, login password interrogation is carried out.

If a key was used which does not presuppose a PIN entry at the chip card end, login password interrogation is carried out.

Access to SINIX is permitted only if all aforementioned identification and authentication checks are positive.

Symmetrical Encryption

Unlike asymmetrical techniques, symmetrical encryption techniques are based on the principle that two partners own the same secret key [Pip82a, Fum88a, Dif76a, Riv78a, Beu87a].

To mutually prove their identity, they send each other random numbers and expect the partner to respond with the random number encrypted by means of this key (Figure 2). The key must be a genuine secret of both partners, otherwise a third party might produce the same "proof".

In daily practice, this means that each person or computer which is a partner of many other communication entities must own a corresponding number of secret keys. This problem of key management is simplified with the aid of a "global key" that makes it possible to derive the "individual key" of each partner largely from the name of that partner.

At the moment, SCA-85 (Smart Card Algorithm) is used as the encryption algorithm[Beu87b]. The SCA algorithm is a symmetrical block cipher algorithm in which encryption takes place in 64-bit blocks. It resembles the DES (Data Encryption Standard) but requires considerably less storage space for the algorithm.

Chip Card

The chip card is the Siemens Computer Card, Version 1, with SCA-85 [Kru87a, Kru89a]. It consists of the processor chip and the chip card proper with contact layout and assignment in accordance with [ISO78a].

As a rule, one chip card contains several individual keys †, which are loaded on the chip card in several stages using a separate procedure. This procedure is based on corresponding ISO standards [ISO10a]. It requires a separate chip card personalization processor which will at first be available as a personal computer (PC) and which must be operated in a secure environment. Naturally, an organizational service will also be provided to personalize chip cards on request. In a further step it will be possible to load authentication keys onto chip cards and likewise into security units using your SINIX computer, with no danger of these operations being tapped.

The keys can be marked differently on the chip card so that one PIN entry will suffice for all subsequent chip card authentications, or so that each authentication forces a new PIN check, or even so that no PIN check is required.

The chip card is the authentication partner of the control entity. On the authentication protocol level it interprets a series of instructions for reading non-secret data, selecting a key system, generating random numbers, encrypting random numbers, evaluating a PIN entered by the user, and much else besides.

Control Entity

The control entity is located in the relevant target computer. Basically, it is an authentication process which is assigned to the SINIX terminal or to the window and caller process (e.g. login process), and which in turn uses for its authentication functions a security unit specific to the SINIX computer. In addition to this special hardware, this control entity also comprises in particular the authentication interface linked into the caller program, a central daemon process for chip card authentication and, of course, the caller-specific authentication process.

† Mechanisms for read-locking the keys on chip card are protected by the rights of Innovatron.

Internally, the security unit has all the secret global keys which can be used for authentication purposes in the associated SINIX computer. The security unit generates random numbers, calculates the individual key, evaluates the random number encrypted by the chip card, encrypts random numbers of the chip cards and sends its authentication result to the SINIX computer. As far as authentication is concerned, it is the actual communication partner of the chip card. The security unit is located outside the SINIX computer and is connected to it in a conventional manner via a serial interface (SS97).

The authentication process uses special mechanisms to detect even highly sophisticated manipulation of the authentication protocol, both on the links to the security unit and on the chip card.

Chip Card Terminal

The chip card terminal is used to connect the SINIX computer, or more precisely the control entity of the chip card, with the chip card itself. The chip card terminal has a contact unit for the chip card and makes use of its own protocol vis-a-vis the chip card [Bud89a].

Moreover, the chip card terminal is a kind of secure terminal for the chip card. It guarantees tap-proof transfer of the PIN to the chip card [ISO95a].

Like the security unit, at the moment the chip card terminal is connected to a SINIX computer via a serial interface (SS97 or V24). Moreover, a connection to a character terminal is currently being developed.

The chip card terminal is controlled on both the linkage and application levels by the caller-specific authentication process, which, in the case of remote authentication, has its own partner process in the local computer.

At the application level, the chip card terminal basically recognizes instructions for operating the chip card (reset, eject), for issuing messages to the chip card terminal display, for transmitting messages between control entity and chip card, and for handling the PIN check. The PIN is entered at a separate keyboard, and due to the properties of the device it can only be transported to the chip card.

Changing the PIN is a function which is performed offline by the chip card terminal, i.e. independently of the SINIX system.

A special variant of the chip card terminal makes it additionally possible to read the magnetic strip.

A third variant has neither a keyboard nor a display, and can only be used in connection with keys which do not require a PIN from the user. Normally, devices of this sort are only used for authentication in the local computer. In this case, a SINIX password is additionally requested which, however, is not transported via the network and therefore cannot be tapped on a network connection.

3. System Access with Chip Card Authentication

As an example, Figure 3 illustrates a hardware configuration as it appears to the user.

