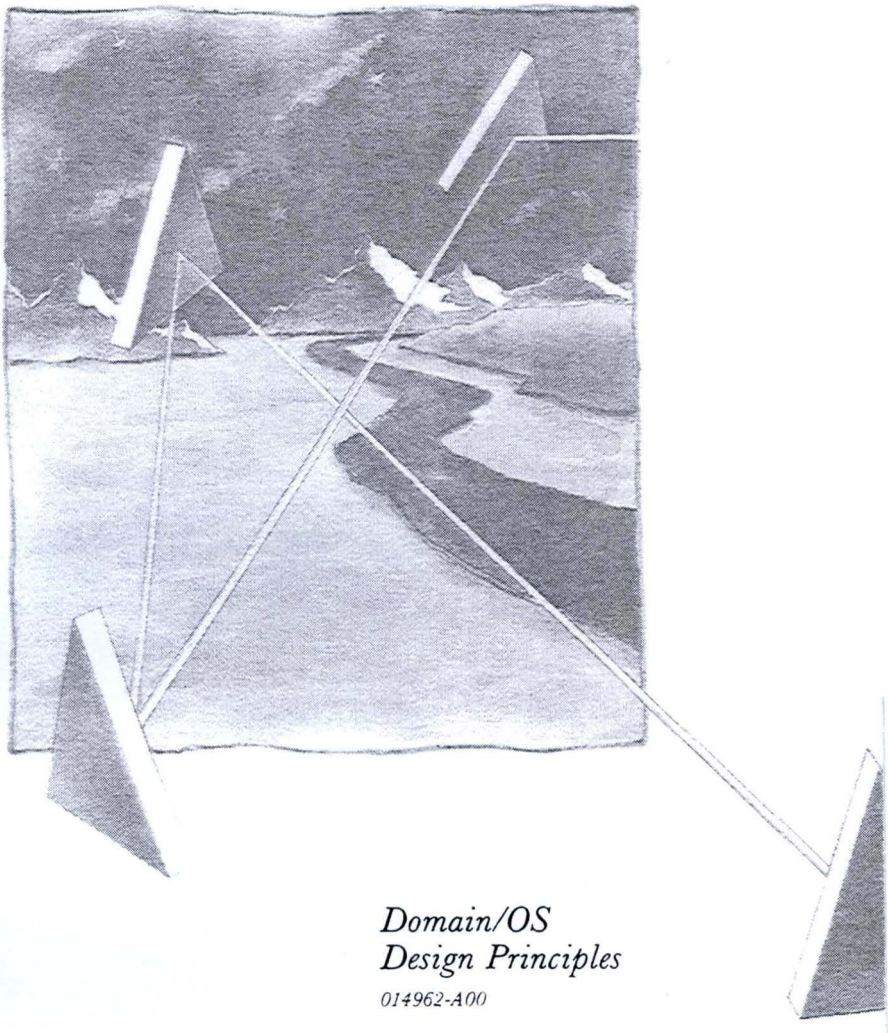


Domain/OS Design Principles

014962-A00

apollo

Domain/OS Design Principles

Order No. 014962-A00

Restricted Rights Notice

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824
(508) 256-6600

Notice: Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software - Restricted Rights clause.

First Printing: January, 1989
Latest Printing: January, 1989

This document was produced using the Interleaf Technical Publishing Software (TPS) and the InterCAP Illustrator I Technical Illustrating System, a product of InterCAP Graphics Systems Corporation. Interleaf and TPS are trademarks of Interleaf, Inc.

Preface

Domain/OS Design Principles describes the architecture and design of Domain/OS, Apollo's workstation operating system.

We've organized this manual as follows:

Part 1	Introduces the origins of Domain/OS and discusses the fundamental design principles we used to develop the system.
Part 2	Contains technical papers by Apollo engineers that describe in greater detail some of these design principles.

References for the material in each chapter are at the end of that chapter.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

literal values	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Pathnames are also in bold. Bold words in text indicate the first use of a new term.
user-supplied values	Italic words or characters in formats and command descriptions represent values that you must supply.
sample user input	In examples, information that the user enters appears in bold.
`output`	Information that the system displays appears in `this typeface`.
[ ]	Square brackets enclose optional items in formats and command descriptions. In sample Pascal statements, square brackets assume their Pascal meanings.
{ }	Braces enclose a list from which you must choose an item in formats and command descriptions. In sample Pascal statements, braces assume their Pascal meanings.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/	The notation CTRL/ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you press the key.

. . .

Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

.
.
.

Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

———— ⊞ ————

This symbol indicates the end of a chapter.

Contents

Part 1. Domain/OS

Chapter 1 Overview

Chapter 2 The Origins of Domain/OS

Part 2. Technical Papers

Chapter 1

The Origins of Domain/OS

Domain/OS, which first shipped to customers in July 1988, is Apollo's workstation operating system software. It comprises a common kernel and three environments, **BSD**, **SysV**, and **Aegis™**. BSD and SysV provide the two major UNIX* operating environments and Aegis supplies Apollo's original operating environment.

This chapter explores the context from which Domain/OS arose. It includes a brief overview of operating system architecture in general and of Domain/OS in particular, a discussion of some of the special operating system requirements of the workstation market, and some remarks about the role that the UNIX system played in the development of Domain/OS.

* UNIX is a registered trademark of AT&T in the USA and other countries.

Architecture of a Workstation Operating System

The purpose of any computer's operating system is twofold. First, it implements an abstract machine that is far more convenient to use than raw hardware. Second, it allocates, controls access to, and otherwise manages such physical computing resources as processors, memory, and peripherals.

For its users, however, an operating system should allow the most effective and efficient use possible of all the resources in the computing environment and give each of many users the illusion of exclusive use of the machine. For workstation users, these resources include CPU power, memory, network bandwidth, and disk space.

However, the most important resource is clearly the users' time. For software developers, in particular, the operating system has to minimize development time, provide tools and mechanisms to facilitate innovation and realize, as much as possible, the maximum power of the hardware. It must also leverage developers' efforts by ensuring portability for their software.

New technology has changed the character of operating systems, as well. In recent years, we have seen the advent of inexpensive, high-speed local area networks, high-resolution bitmap displays, multi-MIP CPUs, multiprocessor hardware, and less expensive semiconductor memory and high-capacity disks. All of these have put new demands on traditional operating systems.

Domain/OS Overview

Apollo's Domain/OS is a high-performance UNIX workstation operating system that is built on object-oriented principles. It effectively exploits the new hardware technology of the last few years, both in exclusively Apollo environments and in heterogeneous environments. The issue of heterogeneity has become increasingly important, as fewer and fewer customers are willing, or even able, to restrict their sites to using computing equipment from a single manufacturer.

Domain/OS consists of a common kernel with three operating environments. The Aegis environment provides all the functionality of the Aegis operating system, Apollo's original operating environment, and the BSD and SysV environments provide users with enhanced Berkeley Software Distribution 4.3 and AT&T System V Release 3 UNIX environments, respectively.*

Each of these environments can run without relying on the others. The environments can also run concurrently, so that any Domain/OS site can use two or three environments and enjoy a great deal of flexibility. By providing separate implementations of the two major UNIX development threads, rather than one "amalgamated" UNIX system, Domain/OS can track both standards as they evolve. The availability of the two UNIX environments also fulfills our customers' needs for software portability.

Domain/OS uses Apollo's Open System Toolkit™ (OST) to enable customers to extend the power of the operating system and to support true distributed computing in multi-vendor environments with the Network Computing System.

The Open System Toolkit provides tools that allows operating systems programmers to create new types of I/O targets (that is, devices) without modifying the operating system's source code. The Open System Toolkit also includes facilities to add new object types to the system.

* SysV is compatible with the System V Release 3 Interface Definition (SVID) for Base OS, Base Libraries, and Library Extensions.

In traditional UNIX systems, when device drivers are written for new devices, a systems programmer must modify and rebuild the operating system source code to install the new driver. The OST approach makes it far simpler and cleaner to add device drivers to the operating system. Other concepts associated with the OST (object types, type managers, extended naming) make it significantly easier to customize the Domain/OS system to answer a particular site's needs. Chapter 2 discusses these concepts.

One of Domain/OS's major strengths is the distributed file system, which provides transparent access to data anywhere in the network. In addition to having distributed data, however, Domain/OS offers true distributed computing in the form of the Network Computing Architecture. In addition to providing ways to make optimal use of computing resources, the Network Computing Architecture provides ways to take advantage of parallel processing and specialized hardware as well.

The Network Computing System™ is a portable implementation of the Network Computing Architecture that runs on both UNIX systems and other systems. In addition to being object-oriented, the Network Computing System supplies a transport-independent remote procedure call facility. The system is built on a concurrent programming support package that allows multiple execution threads in a single address space and also contains a replicated global location database for objects.

The Role of the UNIX Operating System

Traditionally, each computer manufacturer designed its own operating system to meet the needs of both its own customers and its internal software developers. Recently, however, it has become clear that one of the best ways to maximize the productivity of software developers and to provide the customer some measure of vendor independence is to make the operating system's services available through interfaces that are standard across many manufacturers' equipment. In the workstation world (as in the minicomputer, and, increasingly, the mainframe worlds), the standard is defined by the UNIX operating system.

More than a standard set of interfaces, however, the UNIX system is a framework into which new interfaces will be incorporated, as well as a context and a common language for discussing new operating system ideas. The style already established by existing UNIX features will, to a large degree, determine the shape and feel of future operating system functionality.

Domain/OS provides, in addition to the UNIX interfaces, compatible extensions. One example of these extensions is the Domain/OS protection system, which provides all the functionality of the traditional UNIX modes, but extends them by providing more flexible and more finely grained protection levels. These extensions are entirely optional and do not interfere if users wish to run a "pure" UNIX environment on Apollo hardware.

UNIX Compatibility — Implications

It is difficult to overestimate the effects of having the UNIX operating system as a standard. Many types of application software are being developed exclusively, or at the very least, first, for the UNIX system because of the tremendous leverage the system provides in making a developer's software available to a wide audience. Many new standards (in windowing systems, for example) are being developed for UNIX systems, and are therefore coming into widespread use much more rapidly than they would otherwise.

But the notion of UNIX compatibility extends beyond the programming interfaces to the areas of performance and user environment. Developers porting UNIX code expect the relative performance of

various system facilities to be more or less the same from one UNIX system to another.

Furthermore, UNIX systems provide a user environment that is fairly consistent from machine to machine. As a result, programmers, system administrators, and other end users all find that many of their skills and work habits carry over from one UNIX system to another. This allows organizations that run or develop software to make productive use of newly hired employees more quickly.

So the first requirement for a competitive operating system in the workstation market is UNIX compatibility in all aspects. In its UNIX environments, Domain/OS provides both the interfaces and the feel of a UNIX system. In addition, Apollo continues to use the UNIX software as a base for implementing compatible extensions, and proposing those extensions within such standards bodies as the IEEE POSIX group, the Open Software Foundation (OSF), and in the UNIX community at large.

The UNIX Philosophy

The original developers of the UNIX system were programmers who had pragmatic ideas about how an operating system should be structured and used. These ideas have seeped into the software development and engineering culture, and are known collectively and informally as the UNIX philosophy. The major points of this philosophy are generally agreed to be:

- Provide just enough mechanism to get the job done, and no more.
- Let each command do one thing well.
- Don't try to do the user's job. Provide tools as building blocks, which can be arranged as necessary to perform a task.
- The output of one command should be usable as the input of another command.
- Files are unstructured byte streams. Applications may impose any internal structure, but to the system, a file is a file.

Some aspects of this philosophy are no longer as relevant as they once were. As UNIX systems have become platforms for applications more than just a means to support programming and word processing, we have seen the emergence of applications programs that approach (or even exceed) the operating system itself in size and complexity. Areas such as computer-aided drafting (CAD) and desktop publishing create self-contained, highly interactive, graphics-intensive environments in which the end user is largely unconcerned with the nature of the underlying operating system.

However, one aspect of the UNIX philosophy, often implied but seldom stated explicitly, is still extremely relevant. It has to do with finding the proper relationship between generality and performance. There are two parts to this:

- Find the right balance between the current and future needs of the customer.
- Provide what you can implement efficiently today. Don't offer functionality that demands more performance than the current technology can offer, because the functionality won't get used.

However this point is stated, it is clear that trying to achieve these goals is the essence of skillful engineering. It is also clear that the technology available today makes it practical (and even necessary) to offer features that might have been too costly just a few years ago.

While important, UNIX compatibility is by no means sufficient to solve all of today's operating system problems, and the desire to maintain compatibility with the UNIX system should not obviate the possibility of improving and extending it.

Beyond UNIX Compatibility

Current UNIX implementations fail in several ways to provide maximum value to users. This is not surprising. Although the UNIX kernel has evolved considerably since its original implementation in the early 1970s, its basic characteristics have not changed in any fundamental way.

At the time of its inception, most of the machines available were minicomputers with low-powered (by today's standards) CPUs,

limited physical memory, small disks, and little or no networking capability. The expense of the hardware dictated that these machines be timeshared.

As time has passed, much new functionality has been added to the UNIX system, but most of it has been added to the kernel, making it quite bloated. As a result, the kernel has lost much of its original elegance and simplicity, and is no longer the best match for current technology.

Other developers have begun to recognize the limitations of the traditional UNIX kernel [1], and are exploring new ways of structuring the operating system. They hope to demonstrate that an operating system can realize the benefits of the UNIX system without being subject to the constraints of a monolithic kernel.

It is possible to enumerate a number of principles and general properties that such a restructured UNIX system should possess, and Domain/OS already embraces many of these principles. In the next chapter, we discuss in more detail the design principles underlying the Domain/OS operating system.

References

[1] R. Rashid. *Threads of a New System*, UNIX Review, 4, No. 8 , pp. 37–49. August, 1986.

Chapter 2

Domain/OS Design Principles

Domain/OS is a complex operating system. It employs several interesting design principles to manage that complexity and to avoid some of the pitfalls of traditional, monolithic UNIX kernels. Among these principles are:

- Object orientation
- A small kernel
- System functionality in user space
- Dynamic loading, linking, and sharing of system libraries

The goals of Domain/OS's design include:

- Support for very large virtual address space processes
- More efficient use of physical memory
- Single-level store and transparent object location
- Network-wide access to file system objects through virtual memory management
- Scaling to many machines and/or users

- Extensibility by users
- Support for multiple operating system environments and multiple processors

The sections that follow expand on these fundamental Domain/OS design principles and goals.

Object Orientation

Briefly, under the object-oriented model, the world consists of a collection of opaque abstract objects. The behavior of these objects, but not their internal implementation, can be discerned by other objects. The state of an object can be altered or observed only through the interfaces (operations) that the object exports to the rest of the system.

Because the behavior of one object does not depend on the implementation of another object, any object's internal implementation is free to change, as long as it still presents the same behavior to the rest of the world.

Object-orientation in Domain/OS is largely by convention, since the language used to implement the kernel does not enforce information-hiding and data abstraction. This allows Domain/OS to violate the rules of the object-oriented model, when necessary, to avoid the expense of excess layering that is inherent in a pure object model.

System Functionality in User Space

A general goal of the Domain/OS design was to minimize the size of the kernel by implementing system functionality in user space, where doing so would not compromise security or performance. This has several benefits:

- It makes it easier to prototype, implement, and debug these facilities, because all of the interfaces and tools available to any application program are accessible in user space.
- In some cases, better performance results because traps to the kernel can be avoided.
- It makes it possible to avoid paying the cost of functions that aren't necessary. User space libraries and servers that implement optional functionality need not be installed.
- It allows users to substitute for or extend system functionality without modifying the kernel.

Domain/OS makes user-space implementation of system functionality practical by providing some of the same mechanisms in user space that traditional kernels (including the UNIX kernel) depend on internally.

The most important of these are a shared memory facility for cheap, high-bandwidth communication among processes and an inexpensive mutual exclusion mechanism, based on eventcounts. Additionally, a user-space cleanup mechanism is needed to make sure that user-space global state (see below) is properly cleaned up when a process dies, as well as an inexpensive way to defer asynchronous signals while critical sections of user-space code are being executed [5].

Most UNIX implementations do not have these facilities available outside the kernel. It is critical that such mechanisms be cheap, so that moving functionality out of the kernel does not incur a performance penalty. For example, the mutual exclusion call for entering a critical section only executes six machine instructions (and no system calls) in the case where there is no contention for the mutual exclusion lock.

A typical example of these facilities in Domain/OS is Apollo's TCP/IP implementation, which is implemented entirely in user space (using kernel network device drivers), and achieves a highly competitive performance.

In addition to these facilities, Domain/OS includes the GPIO subsystem, which allows device drivers to be written, debugged, and installed entirely in user space. This makes the work of adding a new device driver to the system much simpler than in traditional architectures where a new device driver must be bound into the kernel.

Perhaps what is most important about these design principles is not that they have been used internally to structure the operating system, but that they also form the basis for allowing users to extend the functionality of the system in ways that Apollo may not have anticipated.

In particular, the object-oriented approach makes it possible for users to customize or extend the system without knowledge of how other portions of the system are implemented. This user extensibility is discussed in a later section.

Dynamic Loading, Linking, and Sharing of System Libraries

UNIX applications can use services that are provided by the kernel as well as services provided by libraries. The kernel services are obtained by trapping into the kernel from user mode. The library services are obtained by binding the necessary routines into the executable image of the library. Thus, binding to kernel services is effectively done at system boot time, while binding to library services is done at program bind time. This scheme has three obvious disadvantages:

- It wastes disk space, since copies of the library routines are duplicated in every command.
- It increases the total working set of processes, since each process is accessing its own copy of the routines.
- It creates a serious maintenance problem for third-party software vendors: if a manufacturer fixes bugs or enhances performance in the library routines, the software vendor's customers will not see those changes until the vendor rebuilds and redistributes its applications to its customers.

These problems are solved in Domain/OS by the use of shared libraries. There are two kinds of shared libraries: global and private. The implementation of shared libraries relies on three basic mechanisms present in Domain/OS: position-independent code, the known global table, and global user address space.

Position-Independent Code

The Domain/OS compilers can generate position-independent code (PIC). This has the effect of inserting an extra level of indirection for each external procedure and each reference to global data. For a procedure, this extra code is called a transfer vector, and is located in the read/write data section of a module. The transfer vector simply jumps to the actual procedure entry point.

This allows the procedure text of a shared library to be loaded anywhere in an address space without relocation. The procedure text

can therefore be read-only, and thus be shared by all the processes that use the library. All load-time relocation takes place in the data area, which already must be writeable anyway.

Known Global Table

The Known Global Table keeps track of all the symbols exported by the libraries currently installed. It is consulted whenever a program that has unresolved external references is loaded. Any such references that are found are filled in with the symbol's value from the Known Global Table.

Virtual Address Space Layout

Domain/OS supports the concept of global user address space. Actually, it divides the address space into four partitions: global supervisor, global user, private supervisor, and private user. Global address space is shared among all processes, with global supervisor space being the portion where the kernel text and data are mapped. It is readable and writeable only when the system is executing kernel code as a result of a system call, an interrupt, or a context switch to a kernel-only process. Global user space, on the other hand, is accessible to any user space code, just like the more traditional private user portion of the address space.

Global Libraries

Global shared libraries are installed into global user space once, at boot time. Each node has a system configuration file that specifies which libraries are to be treated as global and which as private. All libraries specified in this file as either or global or private (the exact terms are **global** and **shared**) have their entry points stored in the global Known Global Table.

When a program that makes calls to global library entry points is loaded, that program's transfer vectors are filled in with the values found in the Known Global Table, and the program is then ready to run.

Many functions traditionally found in the kernel are implemented in the global libraries. For example, the entire device-independent I/O layer (i.e., **open, close, read, write**) is in a user-space global library.

Private Libraries

In addition to being marked as shared in the system configuration file, private libraries can be marked as either **static** or **dynamic**. Marking a library static means that the library is loaded when a program containing a reference to the library is loaded; marking it dynamic means that the library is loaded when a program actually executes a call on the library.

The configuration file also allows you to designate libraries as optional; no error is reported if the library is not found. A default set of libraries is always loaded if the configuration file is not present.

Support for Large Virtual Address Space Processes

Traditionally, UNIX systems require an area of the disk or disks to be preallocated as swap space. This space can be used only for swapping or paging, and cannot hold regular file system objects. The sum of the sizes of the virtual address spaces of all current processes must not exceed the amount of swap space. Thus, for example, if 8 megabytes of swap space are allocated, no process (or combination of active processes) can use more than 8 megabytes of virtual memory.

If the preconfigured swap space is too small, the system must be reconfigured. That process generally requires dumping all the file system objects on that physical volume to tape, rebuilding all the file systems, and then restoring the data from tape.

With the advent of processors that support huge virtual address spaces and applications that use them, these limitations are clearly no longer practical. They impose arbitrary limits on process virtual address space size and are an inefficient use of disk space.

Domain/OS allocates backing store for disk space dynamically, as it is requested by a process. Disk space does not have to be designated in advance as file space or swap space. Thus, the size of a process's address space is limited only by what the processor can support and by the total amount of free space on the logical volume. This means, for example, that when a process performs an **sbrk**() operation to increase the size of its data area, it can obtain the space needed to back up the newly added pages of address space from anywhere on the file system volume.

More Efficient Use of Physical Memory

Another problem related to the size of a process's virtual address space size is that current UNIX kernels generally require that all the page tables for a process reside in physical memory while the process is active. This represents an inefficient use of physical memory, and adds another unnecessary constraint on process virtual address space size.

Domain/OS can swap the page tables for an active process out to the disk. This means that there can be a fixed upper limit on the amount of physical memory devoted to page tables, with no effect on the amount of virtual address space a process can use. If all of the currently active processes have relatively modest combined virtual memory requirements, then all of their page tables can fit in memory. In that case, no swapping takes place, and there is no impact on performance.

Single-Level Store

Some operating systems divide storage into several levels. A machine's main memory acts as the primary storage level, while the disk acts as secondary storage. In this scheme, programs have direct access only to the primary storage level; they must explicitly copy data from secondary to primary storage before they can operate on it.

Domain/OS uses a single-level storage mechanism, whereby a program gains access to an object by mapping object pages directly into the process's address space. With the single-level store, all objects in the network are accessed in the same way, regardless of whether they reside on the local disk or on another disk in the network. Users can share the same program and/or data file, and can execute a program without regard for the location of files that it uses.

Both disk and network I/O are implemented by way of demand paging. The file I/O manager maps file pages into the virtual memory of a process. When a program attempts a read or write operation, the file I/O manager starts at the current seek pointer location and copies the pertinent data from the place in the process's address space where the file has been mapped to the user's program buffer. (A program can also be set to perform I/O in a mode that eliminates the data-copying step.) If the data requested is not in real memory, a page fault occurs.

Thus, there is a direct mapping between object pages (regardless of where they reside on the network) and process virtual address space. With this direct mapping feature, processes can access objects using programming language variables, arrays, strings, and other constructs. In addition, once the object is mapped into a process's virtual address space, the system does not demand-page any data until the process actually refers to it. Thus, processes can map the objects without excessive system overhead.

Domain/OS makes more efficient use of physical memory by allowing all of it to be available as a cache over the file system. The system uses the same mapping and demand paging mechanism for program execution, as well. Because the demand paging mechanism operates transparently over the network, you can execute programs

on nodes with or without disks, without additional special mechanism.

Of course, Domain/OS provides the standard UNIX I/O interfaces: **open, close, read, write,** and **seek**. In addition, Domain/OS allows direct access to the mapped file interface for applications that require the maximum possible I/O throughput.

Areas

One exception to the single-level store mechanism lets Domain/OS create processes faster and less expensively. Instead of using a file system object to back up portions of virtual address space, we use a pseudo-object called an **area**. The area mechanism is based on the System V **regions** model, but since that term has another meaning specific to Domain/OS, we refer to this pseudo-object as an area.

An area is a set of contiguous segments in a process's virtual address space. The process itself creates and maps the area. Other processes can map any portion of an area into their own address spaces.

An area has a unique identifier (UID) by which it can be mapped, just as an object can. It appears to the rest of the operating system as a file system object, but without the overhead of creating, deleting, and manipulating.

Areas let you avoid manipulating objects directly. Since all allocation, deallocation, and growth operations are performed in memory, there are no file maps to adjust and thus no disk buffering or other disk operations to perform.

Virtual Memory Management

The Domain/OS virtual memory management scheme provides network-wide access to objects with two related operations: mapping and demand paging.

Mapping — the Single-Level Store

We discussed mapping in a previous section on the single-level store. Briefly, the single-level store allows a program to gain access to a file system object by mapping the object's pages into the process's address space. Once the object is mapped in, individual segments of it are moved in and out of the address space by a mechanism called **demand paging.**

Demand Paging

In demand paging, the system dynamically transfers pages of an object in and out of physical memory, both locally and over the network. The object may reside on the local disk or on a remote node's disk. Each node has a remote paging server process to handle remote requests to read and/or write pages of objects that reside on that node. When another node references an object belonging to that node, the paging server dynamically transfers the data to the requesting node.

The paging system on a node caches copies of pages that have been transferred to that node, so that any subsequent reference to the same page is very fast. A concurrency checking mechanism ensures that the cached pages are valid upon subsequent reference.

In Domain/OS, both disk and network I/O are implemented by way of demand paging. The file I/O manager maps file pages into the virtual memory of a process. When a program attempts a read or write operation, the file I/O manager copies the pertinent data, starting at the current seek pointer location, from the place in the process's address space where the file has been mapped, to the user's program buffer. If the data requested is not in real memory, a page fault occurs.