Whether all or just part of the time, a user works at a particular "workstation". From this station one or more character terminals and also graphics terminals can be used simultaneously. At each terminal the appropriate window system (ALPHA-COLLAGE, COLLAGE ® or X Window) is available. In extremecases, for example, the su command can be issued from each work place terminal, using every window each time, thereby requesting chip card authentication.

The first chip card request, e.g. with a login, causes the workstation to be permanently assigned (until further notice) a chip card terminal which, in principle, the workstation is permitted to use and which, at the same time, is :currently: available.

The chip card terminal operation prompt appears at the relevant terminal of the workstation, if applicable in the relevant window within the reference to the assigned chip card terminal (ttyname). The PIN dialogue is conducted via the display at the chip card terminal. Conversely, each message in this display also contains the reference to the terminal which commissioned the job (ttyname).

Following the PIN check and the first authentication, the chip card can remain in the chip card reader. All further authentications of this workstation are automatically referred to the same chip card terminal and the same chip card, and, in the case of a conditioned PIN check, they do not request any further PIN entry or password. Users can take advantage of all chip card-related privileges via all the terminals at their work place without seemingly having to offer further proof of identity.

If the chip card remains in the chip card terminal, the terminal display will show from which workstation the chip card terminal has been reserved. A special command permits users to terminate this reservation and to eject the chip card. They can enter this command on any terminal at the workstation. In case of error, the chip card terminal has a local eject key.

It goes without saying that authentication operations are also supported, with the chip card being ejected automatically following completion of the authentication operation so that the chip card terminal is immediately free for other workstations. Once again, exceptions to this procedure are those chip cards which are marked "professional" and whose owners therefore make use of authentication more frequently.

Similarly, when corresponding keys are employed, those authentication operations are supported that force a separate PIN entry for each instance of authentication (security concept for dead man detection).

It is possible to restrict the use of a workstation to particular chip cards.

4. Authentication Service

"login" and "su" use a widely applicable program interface for chip card authentication. This interface is part of ASECO and can be used for customer-specific application programs. In this way the user is provided with a general chip card authentication service.

The central concept for this interface is a particular application concept which is outlined in the next section.

Application Concept

An "application" is a tightly coupled link between a concrete item of information to be protected and its control entity as well as the access protection functions of that control entity, including the specific chip card authentication.

An application can readily be understood in terms of object-oriented programming as an object. Access to the particular item of information is permitted if and only if the accessing entity can prove that it is known to the control entity as a person or computer and that it possesses a suitable secret key. In actual practice, this control entity determines which key or keys are permitted and which authentication features in particular are to be performed.

Figure 4 shows the application concept as a semantic network illustrated within an object-role model [Nij85a].

An application belongs to an application class (applclass) and inherits from this class a specific EUID (Effective User IDentification) or EGID (Effective Group IDentification). This ensures that an authentication request will be permitted for a particular application only if the calling program possesses the EUID or EGID of the associated application class.

An application may use two or more keys, but only those which are known to the system itself (adfknown). During authentication, a key is selected which is also active at this point in time (adfactiv). Preparations have already been made for the use of two or more keys within an authentication operation as a prerequisite for access. This makes it possible, for example, to implement the "two pair of eyes" principle, where at least two persons of an authorized group have to prove their identity before access is permitted.

An application can either have its own login names or inherit them from another application. The same applies to the application-specific rights of a chip card with regard to permissible login names.

This application concept makes it possible to implement various interesting system access requirements.

For example, the access modes login and su for the application class "SINIX access" are distinguished according to local and remote access, i.e. four applications are normally set up in this case. The user identifications and the chip card user identifications may therefore be defined separately for each application as required. Otherwise, "login via locally-connected terminals" is inherited one step at a time, possibly via the parent.

Thus, for both "login" and "su", this makes it possible for access via terminals connected locally at the computer and access via terminals connected to a remote computer to be handled in different ways, since access to the user identifications and chip card authorization for chip card identifications can be defined specifically for each case. Moreover, separate keys can also be required for each case.

In this way, for example, the root identification can be generally locked for login and permitted for su (and here only in connection with local access) (F2/Q2). In conjunction with chip card authentication, this uniquely assigns each operation under the root identification to one and only one person.

Each attempted access operation is logged separately according to positive and negative authentications, and separately according to login name and chip card.

Authentication Interface

The authentication interface is implemented as a C function and is provided in the form of a library. The interface expects to receive, as arguments, the name of an application and a login name referencing this application. At run time, of course, all data required for authentication must exist in the database of the chip card system.

This function returns the authentication result and also indicates whether callers should dispense with their own password check. The password check should be dropped whenever it does not enhance security but merely represents an unnecessary security risk since the password would be transmitted over a network connection.

In addition to this entry authentication, preparations have also been made for an exit and a sporadic authentication. In the latter case, authentication is activated and performed at random, regardless of the caller process involved. If the result is negative, the caller process is informed via a function which must additionally be included.