Scaling to Many Machines and/or Users

Unlike traditional UNIX implementations, Domain/OS anticipated the availability of high-speed local area networks from the time of its initial design. We felt that to truly exploit the current and future capabilities of such networks, the distributed file system supported by the system would have to adhere to two principles:

- The design must scale well to very large networks, consisting of thousands of nodes.
- Sharing of data without prior arrangement must be the default.

Simple Administration in Large Networks

The first principle implies that administering the distributed file system would have to be extremely simple. Adding a new node or a new user to the network must be as simple as adding a new user on a timesharing system. The ability to have new nodes join the distributed file system must not be limited by fixed size mount tables, nor must other machines have to perform any explicit action to access a new node.

In Domain/OS, when a new node joins the network, it issues only one command to make its presence known to a network-wide naming server. Thereafter, other nodes automatically learn the address of the new node when they attempt to refer to it by name.

One of the most important benefits of these design decisions is that diskless nodes can be used and administered very simply and flexibly. Any workstation can boot diskless off any disked workstation in the network at any time, without prior arrangement.

Nor does a diskless workstation require its own file system partition on its partner. Instead, it shares the root volume of the partner. Like every other node, it sees the same view of the name space and has full access to the file system, no matter who its partner is.

One other point concerning protection is worth noting. In supporting very large user communities, one soon finds that the simple owner–group–world protection scheme offered by the UNIX system is not always flexible enough to carefully control the sharing of data. One often wants to grant or remove rights for certain persons or groups of people. Or one wants to have the protection placed on a file be determined not by who is creating the file, but by what directory it is being created in.

To address this problem, Domain/OS optionally allows the extension of the UNIX protection mechanism with Access Control Lists [3]. ACLs are ordered lists of subjects (persons, groups, and organizations) and the rights granted to those subjects. Protection inheritance for newly created files can be specified on a per–directory basis, and can be set to be "from process" (traditional UNIX semantics) or "from directory," where each directory has associated with it the initial protections to be applied to new files.

If a user or system administrator makes no special arrangements, the default is for the system to provide nothing more than standard UNIX protection semantics. If the extra flexibility of ACLs is needed however, it can be used without needing to modify existing UNIX programs to deal with the new protection information. In other words, ACLs interact well with such UNIX system calls as **open**, **creat**, **stat**, and **chmod**.

In addition, rather than having a copy of an **/etc/passwd** file on every node, a unified server–based account registry covers an entire network [6]. Users therefore have a network–wide identity. Adding a new user simply involves sending a series of messages to the registry servers via an editing tool. No files need be copied to every node in the network to make the new user known.

This illustrates another implication of the desire to support large networks: the only viable way to deal with network–wide databases and services is to get at them through servers (possibly replicated), and if necessary, to keep local caches of those databases on each node. It is not practical to distribute full copies of such databases to every node each time the databases change.

Shared Data

The second design principle of the Domain/OS distributed file system, sharing by default, implies a global uniform name space. The name space of the distributed file system appears to users like that of a giant timesharing file system. It is a traditional UNIX hierarchical name space, except that absolute pathnames can begin with the name of the network root (called //). It is also possible to express pathnames relative to the root of the local node (the / directory).

The network root is a database maintained by the network naming server. For efficiency, each node has its own cache of the network root. When the system tries to resolve a pathname beginning with //, it first looks in the local cache; if the node name is not found there, it consults the naming server.

The important point is that no matter what node on the network a user sits at, a given file has the same pathname. In order to share public files, or other users' private files, no prior arrangements (such as mounting file systems) need be made. Access is controlled by the normal file system protection mechanisms, which apply network-wide.

After using this facility for a while, the virtues of sharing become apparent. For example, it is routine within Apollo's 2500-node corporate network (spread over eight buildings and two states) for a user to send a mail message to a group of users that says "Please look at my proposal in **//mynode/user/proposal**, and record your comments in the file **//mynode/user/comments**."

Recipients can then simply move a cursor onto the first pathname in the mail message, click a mouse button, and open an edit window displaying the contents of the file. They can then point at the second pathname, click, and edit the comments file. This ease of use greatly improves the efficiency and bandwidth of communications, especially among geographically dispersed groups.

Extensibility by Users

An important aspect of an open architecture is the ability of system builders and users to extend the system's functionality. Such extensions represent as much of an investment as applications development.

In traditional UNIX systems, functionality is extended by modifying the kernel. The most common kind of extension is the addition of device drivers for new devices. This requires a certain level of expertise in kernel internals and, usually, a set of kernel source code.

More seriously, there is no guarantee that when the vendor supplies the next release of the operating system, the user's extensions will still work. At the very least, any kernel changes will have to be reintegrated with the source and/or binaries provided by the vendor.

Open System Toolkit

Domain/OS solves this problem by its adherence to object-oriented structuring techniques. Each object in the file system is marked with a unique type identifier (type UID). I/O on each object type is handled by a different manager. When an object that is not one of the built-in types is opened, the device-independent I/O subsystem (the switch) locates the manager for that type, and dynamically loads it. It then calls a manager initialization routine which exports a vector of procedures implementing operations like **open**, **close**, **read**, **write**, and **seek**.

If a manager supports other semantics, say those of a tty, it may export other sets of operations, such as those for setting erase and kill characters or for setting raw and cooked modes. Subsequently, all operations on the new file descriptor will be switched through to the manager's exported procedures.

Apollo documents the interfaces expected by the switch and guarantees that they'll remain the same from one software release to the next. These published interfaces and the tools for defining new types are known as the Open System Toolkit. Open System Toolkit

managers run in user space, and therefore have all the above-described facilities available to them for shared memory and synchronization.

Developers can use the Open System Toolkit to add interesting new file types to the system, while applications that use these new types continue to work without change. A simple example might be a circular log file type. Another useful type might be one which maintains all the versions of a text file under source version control. When an application opened a text file under version control, it would read the most recent version of the text. This obviates the need to perform a separate "fetch" operation before an application can look at a source module. The GPIO system can also combine with the Open System Toolkit to give standard access to new devices.

Objects

An object is a container, and each object has a unique identifier called a **xoid** associated with it. The operating system does not care what the contents of an object are. Objects that the system manipulates can "contain" such diverse things as ASCII files, printers, and tape drives.

Applications programs generally want to perform certain kinds of functions on objects, like reading, writing, creating, and deleting. Traditionally, in the case of peripherals, an operating systems programmer would write a device driver for a new class of device that the system would support, add the driver to the operating system source code, and then rebuild the system software.

In Domain/OS, subroutine libraries supplied with the operating system perform basic operations like reading and writing on their associated type of object. These subroutine libraries are called type **managers**. Each type manager supports certain **traits**. A trait is an ordered set of the operations that can be performed on an object.

Users can add new types of objects and their associated managers with the Open System Toolkit. Rather than having to rebuild and reboot the operating system, the user can install a new type manager with a single shell command.

Type managers can implement standard operations like read and write in any way appropriate. New system functionality can be added without disturbing system operation. Thus, the new functionality is immediately available to all programs.

Not all functions need be implemented for every type manager, of course. For example, the type manager for a line printer would probably not implement a function to move a file pointer.

Type Managers and Traits

The subroutine that implements the functions for a given object type is called a **type manager**. In Domain/OS, there are managers for physical resources like disks, network controllers, and memory, as well as for abstract concepts like files, processes, and address spaces.

Since an object is only a container, each type of object on the system looks the same to an applications program. Because of this, the application can contain, say, a **read** call that is compiled into the program. At execution time, the user can redirect the program and the call to operate on any kind of object. For example, if a program attempts to **open** an object of the ASCII file type, the ASCII file type manager performs the appropriate functions. Thus, developers can create a group of general-purpose utilities that operate on all object types instead of creating and maintaining programs for each individual type on the system.

Much of this is true of traditional UNIX implementations, but few let you easily add new types to the system. As a result, in most UNIX implementations, users cannot be sure that a new type added at one release level of the operating system will still operate correctly with a new release. Under Domain/OS, new types will work correctly with later operating system releases.

A **trait** represents a certain behavior that an object supports. Each trait is an ordered set of operations. An object supports a trait if the object's type manager implements the operations that define the trait. For every trait that a type manager supports, the manager provides a list of pointers to procedures that implement the operations in the trait. For further details on the trait/type system, see the paper *An Extensible I/O System* in Part 2 of this book.

How Type Managers Are Loaded

When an applications program written using the Open System Toolkit attempts an operation on a pathname, the system resolves the name of the object into a xoid and then ascertains the object type that the xoid identifies. If the manager for that type is not currently loaded, the system loads it from the trait/type database (which tracks which type managers are currently loaded) into the address space of the process that requested the object.

A type manager is loaded only when a process demands it. Unlike a device driver, the manager does not need to be bound into the operating system in advance. Thus, a type manager doesn't consume system resources until it actually is loaded.

Extended Naming

One of the traits that a type manager may optionally support is known as **extended naming**. When the system name resolver encounters a pathname component that is not a standard directory (called an "extended naming object") but unresolved pathname text still remains, the rest of the text is passed to the **open** operation of the type manager that supports the extended naming object type. The interpretation of the residual text depends entirely on how the manager is written.

For example, a manager to control source versions could be written in such a way that trying to access the pathname **/progs/main.c** would give the application the most current version of the source file for editing and compiling, while specifying the pathname **/progs/main.c/7** would provide version number 7, and naming **/progs/main.c/-1** would provide the penultimate version.

A more typical use of extended naming is to provide gateways to non-Apollo file systems. For example, a Network File System (NFS*) mount point could be an extended naming object supported by the NFS type manager. All pathname text beyond the mount point would be interpreted by the NFS type manager as being relative to the root of the remote file system mounted at that point.

---

* NFS is a trademark of Sun Microsystems, Inc.

This mechanism provides a relatively simple way for users to build gateways to any kind of foreign file system to which they might want transparent access. All that is necessary is to write the type manager and a server to run on the foreign machine to handle remote file system requests.

Network Computing System

Another important way in which users can extend the functionality of the system is via the Network Computing Architecture (NCA). The Network Computing Architecture is a framework for developing distributed applications. It allows optimal use of computing resources and even allows users to take advantage of parallel processing and specialized hardware. It also provides a way for machines to advertise idle computing facilities.

The Network Computing System (NCS) is a portable implementation of the Network Computing Architecture that runs on UNIX systems and other systems. It provides tools, servers, and information brokers to develop distributed applications. NCS extends the Domain/OS concept of objects to include replicated objects — copies of a single object, all with the same unique identifier. These replicas are weakly consistent, that is, all copies of an object may not always be in an identical state. However, a weakly consistent replica is more likely to be available than a strongly consistent replica (one of many copies guaranteed to be identical).

NCS consists of two major pieces, **remote procedure calls** and **location brokers**. Remote procedure calls operate in programs as local procedure calls. However, they allow a process on one machine access to data, programs, or devices on another machine, by causing a server on the remote machine to execute subroutines on the program's behalf. Remote procedure calls enable an application to be run in a distributed fashion, without a programmer having to rewrite the application.

In a network where considerable distributed processing occurs, applications must have access to information about available machines and CPU cycles. Location Brokers store and administer this type of information, allowing NCS applications to bind dynamically to different services without being rewritten. Location Brokers can be replicated easily, to provide reliability. Many services augment the

NCS tools, including license servers that provide more flexible software licensing.

NCS is portable and allows system software and applications to obtain services that are implemented on both Apollo's and other manufacturers' machines. Because these services may be impossible or expensive to provide locally, NCS truly extends the power of the local system.

In addition, NCS increases homogeneity in system administration. For example, Apollo uses a distributed registry — a database that holds all user account and group and organization information. This distributed registry is portable to any UNIX system. Thus, a mixed-vendor network can handle user accounts under a single unified mechanism.

Support for Multiple Operating System Environments

The state of affairs in the UNIX world today is such that there are really two de facto standards: AT&T's System V and Berkeley's VMUNIX. Until a single standard is accepted by an overwhelming portion of our customer base, Domain/OS will include and support both variants.

We view Domain/OS as a single operating system with a very rich set of system services (primitives). Some of these primitives are found in other UNIX systems (system calls and library routines). Other interfaces are unique to Domain/OS.

Programmers are free to use whatever primitives they like, but programs designed to be portable to other UNIX systems should restrict themselves to the standard UNIX interfaces. Programs that use only the proprietary Domain/OS primitives are sometimes known as "Aegis programs," mainly because they use calls that have historically been documented in the Aegis reference manuals.

In order to provide maximum portability of software from Domain/OS to other Berkeley or System V UNIX systems, Apollo provides

two complete and separate UNIX environments, rather than a hybrid of the two. Any workstation can have one or both UNIX environments installed, and users can select which environment to use on a per–process basis.

Two key mechanisms support this facility. First, every program can have a stamp applied that says what UNIX environment it should run in. The default value for this stamp is the environment in which it was compiled.

When the program is loaded, the system sets an internal run–time switch to either **berkeley** or **att**, depending on the value of the stamp. Some of the UNIX system calls use this run–time switch to resolve conflicts when the same system call has different semantics in the two environments.

The other mechanism is a modification of the pathname resolution process, such that pathname text contains environment variable expansions. For example, the pathname **/tmp/***$(ABC)* would expand to **/tmp/Ex00324**, if the environment variable *ABC* had the value **Ex00324** in the current process.

When UNIX software is installed on a node, the standard trees (**/bin**, **/usr**) are installed under directories called **bsd4.3** and **sys5.3**. The names **/bin** and **/usr** are actually symbolic links dependent on the value of an environment variable named *SYSTYPE*. That is, **/bin** is a symbolic link to */$(SYSTYPE)/bin*. When the program loader loads a stamped program, it sets the value of *SYSTYPE* to either **bsd4.3** or **sys5.3**, according to the value of the program stamp. Therefore, a program that refers to a name in one of the standard directories will access the correct version for its environment.

Support for Multiple Processors

The new generation of workstations includes multiple tightly coupled CPUs in one system. However, the UNIX system was originally designed to run on single processors. Jans points out [4] that a major problem is not just to fix the way critical sections in the UNIX kernel are protected for a multiprocessor, but simply to identify all the critical regions. This is a problem because a UNIX kernel process assumes that other kernel code will not refer to kernel data structures unless the kernel process explicitly gives up the processor through a call to **sleep()**, or unless an interrupt occurs.

Single-processor UNIX systems handle the interrupt case by temporarily raising the processor priority to a high enough level to prevent any interrupt handlers from altering a critical data structure. In a multiprocessor system this is not sufficient, since multiple kernel processes could be running simultaneously. Therefore, access to data structures inside critical sections must be explicitly synchronized with a mechanism like semaphores or eventcounts.

In Domain/OS, eventcounts form the basis of synchronization among processes. Every critical section is protected by a mutual exclusion lock/unlock pair. When a process reaches a critical section, it must be able to acquire the lock before it can continue.

Conclusions

Domain/OS is designed to meet the challenges offered by today's technology, and to solve the problems faced by today's users and system developers.

Foremost among these challenges is the need to share computing resources of diverse origins as seamlessly as possible. Meeting this need implies strong support for existing standards and pushing beyond the standards where they fall short of users' needs. At the same time, part of the Domain/OS philosophy is to take advantage of the inherent homogeneity among Apollo systems to provide high levels of performance and transparency. A prime example of this is the Apollo distributed file system with its uniform name space and simple administration.

An important goal of Domain/OS is to provide flexibility for future changes and expansion, both by Apollo and by its customers. To this end, the system employs the principles of object-oriented design, in which pieces of functionality can be replaced with little or no impact on other portions of the system.

To provide the openness necessary for users to adapt the system to their own needs, the same mechanisms used internally to structure the operating system are all made available to users.

In these ways, we expect that Domain/OS will prove to be a superior base for meeting the challenges of the next several years.

References

[1] M. Bach and S. Buroff. Multiprocessor UNIX Systems. *Bell System Technical Journal*, 63, No. 8, pp. 1733–1749. 1984.

[2] T.H. Dineen, P.J. Leach, N.W. Mishkin, J.N. Pato, and G.L. Wyant. The Network Computing Architecture and System: An Environment for Developing Distributed Applications. *Proceedings of the Summer 1987 USENIX Conference*, pp. 385–398. 1987.

[3] G. Fernandez and L.W. Allen. Extending the UNIX Protection Model with Access Control Lists. *Proceedings of the Summer 1988 USENIX Conference*. 1988.

[4] M.D. Janssens, J.K. Annot, and A.J. Van De Goor. Adapting UNIX for a Multiprocessor Environment. *CACM*, 29, No. 9, pp. 895–901. 1986.

[5] D. McCracken. Extensions to UNIX Signal Functionality for Modern Architectures. 1988.

[6] J.L. Pato, E.M. Martin, and B. Davis. A User Account Registration System for a Large (Heterogeneous) UNIX Network. *Proceedings of the Winter 1988 USENIX Conference*, pp. 155–161. 1988.

Extensions to UNIX Signal Functionality for Modern Architectures

by

Dave McCracken

Abstract

While signals have long been a useful feature in UNIX systems, some modern innovations, such as shared global libraries and user space shared semaphores, have made the current implementation inadequate. In this paper we look at two extensions to the signal mechanism that provide the extra functionality required.

A simple user space inhibit/enable mechanism provides cheap protection from signals for critical sections of code, allowing some code that is only in the kernel for this protection to run in a shared library.

A nested cleanup handler allows reliable cleanup of semaphores in shared memory, as well as other resources the kernel may not otherwise be able to restore on process death.

Introduction

Signals were originally developed as a way of prematurely terminating a process, either because of some fault the process generated or by some external event. After the ability to trap these signals was provided, the meaning expanded to include other events, such as timers, suspend/resume mechanisms, and software I/O interrupts.

Some of the flaws in the early implementations, such as the inability to protect critical sections of code without missing the signal, were addressed by Berkeley and, later, by AT&T with the ability to delay a signal's delivery.

New architectures now gaining popularity are again making the existing signal model inadequate. With the introduction of semaphores in shared memory comes the need for cheap concurrency control, along with the need to clean up the locks if a fault occurs. Global shared libraries are taking over some of the functionality previously found in the kernel, which requires protection from asynchronous events.

In this paper we examine some of the solutions implemented as part of the Apollo Domain/OS implementation of a UNIX operating system. Cheap nested cleanup handlers solve the semaphore problem, and a fast inhibit/enable mechanism protects the libraries during critical code.

Global Shared Libraries

The Problem

In Apollo's Domain/OS, the use of global shared libraries has allowed some of the functionality traditionally found in the kernel to be moved into user space. In fact, a large portion of the kernel commonly considered part of the "system call" environment has been moved to libraries, with more primitive calls into the kernel for things that really require the additional privileges the kernel provides.

A problem with this model is that, while this code does not really require kernel protection and privileges, it does require protection from asynchronous interrupts, or signals. While this can be provided by a kernel primitive, in many places performance is also important and another system call, with its attendant overhead, is too expensive.

The Solution

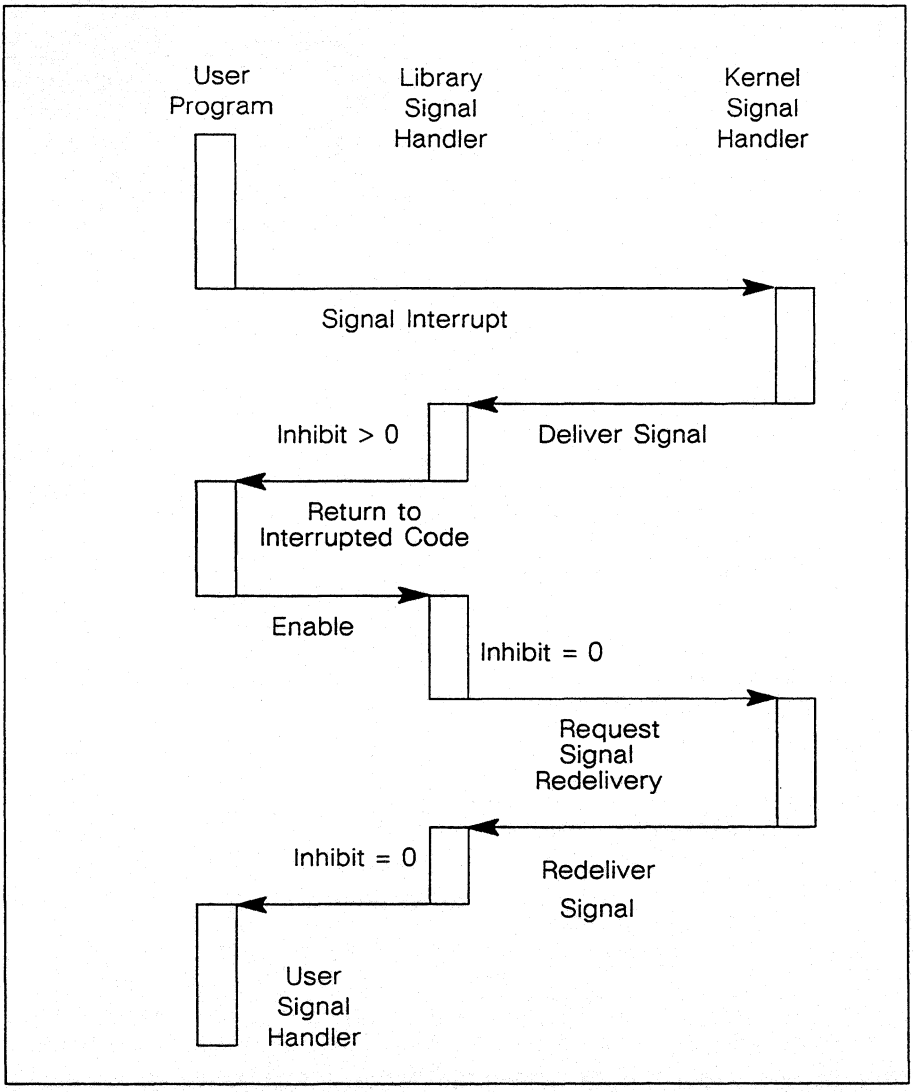
One area that has benefited from moving functionality into libraries is the UNIX signal subsystem. While parts of the delivery mechanism must be in the kernel, particularly the decision whether to interrupt the target at all, much of the user–supplied handler information can be stored in user space and only a subset of information forwarded to the kernel portion so it can make its initial delivery decision.

The two parts of the handler are tightly coupled and include a handshake mechanism, so the kernel handler does not consider the signal delivered until the library code acknowledges receipt of the signal. This occurs just before the user–specified handler is called.

With this model, the problem of inhibiting the delivery of signals becomes simple to solve. We have created an inhibit counter in user space that may be incremented and decremented by calls to a pair of small library routines. Since there are no kernel traps involved, these calls are very fast, typically less than ten machine instructions. This allows even commonly used code to easily protect its critical code sections without seriously degrading performance.

The inhibit counter is integrated into the signal handler at the point where the signal is delivered to the library handler in user space. The first operation performed in the handler is to check this counter. If it is non–zero, the handler sets a flag indicating a signal is pending and returns to the interrupted code without delivering the signal any further. Since the kernel handler is waiting for the library handler to acknowledge receipt of the signal, it is left pending, and other signals are blocked until this condition is cleared.

When the inhibit counter is decremented to zero and a signal is pending, the library requests a re–signal from the kernel handler. This time the check in the library handler will pass and the acknowledgment will be sent, allowing the kernel handler to mark that signal as delivered and no longer pending (see figure).



Signal Delivery Sequence

Semaphores in Shared Memory

The Problem

Some form of handler is necessary to trap any fault or signal that the process may receive while the semaphore is locked. The signal handling mechanism provides some of this functionality, but falls down in several areas:

- Each signal that may be received must have a handler set for it.
- Cleanup needs to be done even for "uncatchable" signals, i.e., SIGKILL.
- It does not allow nesting of semaphore locks in different sections of code with separate cleanup handlers.

The Solution

Apollo's solution to the problem of cleaning up user-space semaphores, along with other important user state, is to allow the user to set dynamic cleanup handlers. These handlers are similar to **setjmp**, in that the user specifies the save area and that the return status indicates whether it was set or invoked.

The cleanup handler differs from **setjmp**, however, in two important ways. First, these handlers are integrated into the signal handler and are automatically invoked whenever there is no user-specified handler for a signal that would cause process death. Second, the cleanup handlers can be nested. Setting a new handler when there is one already set behaves like a stack push, making the new handler the first one invoked. The previous handler is re-established as the current handler when the new handler is invoked or released.

When a handler is invoked, it returns again from the "set" call, with the return status indicating what signal or error caused the invocation. The user program then has several options. If it was an error the routine was expecting, it can re-establish the cleanup handler and continue or return synchronously to its caller. Otherwise,

the convention is for the routine to call a special routine that will invoke the next handler in the chain, passing the status on.

If a routine that set a cleanup handler completes normally, the handler must be explicitly released before the routine returns to its caller.

An additional guarantee that all cleanup handlers are run is that the exit call invokes the cleanup handlers before the process is terminated. This ensures that any global machine state is reset no matter what path the process takes when it dies.

To ensure that the cleanup handler chain is kept intact during a **longjmp**, all cleanup handlers set between the **longjmp** call and the code where the **setjmp** was done are invoked. A special status is used for these cleanup handlers to indicate that a **longjmp** is in progess.

Conclusion

When new functionality is added to a UNIX system that interferes with the original signal model, we have shown that it is possible to extend that model in a compatible way to support the more complex interactions required by the new features. This issue will gain even more importance in the future as processes rely more on features that cannot allow uncontrolled signal interrupts or process death.