5. Administration Functions

A separate user identification is employed for purposes of administration. A command interpreter supports processing of the chip card system's database.

In the command interpreter, for example, it is possible to handle the aforementioned application classes, applications and all associated objects. Similarly, key names in various levels of activation, chip cards and security unit can also be dealt with, as can terminals, chip card terminals and workstations, including any relations to other objects.

At the moment, the command interpreter interface is designed to handle set processing. It provides options for the external representation of objects, the processing of external representations in an editor, and operation-specific conversion to the internal representation, including the checking of semantic rules governing the database.

Special commands support the transfer of identification data from chip cards and security units which were personalized decentrally by a separate chip card personalization processor.

Further commands are used for transferring SINIX system data such as user identifications and device configurations for terminals and chip card terminals. This transfer can be linked to the dynamic usage of the SINIX system or refer to the current system status.

6. State of Development and Future Outlook

Version 1 of the ASECO software product will be with the associated hardware available to clients from about the middle of 1990. It is based on SINIX V5.22 and will also be usable with the corresponding SINIX version with F2/Q2 quality.

From the standpoint of the SINIX chip card system, the chip card-secure computer will be able to stand in a freely accessible office environment.

Chip card authentication in a remote computer (REMOS) is not possible at the present time, but is already in preparation. In the first version, rlogin can be permitted on an administrative level for identifications without chip card security; access will be denied to chip card-secure identifications. Preparations are being made for authentication in the remote computer by means of its control entity and security unit. In this case, all types of remote access available under SINIX will be included. Client and server authenticate each other mutually "end to end" with the aid of their security units.

This authentication protocol cannot be emulated or manipulated by third parties. Every protocol unit to be transmitted is chained to the most recently transmitted protocol unit with the aid of the key common to both computers. It is planned to provide a corresponding service as a program interface as well so that application programs will be able to pass messages individually and authentically between different computers. This secures the source for each message.

A transport interface to support terminal emulation for accessing chip card-secure mainframe computers is currently being developed. Users will then be able to obtain access to BS2000 chip card identifications [Heia] from their SINIX computers using their SINIX terminals and chip card terminals. If they have already entered the PIN, it is usually not necessary to enter it a second time.

References

[ISO78a] *ISO IS 7816.*

[ISO10a] *ISO DP 10 202 Security Architectures of Financial Transaction Systems using integrated Circuit Cards.*

[ISO95a] *ISO IS 9564-1 Banking Personal Identification Number Management and Security - PIN Protection Principles and Technics.*

[Beu87a] A Beutelspacher, *Kryptologie*, Vieweg Verlag, Braunschweig, 1987.

[Beu87b] A Beutelspacher, "Die SICRYPT-Chipkarte - ein Sicherheitswerkzeug der Zukunft," *Siemens-Zeitschrift*, April 1987.

[Bud89a] Budespost, *Gemeinsamer Vorschlag der GZS und der Deutschen Bundespost fuer ein nationales Asynchrones Halbduplex-Blockuebertragungsprotokoll*, 22nd September 1989.

[Dif76a] W Diffie and M E Hellmann, "New Directions in Cryptography," *IEEE Transaction on Information Theorie vol. 22 no. 6*, pp. 644-654, November 1976.

[Fum88a] W Fumy and H P Ries, *Kryptographie: Entwurf und Analyse symmetrischer Kryptosysteme*, Oldenburg Verlag, 1988.

[Gol83a] A Goldberg and D Robson, *Smalltalk-80: The Language and its Implementation*, Addision-Wesley Publishing Company, 1983.

[Heia] D Hein and H Joerg, "Authentifikationsverfahren in BS2000," *KES, Zeitschrift fuer Kommunikations- und EDV-Sicherheit*, Peter Hohl Verlag.

[Kru87a] D Kruse, *Chipkarten, was sie wirklich koennen*, Peter Hohl Verlag, Ingelheim, July-August 1987.

[Kru89a] D Kruse, "Chipkarten, klein im Format gross in der Leistung," *DuD*, Vieweg Verlag, Wiesbaden, April 1989.

[Nij85a] G M Nijssen and E D Falkenberg, *Introduction to Information Systems, Lecture Notes*, University of Qu_eensland, Deptartment of Computer Science, 1985.

[Pip82a] I Piper and H Beker, *The Protection of Communication*, Chiper Systems, 1982.

[Riv78a] R Rivest, A Shamir, and L Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communication of the ACM, vol. 21, no. 2*, pp. 120-126, February 1978.

For further details, contact
The Secretariat

European UNIX® systems User Group

Owles Hall, Buntingford, Herts SG9 9PL, UK
Tel: +44 763 73039
Fax: +44 763 73255
Network address: euug@EU.net

SBN 0 9513181 6 0