Detailed Signal Delivery Sequence

- The signal is passed to the kernel with **kill()**.
- If the signal is explicitly ignored or "ignore by default," it is dropped, else it is set pending.
- As soon as the signal is not blocked, the target process is interrupted.
- If a signal stack is specified, the user stack pointer is switched.
- The kernel signal delivery pending flag is set.
- The library signal handler is called.
- If the inhibit count is non-zero, the library signal handler flags that a signal is pending and returns to the interrupted code.
- When the enable is called that sets the inhibit count to zero, the kernel is notified and the pending signal is re-delivered.
- When the inhibit count is zero, the library handler acknowledges delivery of the signal, and the kernel signal delivery pending flag is cleared, freeing the kernel handler to deliver another signal.
- The user signal handler is checked and, if it is not default, it is called. When it returns, if it does, the signal is dismissed and the process resumes where it was interrupted.
- If there is no user signal handler, the first cleanup handler in the chain is invoked.

Example of Using a Cleanup Handler

```
write_shared_memory(address, value, sem)
int *address;
int value;
semaphore *sem;
{
   pfm_$cleanup_rec cl_rec;
   status_$t status;

   /* trap any errors */
   if ((status = (pfm_$cleanup(cl_rec)) !=
pfm_$cleanup_set) {
      mclear(sem);
      /* check for bad address fault */
      switch (status.all) {
        case mst_$illegal_address:
          (status.all == mst_$illegal_address))
         return(1);   /* the write failed */
      pfm_$signal(status);  /* an anonymous prob-
lem, pass it on */
   }

   mset(sem, 1);          /* lock the semaphore */
   *address = value;      /* do the write */
   mclear(sem);       /* clear the semaphore */

   pfm_$rls_cleanup(cl_rec, status);
   return(0);             /* the write succeeded */
}
```

Shared Program Libraries — The Domain/OS Library Model

by

Bryan Douros

Introduction

At one time, it was necessary for each program to include everything it needed to solve a particular problem. The scope of problems has expanded, however, and collections of library subroutines and standard system functions have allowed applications to grow into very complicated programs. Programmers draw on graphics, communications, database, and other services to make applications more usable, but increased functionality also makes them larger and more difficult to maintain.

Modular programming practices and higher-level languages enable programmers to maintain their applications more easily. Collections of related routines are combined into libraries of services, so that many separate applications can now draw on the same services. In most systems, library and application routines are bound into a single program at link time. While this has extended what applications can do, there are several weaknesses to this scheme. Linking the same routine to many applications requires the duplication of the routine in each application program. This uses more disk space and increases the working set of the applications.

This scheme causes distribution problems, too. No matter how modular an application is, bug fixes and performance improvements require relinking and redistributing all the programs that use these fixed routines. Distribution and maintenance become even more complicated when different libraries are managed by different groups, organizations, or even different companies.

To overcome some of these weaknesses, Domain/OS provides the ability to share libraries among programs with the following effects:

- Linking to libraries can be deferred until program execution.
- Libraries can be distributed independently of application programs.
- The same libraries can be shared with many concurrently executing programs.

Domain/OS uses shared, direct paging of program code so as to only have one copy of the code in memory. It also uses indirect linkages to shared library references to minimize the relocation necessary at load time and position-independent code so that shared libraries can be loaded into any address space. Domain/OS also uses a table of known globals to make linking the correct routines easy and extensible.

Indirect Linkages

Domain/OS compilers generate linkages to shared library references by way of indirect pointers, so that read-only code doesn't require relocation. Procedure references call through transfer vectors (which are jump instructions) stored in read-writable sections to the shared library routine, and are relocated at load time. Data references use indirect references via an absolute pointer, also stored in read-writable sections and relocated at load time. This allows direct paging of shared read-only program code and minimal relocation of linkage addresses at program loading time.

Position-Independent Code

Domain/OS compilers are capable of generating position-independent code (PIC), so that the bulk of a program may be loaded at any free virtual address space the process has. Thus, a minimum of relocation needs to take place at load time. The compilers use program counter (PC) relative branches and instructions, and take the indirect linkages to the extreme, so that all external linkages are

indirect. This allows shared libraries to be loaded quickly into the address space, relocating only the external linkages.

Global Address Space

Domain/OS divides the user address space of a process into two parts, the first of which is private to the process and the second of which is global to all processes at the same virtual address in every process. This feature allows shared libraries to be loaded once into address space, after which every process has access to them at no cost. Important libraries that are used by every process are dealt with in this way.

The private address space is user accessible and manageable. The global space is managed by the process manager (PM) and loads suitable shared libraries into the process's address space.

Known Global Table

The known global table (KGT) is a system table that keeps track of the symbols exported by all the shared libraries known to processes. Each process maintains its own logical KGT, but by maintaining a global KGT (containing symbols known to all processes) and a private KGT (symbols added to this process), the system can maintain a symbol table that is small but can be completely customized.

When a program with an unresolved external reference is loaded, the KGT is consulted and references that point to shared libraries are linked to those libraries. References to a shared library that has not yet been loaded either cause the library to be loaded and linked immediately, or cause it to be loaded when the program actually runs and attempts to call into that library.

The KGT actually consists of three tables:

- The first table, the private KGT, contains symbol entries currently loaded into the private A space of the process. Symbols in this table are private to this process. Entries contain the ASCII name, the address of the symbol, and information as to the type of symbol (function, data, or common). The table is indexed by a hash of the symbol name.

- The second table, the known library table (KLT), is a table of symbol entries that are in shared libraries. Symbols in this table are private to this process and are entries containing the ASCII name, a reference to the shared library so that it can be loaded when needed, and a set of flags indicating the behavior that this library should have (loaded on program execution or loaded at program reference). The KLT is inherited by child processes (created via **fork**, **exec**, or **pgm_$invoke**). This is useful for building a set of shared libraries to be used in the current environment.

- The third table, the global KGT, is a table of public symbol entries. These are symbols known to all processes. The global KGT contains all the symbols of libraries loaded into the global space and symbols of shared libraries that are to be loaded into the process's address space as needed. Entries contain a compressed form of the name (32 bits), its type (function, data, or common), either an address of the symbol in global space or information similar to that in the KLT that describes the library and when it should be loaded. This table is also indexed by a hash of the symbol name.

  The global KGT is constructed at system boot time and is very large, since it contains all the symbols known to all processes. It can use a highly compressed form of the name, since it is built all at once and compressed symbol collisions are dealt with specially. This allows a table that can be searched very quickly to find symbol names.

The KGT is searched in the following order: the private KGT, the KLT, then the global KGT. This allows private symbols to supersede symbols defined in the public space to allow library development or private program tuning, while leaving the public libraries unchanged. The behavior of duplicate symbols in the same table is undefined.

Global Libraries

Global libraries are a special case of shared libraries. As described above, libraries loaded into global space are visible to all processes and have no cost at program load time. To accomplish this, Domain/OS loads all global libraries at system boot time.

Global libraries have four classes of storage: read–only procedure text, global storage (which gets turned into read–only after library initialization), dynamic storage (run–time stack), and process–private storage (which is zeroed at the beginning of each process).

Pure functions would be very easy to make into a global library. If a library has statically initialized state, it is more complex, but the benefits of global libraries are great and most Domain/OS libraries are global libraries.

Dynamic Linking

Domain/OS supports a limited form of dynamic linking. A program loaded with references to undefined procedures has a special jump vector created which points to a dynamic link snapper routine. (References to undefined procedures were either not found in the KGT or found and specified as load–on–reference.)

During execution, a call to this undefined symbol is vectored to the dynamic link snapper routine and is passed the ASCII text of the symbol name. It looks for the symbol in the KGT. If the referenced library is not loaded yet, it is loaded, the jump vector is patched to the symbol, and the routine is executed. Future references have no extra performance cost.

If the symbol is not found, a fault is generated to the program. This can be useful for programs that have many modes and don't always execute all of their code, for it can defer the expense of loading shared libraries that might not be used at all. Unfortunately, this works only for procedure references and not for data references.

System Configuration

Domain/OS reads the file **/etc/sys.conf** at system boot time to determine which shared libraries should be loaded globally, which should be loaded on program execution, and which should be loaded dynamically, at program call time. It contain entries of the following form:

lib[rary] *shared_library_pathname flags*

The *shared_library_pathname* may either be relative to the **/lib** directory or an absolute pathname. There may be zero or more flags, separated by spaces or commas (,). The flags are **global, dynamic,** and **optional**.

Global means that the library is to be loaded into global address space. Dynamic means that the library loading should be deferred until the actual execution of the routine utilizing the dynamic linking feature. Optional suppresses the error message if the library is not at the specified pathname.

If global or dynamic is not specified, the library is loaded at program execution time. Because global address space is a limited resource on some older Apollo workstations, we included the following flags for compatibility:

not_16mb_va
do not load this global on a 16MB virtual address space machine (DN300, DSP80)

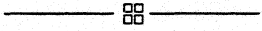
not_64mb_va
do not load this global on a 64MB virtual address space machine (DN330, DSP90, DN5x0, some DN3000)

These commands can set by the workstation's administrator to specify which shared libraries will be in the public KGT, the set of symbols known to all processes. Changes to the **sys.conf** file changes the public symbols only at the next system boot time. Global libraries must have no duplicate symbol definitions and only external references to other global libraries.

Private Configuration

There are several methods whereby users can specify additional shared libraries. Shells distributed with Domain/OS have two added internal commands, **inlib** and **llib**. The **inlib** command adds a shared library to the KLT for the current process. The **llib** command lists the shared libraries in the current process's KLT. Currently, all shared libraries added with **inlib** are loaded at program execution.

Programmers can also specify at program link time what libraries are needed for the program. The **/bin/ld** and **/com/bind** tools have options which allow you to include libraries. When the program is executed, those shared libraries are loaded with the program.



The Domain/OS Input/Output System

by

David A. Buckle

Abstract

While Domain/OS offers several novel features in the area of extensible I/O [3] and user-level device drivers [1], there is still a need for basic I/O services within the kernel. Continual development of new workstation platforms for Domain/OS requires continual enhancement of the kernel I/O system to support these new platforms.

This paper briefly describes the components of the kernel I/O system, and identifies the approaches taken to reduce development costs associated with new peripheral support.

Introduction

Domain/OS is the operating system kernel that runs on all Apollo workstations. It provides support for a variety of different peripherals across the complete range:

- Winchester Disk
- Floppy Disk
- Mass Storage Module
- Apollo Token Ring
- IBM Token Ring

- ETHERNET*
- Cartridge Tape
- Color and Monochrome Monitors

One of the I/O system design goals was to simplify the development of new drivers. This has been achieved by reducing the interaction between driver modules and the other components of the operating system, and by providing common interface modules for the various classes of device present in the system. The next section describes this structure.

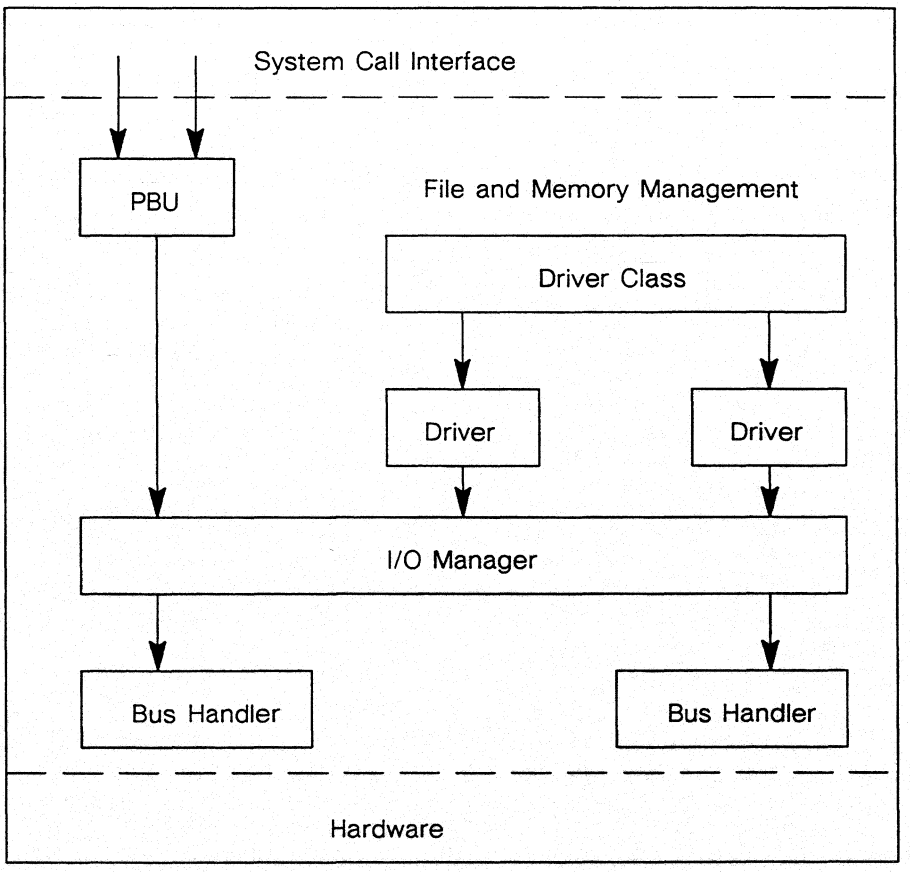
Overall Structure of the I/O System

The I/O system is implemented in several layers:

- Device Class
- Device Drivers
- Resource Manager
- Bus Interface

The following figure shows the relationships among layers.

*ETHERNET is a registered trademark of Xerox Corporation.



Relationships of Layers in the I/O System

While lower levels normally provide services to the layers above, the organization is not strictly hierarchical; class modules can provide device-independent routines for their drivers, and device drivers have access to the control and status registers (CSR) of the peripheral controllers without calling resource level routines.

Device Classes

Domain/OS associates each device with one of a number of device class modules, based on the functions and use expected of the device.

All I/O requests from the upper layers of the operating system are made through the class module, which then vectors these calls to the driver supporting the requested device.

Each class provides a set of procedural interfaces to the upper levels of the operating system, tailored to the class, and not necessarily the same as the interfaces provided by other classes. This approach contrasts with the UNIX device model where all devices fall into one of two types, block and character, all block devices having one set of interface routines, and all character devices having another.

Thus, for example, the network class can provide procedural entry points that reflect the functions expected of network devices, whereas UNIX network drivers are forced to map these network specific functions onto a generic entry point such as **ioctl**.

The device class modules can also provide device independent routines for use by the driver modules. For example, the disk class provides routines to sort queues of transfer requests, enabling disk drivers to easily manage head scheduling.

Device Drivers

All Domain/OS device drivers have the same basic structure; a set of entry routines called by the class module and run in a process context, and one or more interrupt handlers that execute asynchronously.

Driver entry routines are always called from the class module. Synchronization between the in–process driver routines and the asynchronously called interrupt handlers is accomplished via an eventcount mechanism [2]. Mutual exclusion locks are used to protect common data structures.

Resource Manager

The I/O resource manager module provides a central place for the management of the hardware and software resources required by the device drivers. These resources include hardware interrupt vectors and software memory regions.

The module provides services to both system devices and also to the Peripheral Bus Unit (PBU) support module, which in turn provides services to user-level GPIO device drivers.

Most resources are acquired during system initialization rather than being statically reserved at system generation. Devices that are not present in the hardware configuration do not lock up any hardware resources, which are therefore available for user-level devices. This approach to resource allocation works well with the fixed configuration aspects of Domain/OS, and allows the same version of the operating system to be used on different hardware configurations without imposing restrictions on user-level drivers.

Bus Interface

Bus modules are provided for the various I/O buses present on Apollo hardware. All bus modules provide the same set of service routines.

Initialization

During system initialization, the I/O resource manager is called to perform I/O-specific initialization tasks. Associated with each device controller in the system is a device descriptor data structure that describes the hardware characteristics of the controller, and contains a pointer to the supporting device driver initialization routine. The I/O resource manager traverses the controller data structures, and calls each initialization routine.

Each driver checks for the presence of its controller, and if it is present, the driver registers itself with its known class module, and with the I/O resource manager.

Class registration enables the driver to pass an entry point vector (EPV) to the class module. All calls from the class module into the device driver are made through this EPV, which contains pointers to all the exported procedures.

Each device class is free to define the set of entry points most relevant to the class, when then define the formal interface between the class and any driver supporting that class.

During the initialization phase, the driver also registers its interrupt handlers with the I/O resource manager. Thus, the only entry point that needs to be directly exported from the driver module is the name of the initialization routine.

Configuration

Configuration of device drivers within an operating system can be considered in three separate stages:

- Specifying which device drivers are required in the system.
- Specifying the hardware characteristics of the devices.
- Associating a logical name to be used by the remainder of the OS and/or the user programs with a specific hardware device.

Domain/OS traditionally runs on a limited set of hardware configurations and does not provide any means for on-site tailoring of the kernel. All devices needed by the system that could be present on any hardware platform must be bound into the operating system executable.

While this does restrict the system somewhat, it also has these benefits:

- There is no need for a local administrator to perform on-site configuration.
- In a local area network, nodes can be booted from any other node holding the appropriate operating system executable without any problems arising from incompatible configurations. Standard naming conventions are used to locate the actual OS file.

Internally, however, the software is organized around the above configuration requirements. The device descriptors used during system initialization effectively provide a central description of the hardware configuration. Each descriptor represents a device; the

descriptor contents are used to specify the hardware characteristics and the device name to be used to address the device.

Summary

Most new devices fit under one of the existing Domain/OS device classes, and hence can take advantage of the generic device support provided by the class modules.

Disk Class to Driver Interface

The disk EPV contains the following entry points:

disk_open_proc	Open a disk device and return drive parameters to the caller
disk_close_proc	Close a disk device
disk_spindown_proc	Spin down active disks (called only from **os_$shutdown**)
disk_revalidate_proc	Check removeable media changes
disk_do_io_proc	Service data transfer requests; multiple blocks may be requested with a single call to this routine; on return it indicates whether all transfers have been completed
disk_error_que_proc	Check error status of non-blocked queued requests
disk_get_stats_proc	Return disk controller statistics (number of reads/writes, etc.)

Network Class to Driver Interface

The disk EPV contains the following entry points:

net_start_proc	Start network service
net_stop_proc	Stop network service
net_load_proc	Load firmware
net_send_proc	Transmit a packet
net_get_stats_proc	Return network device statistics
net_ioctl_int	Control network sends/receives
net_p2_cleanup	Cleanup (process termination)
net_open_svc	User-level open routine
net_close_svc	User-level close routine
net_send_svc	User-level send packet routine
net_rcv_svc	User-level receive packet routine
net_ioctl_svc	User-level control routine

Bus Module Interface

bus_init_proc	Initialize bus module and data structures
bus_alloc_iova	Allocate an address in the bus address space
bus_xlate_iova	Translate bus address into physical page number
bus_define_int	Associate an interrupt handler with an interrupt vector
bus_enable_device	Enable device interrupts for a device
bus_disable_device	Disable device interrupts for a device
bus_device_interrupting	Check whether device has an interrupt pending

References

[1] Apollo Computer, Inc. Writing Device Drivers with GPIO Calls, Order No. 000959.

[2] David P. Reed and Rajendra K. Kanodia. Synchronization with Eventcounts and Sequencers. *Communications of the ACM*. 1979.

[3] Jim Rees, Paul H Levine, Nathaniel Miskin and Paul J Leach. An Extensible I/O System. *USENIX Proceedings*. (Summer, 1986).

Extending the UNIX Protection Model with Access Control Lists

by

Gary Fernandez, Larry Allen

Abstract

The UNIX operating system uses a simple and straightforward model for the protection of objects in the file system, granting access rights based on the ownership of the object. This simple model, however, may not be flexible enough for large user communities, or for communities with complex requirements for controlled data sharing. This paper describes an Access Control List extension to the UNIX protection model, which preserves the behavior of the existing UNIX programming interface while greatly increasing the flexibility of the protection system.

Introduction

This paper describes our efforts to extend the UNIX protection model by adding Access Control Lists (ACLs). We first provide a summary of the UNIX protection model, then describe our extended protection system. We describe how we integrated the extended protection system with the UNIX protection system and discuss some of the details involved in implementing the extended protection system. Examples demonstrate the extended protection system, and finally, we summarize and describe a few lessons we learned.

Overview of the UNIX Protection Model

This section first describes the UNIX Protection Model, and then discusses why extensions are appropriate.

The UNIX Protection Model

The concepts of owner ids and group ids are the basis of the UNIX protection model [7]. Each file and every process has an associated owner id and group id. With BSD4.3 a process may have multiple groups. Processes have two sets of ids: effective ids are used for rights checking; real ids keep track of the true ids for a process for which the effective ids have been altered temporarily.

The owner and group ids for a process are inherited from the parent process. When a file is executed the effective ids for the process may be taken from the executable file, depending on attributes attached to the file: the setuid and setgid bits. System calls may change a process's owner and group ids, but these operations are highly restricted.

The owner and group ids for a file are inherited from the ids of the process that creates the file. BSD4.3 takes the group id in file creation from the parent directory, not from the process. A file's owner or the super–user may change a file's owner and group ids by using system calls. BSD4.3 restricts these changes to the super–user.

Each file has protection information for three categories of users:

- Owner — processes whose owner id matches the owner id of the file
- Group — processes that are not in the owner category and whose group id(s) match the group id of the file
- Other — processes that are not in the first two groups

Protection is checked in the order owner, group, and other; the first matching protection applies.

Protection permissions are read, write, and execute for files, and read, write, and search for directories.

Only the owner of a file or the super-user may change the permissions associated with a file.

Discussion of UNIX Protection Model

The UNIX protection model works well when the set of persons accessing a file is a single person or a single group. However, once it is necessary to allow special access to more than one person or more than one group, the UNIX system is unable to describe this protection. Specially created groups partially solve this, but prove difficult to administer.

Traditional protection alternatives are capabilities and access control lists. Capabilities operate by passing a ticket allowing access to an object [4]. Restrictions may be placed on the number of copies of a capability, whether it may be passed on to other processes, access rights available, etc. Access control lists provide a list of (person, rights) pairs. By comparing entries in the list with the subject requesting access, a matching entry is located. The matching entry then determines the applicable rights.

Access Control Lists are not new. Multics was an early system using Access Control Lists as a basis for its protection system [6]. Aegis, Apollo's proprietary operating system, also used a protection system based on Access Control Lists [3, 5]. Aegis is the predecessor operating system on which our extended protection system is built.

Overview of Extended Protection System

This section describes the extended protection system. It provides an overview, describes how objects are protected, and explains how protections are assigned.

Extended Protection System Basics

The extended protection system provides several additions to the basic UNIX protection:

The number of recognized organizational divisions is extended from two (person and group) to three (person, group, and organization). This more closely matches the real world divisions that occur in large organizations and large networks. The concept of organization allows a more hierarchical partitioning of the users of the system. The network user registry could be partitioned based on organizations, for example. Each process has, in addition to a real and effective person and group ids, real and effective organization ids associated with it. Each file system object also has associated with it an owning organization.

New protection rights are defined which allow persons other than the object owner to change the protection of an object and which prevent an object from being accidentally deleted.

Additional protection entries describing protection for persons beside the protections for the owner, the owning group and "other" are added. These additional entries may be used to provide more granularity in the granting or denial of access.

Each directory contains "initial" protection information which is used to determine the initial protections which are applied to objects which are created in the directory.

Protection of an Object

Each object has associated with it an owner, an owning group, and an owning organization. Rights may be associated with each of the owning fields or with "world," which is used for processes not matching any other protection entries. Because this information is always present and is used as entries in rights checking, we refer to this information as required entries. Optionally, an object may have associated with it extended entries, which are Access Control List entries. ACL entries are pairs of the form (subject identifier, rights). A subject identifier (SID) consists of three fields: the person, the group, and the organization. SID entries in extended entries may have each portion of the SID wildcarded: the character

% is a wildcard with the meaning "match anything in this field." An example of an extended ACL entry might be:

```
john_doe.%.r_d          rwx
```

This means that **john_doe** has read, write, and execute rights if he is a member of any group in the **r_d** organization. Extended entries provide finer control over access to objects.

Each process has both a real and an effective subject identifier. The real SID corresponds to the original identification of the process, while the effective SID may be different as a result of setID programs. SetID programs work as in UNIX protection; an executable object may be marked so that any process running the program will have its effective SID changed while the program is running. Setid may change all three fields of the SID.

The extended model includes the standard UNIX protections of Read, Write, and Execute (Search). We have also added new rights: Protect, which determines whether an SID may change the protection on an object; and Keep, which prevents an object from being deleted or having its name changed. In addition, a required entry may be marked "ignored." This means that the owner, group, or organization information is present, but that rights checking will not use this information.

Rights checking is similar to rights checking in the UNIX operating system. Rights checking proceeds as an ordered walk through the protections specified in the required entries and the extended entries. Given an effective SID, rights are examined as follows:

- If the effective person matches the owner, the owner rights are returned.
- If the effective person matches any extended entries of the form *person.X.X*, where *X* means don't care (these are **person** entries), the rights from the entry are returned.
- If the effective group matches the owning group, the group rights are returned.
- If the effective group matches any extended entries of the form **%.***group.X* (**group** entries), the rights from the entry are returned.

- If the effective organization matches the owning organization, the organization rights are returned.

- If the effective organization matches any extended entries of the form *%.%.org* (**organization** entries), the rights from the entry are returned.

- Rights available to "world" are returned

Note that the algorithm always checks the specific required entry before extended entries of the same type. Also note that there is no requirement at any point that extended entries of a particular type be present; rights checking proceeds to its next phase if none are present. A summary rights mask limits rights available in extended entries. This mask is described in more detail later.

Assigning Object Protection

Protection information is normally applied to an object when it is created. Inheritance from the directory in which the object is created determines the necessary protection information. Each directory has two sets of protection information: initial file protection, which is used when a file is created in the directory, and initial directory protection, which is used when a sub–directory is created in the directory. These initial protections are in addition to the standard protections associated with all objects.

The extended protection system provides several different styles of initial protection (Aegis, BSD4.3, System V.3) to allow different classes of users to have their favorite object protection. Aegis users typically give specific protection information to be applied to all objects created in a directory. System V.3 users expect objects created in a directory to be given protection information based on the effective SID of the process creating the object. BSD4.3 users expect the SID of the creating process and the containing directory to determine the protection for objects created in the directory. These alternatives are all provided by initial protections associated with directories.

File creation can be viewed as taking place in two steps. First, the object is created using default information based on the creating process. The default information may then be overridden by initial file or directory protection information.

Two pseudo rights for initial protections allow appropriate UNIX behavior: "inherit SID information from process" and "use rights specified by the process, masked by umask." The first pseudo right means the SID information (process, group or organization) is *not* to be overridden by information in the directory. The second pseudo right means the rights specified by the process (and modified by "umask") are *not* to be overridden by information in the directory. These pseudo rights allow UNIX behavior while allowing Aegis style inheritance to continue to override the process information. It is necessary to distinguish between BSD4.3 and System V.3 semantics because BSD4.3 specifies that the owning group is to be inherited from the containing directory, whereas in System V.3 the owning group is inherited from the creating process.

Integrating ACLs with UNIX Protections

This section describes how the extended protection system extends the standard UNIX protection model, paying particular attention to the behavior of the standard protection-related UNIX system calls when extended protections are used. **Goals** describes our goals. We also present the motivation for the initial protection mechanism, how querying and modifying protections works, architectural principles, and the integrated protection model.

Goals

In designing the extended protection system, our primary goal was to make it possible to use unmodified UNIX programs in a system where administrators or users have chosen to use extended protections, and get "reasonable" results. Two sets of UNIX system calls deal with protection:

- File creation calls (**open**(), **creat**(), **mknod**(), **mkdir**()), which specify initial protection modes and ownership information for newly created files.
- Calls for querying about or changing file attributes (**stat**(), **chmod**(), **chown**()), which allow the client to read or modify the protection-related attributes of files.

For file creation, we wanted to enable system administrators or ordinary users to conveniently use the extended protection system

without modifying any UNIX programs; this implies the ability to specify initial protections to be applied to a file external to the program creating the file. We wanted to design a "mechanism" for specifying initial file protections that could support a variety of "policies" for use of the protection system as a whole; in particular, because Apollo's UNIX product is a dual port of Berkeley and AT&T variations, it was important to support both the BSD4.3 and System V.3 policies for initial file protection.

For the system calls that query and modify file protections, it was again important to maintain reasonable behavior for unmodified UNIX programs in the presence of extended protections. Because standard UNIX system calls only deal with the rights of the owner, group, and "others," a secondary goal, time permitting, was to add a new programming interface to the extended protection system. We felt that the protection policy would be set by users or system administrators, or would be specified during installation of software subsystems, and hence it would be uncommon for programs to explicitly create or apply extended protections.

Initial File Protection Mechanism

The UNIX protection model, especially in the Berkeley variations, has the beginnings of separation of the protection policy from the protection mechanism. In the Berkeley UNIX system, three independent specifiers control initial file protections:

- The protection mode supplied in the **open**() call, from the creating program.
- The setting of the per-process umask, from the user running the program.
- Group ownership, from the directory in which the file is being created.

Each of these specifiers is a potential way of specifying extended initial protections; we concluded that the best approach was to extend the notion of inheritance from the directory in which the file is being created. Because we could not require for existing programs to be modified, we could not require changing the **open**() call. We considered extending the "umask" notion to include the ability to specify extended protection information; this had the wrong level of granularity for the specification of a protection policy. There is no

reason to assume that all files created by a given process should be protected the same way (consider a compiler creating temporary files for intermediate results and a binary file for the final output). Specifying the protection policy on a per-directory basis was intuitively appealing; it matched common uses of the file system (for example, all source files in a source tree would have common protections); and it had worked well in practice in the Aegis operating system.

Beginning with this idea, we evolved the model for specifying initial protections. We now describe the algorithm for file creation and setting the initial protection in more detail:

- Start with the owner, group, and organization of the creating process, and the protection mode value passed in to the **open**() call.
- Modify the protection mode value by masking with the process's umask.
- For each required entry (identifiers and protection mode values), if the initial protection specification in the directory specifies an explicit value, override the value supplied by the process for that entry. This operation is called "merging the initial protections."
- If the initial protection specification specifies extended information, apply the extended information to the file.
- When creating a sub-directory, the initial protection specifications for the sub-directory are inherited from its parent directory.

The result is a flexible mechanism for specifying the initial protections to be applied to newly created files. The examples section presents sample protection policies that use this mechanism.

Inquiring and Modifying Protections

The most difficult task in extending the UNIX protection model with ACLs arose in ensuring that the existing UNIX system calls for inquiring and modifying file protections (**stat**(), **chmod**(), **chown**()) continued to have reasonable and consistent behavior, even in the presence of extended protection information. We real-

ized early in the design that there would be circumstances in which the UNIX calls could not provide an accurate representation of the extended protection information; when using the UNIX calls on files with extended protection, there would always be a possibility of unexpected behavior.

Consider the case of the **stat**() call applied to a file with extended protection. **Stat**() returns (among other things) file protection information in three categories: owner rights, group rights, and others rights. Clients of the **stat**() call use this information both to provide information to users (for example, in listing directories) and to make decisions about program behavior (the shell will not execute a script to which the user does not have execute rights). When applied to a file protected with an extended protection, the **stat**() call could do one of two things:

- It could "lie" about the accessibility of the file. It could return the rights for owner, group, and world and ignore the presence of the extended entries. Or, it could combine the rights granted by extended entries into the "others" rights. In either case, clients of **stat**() will be confused: either the client will be unable to access a file to which **stat**() claims it has access, or the client will have access rights not represented in the output of **stat**().

- It could construct a protection mode accurately representing the rights of its client. This is attractive, as the client will never see inconsistencies between the values returned by stat() and the behavior of other UNIX system calls, but has two serious disadvantages. First, **stat**() would return different values for different clients; two users listing the same directory might see different protection modes on files. (Previous versions of our system used a similar scheme, which led to confusion among users.) Second, we felt that the performance degradation caused by computing the protection mode returned by **stat**() on a client-by-client basis would be prohibitive. **Stat**() is a frequently used call in UNIX systems; it is the only way to obtain file attributes, and always returns full information. Expensive protection mode computation affects all callers.

Similarly, what happens when **chmod**() modifies protections on a file with extended protection? The handling of the owner and group rights is straightforward, but it's not clear what the "others" rights supplied to **chmod**() mean. The caller could intend for any extended entries to be disabled, and the "others" rights to be

granted to everyone except the file's owner and group. Alternatively, the caller could be a naive user knowing nothing about extended protection and simply adding rights for the owner of the file, and using **chmod**() because it was the only UNIX system call he knew.

Architectural Principles

Where we could not completely meet our primary goal of "expected" behavior from UNIX system calls in the presence of extended protections, we felt there should be architectural principles from which the behavior can be derived. We identified the following principles as being important:

- In the absence of extended protection information, the protection system must exactly implement the UNIX semantics. It must be easy to configure a system to use only UNIX protections; having done so, the extended protection system should be invisible to the UNIX user who does not use extended UNIX commands. This principle was met by the system for specifying initial protections.

- The behavior of the UNIX system calls should not be dependent on the identity of the user making the call. For example, **stat**() should return the same value for the protection mode irrespective of the identity of the caller.

- UNIX system calls should always err towards increased security. For example, if any user has read access to a file, **stat**() should represent that fact, even if that means over-representing the rights of some users.

- It must be possible, using UNIX system calls, to disable the effects of the extended protection system. Using **chmod**() to deny group and world access to a file should also disable rights granted through extended entries. This follows from the previous principle: err towards increased security.

These principles match existing UNIX standards; the IEEE POSIX specification [8], for example, demands that **chmod**() disable extended protection information.

The Integrated Protection Model

The model for integrating the extended protection system into the UNIX protection model is derived from the architectural principles listed above. The protection information for the owner and group of a file, as returned by **stat**() and controlled by **chmod**(), is precisely the protection information for the file's owner and group as maintained by the extended protection system, while the protection information for "others" is a summary of all other protection information maintained by the extended protection system. This summary is normally the logical OR of the organization rights, world rights, and all rights granted by extended entries; but may be modified by **chmod**(), as described below. To improve performance, the inode includes this summary information; it serves a dual purpose:

The **stat**() call returns the summary information as the "others" rights. For example, if any user other than owner and group has rights to read a file, the "others" rights returned by **stat**() will include read rights.

When checking access rights, the system masks the extended protection information in extended entries with the summary information from the inode. **Chmod**() sets the summary information to the "others" rights supplied in the **chmod**(), permitting **chmod**() to disable extended entries if desired.

Consider a file with the following protections:

```
Owner:     frank                prw-
Group:     acct_r               -r--
Org:       finance              -r--
World:                          ----
Extended Entries:
  donna.hdr.finance             -rw-
```

The protection mode returned by **stat**() will be **rw-r--rw-**. If the owner of the file changes the file's protection mode to **rw-r--r--**, this changes the summary information to **r--**. By the masking operation described above, this will effectively remove the Write rights granted to **donna.hdr.finance** by the extended entry.

Note how this protection model meets the goals described above:

- **Stat**() is fast because it never has to examine the extended entries. It returns the same protection rights irrespective of the identity of its caller.
- **Stat**() errs in the direction of increased security: if any user has Write rights to a file, this fact will be reflected in the "others" information returned by **stat**().
- **Chmod**() can completely disable the extended entries.

We have also added a set of utilities that allow UNIX users to list, copy, and edit ACLs.

Implementation Description

This section provides the reader with a description of some of the more interesting aspects of the implementation of the extended protection system.

Storing Protection Information

Protection information is abstracted into two pieces: required information and extended information. The object's inode contains the required information. Extended information, if present, is stored in a separate object called an ACL object that is pointed to by the object's inode. ACL objects are immutable and read–only; once they are created they may not be changed. Changing the protection associated with an object is accomplished by creating a new ACL object. ACL objects are shareable; one ACL object may specify the extended protection information for many data objects. ACL objects have reference counts; when the last reference to an ACL object is removed, the ACL object is deleted.

Protection information for initial file and initial directory ACLs is implemented in a similar manner. A header in the directory contains the initial file and initial directory required information. Separate ACL objects contain extended information, if present. When applying an extended ACL to a newly created file, the reference count on the ACL object is merely incremented, reducing file creation costs.

A file system scanner allows ACL objects with identical protection information to be compacted, reducing disk space required for ACL objects.

Caching of Protection Information

Several levels of caching improve the performance of the protection system. The first cache is the inode cache. Whenever an object's attributes are examined, they are placed into an operating system cache. The primary purpose of this cache is to maintain location and object information. However, since the inode includes the required portion of the protection information, this information is cached as well.

When a file is opened, the protection rights associated with the opening process are maintained in an open file table. This information is available for later rights checking. With this cache rights checking can immediately provide rights information without sending network messages to the owning node to re-determine the rights available.

The contents of ACL objects are also cached. Each extended ACL object is identified by a unique id (UID) [2]. This cache provides UID to extended ACL information mapping. Whenever it is necessary to consult the information stored in an ACL object, this cache is consulted. If the cache does not contain the ACL object, a cache replacement algorithm selects an entry to replace; the ACL object is then read into the cache entry. Because users tend to protect objects in a few unique ways, there is a high probability of locating the ACL information in the cache.

Finally, there is a higher level cache used when copying objects from volume to volume. Each ACL object must reside on the same volume as the object it protects. As a result, if an object with an associated ACL object is copied from one volume to another in a mode preserving protection information, it is necessary to create a new ACL object on the destination volume. To minimize the number of new ACL objects created a cache keeps track of source ACL object to destination ACL object mappings. This means that if a complete tree is copied from one volume to another, objects sharing ACL objects on the source will share ACL objects on the destination.

Using a Copy Semantic

For copying protection information from one location to another, we decided to use a copy semantic. What we mean by a copy semantic is the following: when a user program wishes to copy protection information from one object to another, the program identifies the source, destination, and source and destination types and calls a protection copy procedure. Directories require the type information, as they have three sets of protection information (i.e., the directory protection information, the initial file information, and the initial directory information). The copy semantic seems to be a natural way to handle protection information. In addition, it has the advantages of making protection copying easy and of insulating programs from the low-level protection information. This contrasts with another frequently used method, where a program gets the old information and then applies it to a new object; this method requires the program to allocate temporary data structures of correct types to hold the protection information. Different calls may also be necessary for different object types. We expect that a system call interface to ACLs would include a call based on the copy semantic.

Using the Protection System

This section provides several examples of how we use the extended protection mechanism to implement various protection policies.

Using BSD4.3 Style Protection Policy

A person using strictly BSD4.3 style protection information would not use extended entries at all. Required information would specify the protection information. An example of the protection information for a file might be:

```
Owner:    frank      prwx
Group:    osdev      -rwx
Org:      r_d        [ignored]
World:               -r-x
```

Notice that the owner has Protect rights, allowing him to change the protection information for the file.

Directories would have protection information set so that the BSD4.3 protection policy would be applied to files and directories created within the directory. An example of initial file information:

```
Owner:    [from process]     [specified by process]
Group:    osdev              [specified by process]
Org:      [from process]     [ignored]
World:                       [specified by process]
```

By the notation **[from process]** we mean that the corresponding field of the creating process's SID is substituted. By the notation **[specified by process]** we mean that the rights are generated by taking the rights supplied by the process to **open**() or **creat**(), and modified by the process's umask. If a process has an effective SID of **mark.testing.r_d**, specifies rights of 775 to **open**() and has a umask of 011, the resulting protection information would be:

```
Owner:    mark               prwx
Group:    osdev              -rw-
Org:      r_d                [ignored]
World:                       -r--
```

Using System V.3 Style Protection Policy

System V.3 style protection information is similar to BSD4.3 protection information. The main exception is in the initial protection in a directory. An example of initial file protection information:

```
Owner:    [from process]     [specified by process]
Group:    [from process]     [specified by process]
Org:      [from process]     [ignored]
World:                       [specified by process]
```

The notation is the same as in the previous example.

Using UNIX Protection Plus Simple Extended Entry

A UNIX user can take advantage of the extended protection system by adding extended entries. For example, if the owner of a file is **frank.acct_r.finance** and he wishes to allow **donna.hdr.finance**

the ability to write the file while denying others write access, he would protect the file as follows:

```
Owner:    frank                prw-
Group:    acct_r               -r--
Org:      finance              -r--
World:                         ----
Extended Entries:
 donna.hdr.finance             -rw-
```

Note that the world entry has been given no rights.

Using More Sophisticated Protection

This example shows how one might set up initial protections on a directory to enforce a policy that preserves information about an object's creator while maintaining explicit protections:

```
Owner:    [from process]       [ignored]
Group:    [from process]       [ignored]
Org:      [from process]       [ignored]
World:                         -r--
Extended Entries:
 frank.acct_r.finance                    prw-
 john.acct_p                   -rw-
 acct_r                        -rw-
 finance                       -r--
```

When files are created in the directory, the required entries will record the SID information of the creating process, while the extended entries control the protection rights available. If **mary.acct_r.finance** creates a file in this directory, its protection will be:

```
Owner:    mary          [ignored]
Group:    acct_r        [ignored]
Org:      finance       [ignored]
World:                  -r--
Extended Entries:
 frank.acct_r.finance             prw-
 john.acct_p          -rw-
 acct_r               -rw-
 finance              -r--
```

Conclusions and Lessons Learned

Some of the decisions made during the design and implementation had wide-reaching effects in the operating system. For example, we decided to make UNIX ids (the small integers associated with UNIX user ids) part of the object attributes to speed up **stat()** performance. This decision implies that the system must be careful when manipulating SIDs so that the UNIX ids are always available for file creation. We wrote a file system scanner that takes the unique ids associated with SID information (i.e., the person, group or organization) and recalculates any UNIX ids that are incorrect. This would normally only be necessary for conflicting UNIX ids when merging networks. During implementation it was a useful tool for tracking and repairing incorrect id information.

Reconciling the different origins of the new system was often difficult. This was partially described in the discussion of integrating ACLs with UNIX protections. Often the Aegis way of performing an operation conflicted with the way that the UNIX system performs an operation. Sometimes it was possible to compromise and allow both ways to work (i.e., initial protections). In other cases it was necessary to just do something the UNIX way (see the discussion of **chmod** in "Integrating ACLs with UNIX Protections" in this article). Often it was not a question of which alternative was right or wrong, but merely that the two systems had chosen to do things in a different manner.

Compatibility with previous operating systems is a feature that Apollo feels is important in a network operating system. Workstations are encouraged to reference data residing on other workstations. Compatibility adds a series of problems that would not occur in a system where ties between machines are weaker. For a major change to the operating system such as the extended protection system changes described in this paper, compatibility is a major concern. We would estimate that 50% of the new code associated with the extended protection system was compatibility code.

Our protection system includes the concept of super-user. In a network of workstations, this concept causes problems in the administration of workstations. It would be desirable to eliminate or modify the way super-user is implemented (a possible alternative is described in [1]); we have left this as an issue in future work.

We believe that we have shown how Access Control Lists can be viewed as an extension by UNIX programs, how they allow better granularity over the control of access to objects in a file system, and have provided information describing our implementation.

References

[1] M. S. Hecht, M. E. Carson, C. S. Chandersekaran, et al. 1987. UNIX without the super-user. *Proceedings of the USENIX Association Summer Conference.*, pp. 243–256. June, 1987.

[2] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, Paul H. Levine. 1982. UIDs as Internal Names in a Distributed File System. *Proceedings of the First Symposium on Principles of Distributed Computing*, pp. 34–41. August, 1982.

[3] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf. 1983. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications*, pp. 842–856. November, 1983.

[4] Henry M. Levy. Capability Based Computer Systems. Digital Press, Bedford, Mass. 1984.

[5] David L. Nelson, Paul J. Leach. 1984. The Architecture and Applications of the Apollo Domain. *IEEE Computer Graphics and Applications*, pp. 58–66. April, 1984.

[6] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, Mass. 1972.

[7] D. M. Ritchie, K. Thompson. The UNIX Time-Sharing System. *Bell System Technical Journal,* pp. 1905–1969. July–August, 1978.

[8] Technical Committee on Operating Systems of the IEEE Computer Society, P1003.1. *Portable Operating System Interface for Computer Environments*, Draft 12. October, 1987.

A User Account Registration System for a Large (Heterogeneous) UNIX Network

by

Joseph N. Pato, Elizabeth Martin, Betsy Davis

Abstract

Three problem areas arise when considering a user registration system for a large heterogeneous distributed computing environment. Large environments demand controls on the complexity of administration. Heterogeneity requires an examination of the notion of identity in the network as well as the interoperability of software on different hosts. Distribution raises the problems of availability, reliability, and security.

Generally available UNIX environments (BSD4.3 and AT&T SYS5.3) provide few tools for solving these problems. Account administration is typically handled through manual editing of a single **/etc/passwd** file. Consistency is maintained on multiple machines by periodically copying the **/etc/passwd** file to each machine in the network. For large networks with thousands of users and machines, these mechanisms are clumsy and error prone, and they vest too much power in a single system administrator.

RGY is a replicated user registration system built on Apollo's portable Network Computing System (NCS). The system consists of a set of daemons which maintain a replicated user registration database. Remote access to the user registration database is provided at each client site through remote procedure calls in a portable subroutine library that replaces the **getpwent**(3) and **getgrent**(3) C library calls. Weakly consistent replication provides a high degree of availability and reliability. Propagation of individual updates is performed yielding an inexpensive mechanism for maintaining consistency. Updates are securely performed using authenticated interfaces, allowing any client site to update the database.

Introduction

In a conventional single host UNIX environment, system account administration is managed through manipulation of the **/etc/passwd** and **/etc/group** files. Generally a system administrator is responsible for properly editing these files as well as performing the system house-cleaning associated with the arrival and departure of users. When UNIX was primarily an operating system associated with departmental minicomputer environments that consisted of few (under 100) users, the burden on the administrator was tolerable.

In the mid 1980's, networks of inexpensive UNIX workstations became common, providing a workstation owner with a high degree of autonomy, as well as guaranteed response time in the absence of time-sharing. As long as each workstation or cluster of workstations remained autonomous, the administrative burden associated with each machine remained low. Account management could be delegated to members of the user community for each workstation or cluster.

With this autonomy, however, early workstation users also encountered isolation. Data and resource sharing became cumbersome, relying on bulk data transfer protocols like FTP and virtual terminal protocols like Telnet. To recover some of the cooperation found in time-sharing systems, computer vendors introduced distributed file systems like Apollo's Domain [6], Sun's NFS [10] and AT&T's RFS [9], and later developed network computing environments, like Apollo's Network Computing System (NCS) [4] and Sun's ONC [11]. Network computing environments provide heterogeneous compute and resource sharing while distributed file systems provide data sharing. Distributed file systems can be considered a subset of network computing environments. Therefore, for the purposes of this paper, we will use the term network computing environment to refer to either of these forms of network resource sharing.

A network computing environment transforms a network of workstations from independent administrative jurisdictions to a federa-

tion of loosely coupled systems. For access control mechanisms to be meaningful, every system must share a single representation for users' **credentials** (user names and user IDs). With independent workstations, the assignment of user names and user IDs needs to be unique only on each machine; in a network computing environment, this assignment must be unique across the network.

Since a user's credentials must be unique across the network, system administrators can no longer delegate account management responsibilities to individual workstation user communities without compromising the security of other user communities in the federated system. In contrast to isolated workstations, which can diffuse the administrative burden of account management, a network computing environment forces account management responsibilities to be assumed by a network administration authority. This network administrator accumulates the requirements of each workstation user community and then must redistribute the account information to each workstation in the federation.

RGY, a replicated user registration system built on Apollo's NCS, has been developed to allow the administration of large network environments. Our goal is to provide a network user registration system that will work well in a network of tens of thousands of hosts and users. To accomplish this we have developed the replicated user registration database, RGY, to serve as the secure repository for network system account management information. Access to the RGY database is provided through NCS remote procedure call interfaces exported by a collection of daemon processes known as **RGYD**s.

Hosts that wish to participate in the federated system access the RGY database through existing **getpwent**(3) and **getgrent**(3) C library calls as well as through additional query and update primitives. Each host is the final authority in granting access to its resources. The RGY database allows the federated hosts to provide a consistent view of the user community, but each host is free to filter the information from the RGY database to restrict access, or to correct for differences in the local file system.

This paper examines the following topics:

- Existing mechanisms for coping with network account administration
- The data model and tools we developed to allow for division of labor in maintaining the user registration database
- Providing highly available, reliable and efficient access to the RGY database
- The mechanism to secure update access to the RGY system

Existing Systems

Password file maintenance is a real and present problem for administrators of large UNIX installations. The 1987 USENIX Large Installation System Administrators Workshop drew numerous position papers on UNIX account management and the distribution of information across networks. At this workshop, attendees discussed the evolutionary processes that result in large installations and the need to unify account information in the resulting network [3]. Other attendees described the use of structured editors for adding entries to the password file and the subsequent semi–automatic copying of the file to all hosts in the network [7]. These current approaches, centered around the existing UNIX data and administrative models, are cumbersome in today's small networks (fewer than 100 hosts) and hold little promise for the large (thousands of hosts) networks of the future.

Remote access to the login account database allows replication strategies to limit their focus to a strategic subset of the network. Sun's Yellow Pages (YP), part of ONC, is a simple network lookup service that has been used to provide remote access to password file information. While YP did not modify the **/etc/passwd** data model, it did introduce the use of remote procedure calls to remotely access the UNIX login account database. Outside the UNIX environment, the Xerox Grapevine system [2] has addressed many of these issues. Grapevine was intended to be primarily used as a delivery mechanism for a large, dispersed computer mail system. It

did not directly address the issues of maintaining a replicated user login account database. It maintained a replicated database of mail users which, in some instances, could also be used for user logins. Its concern for the issues of authentication, access control, decentralized administration, replication and scalability has served as inspiration for much of the work described in this paper.

Administrative Model

Large computing environments tend to contain a large number of both users and machines. Frequently these consumers and resources span administrative or organizational domains. To accommodate this type of environment we have enhanced the UNIX identity model to include the notion of organization. In addition, we have added access control objects to each entry in the user registration database. These changes allow mutually suspicious system administrators to cooperatively manage a logically partitioned user registration database. Administrative complexity is further reduced through the use of a structured editor for database manipulations.

The RGY Data Model

The RGY user registration system maintains a database consisting of **naming** information for people, groups and organizations, **login account** information for people, and general system **properties** and **policies**. In the **/etc/passwd** file people and accounts are combined in a single record. We, however, feel that people and accounts are distinct objects in a user registration system: accounts represent active roles that people can play when accessing the system, whereas people maintain passive roles through the ownership of files, receipt of mail, etc. that persist independent of the existence of an account.

Groups and organizations are collections of people. Groups retain their conventional UNIX semantics and exist to allow a collection of people to share privileges to system objects. Organizations provide another dimension for sharing. Apollo has extended the UNIX file protection model from user, group, others (rwxrwxrwx) access

to user, group, organization, others access. Organizations can be used just like groups, where a person can be a member of any number of groups. More typically, however, we use organizations as a means of partitioning the global user community into administrative jurisdictions, where each person belongs to a single organization.

In addition to maintaining information about the users and logical groupings of the networked system, the RGY database contains system policy information. Policy information consists of system configured minimum password length, password content restrictions, password expiration lifetime, absolute password expiration date, and account lifespans. Policy is never enforced by the user registration system. It exists as a guide for clients of the system.

Naming Information

The naming database is divided into three relations, also referred to as naming domains, that establish the existence of individual persons, groups, and organizations within the registry. An entry in one of these naming domains is called a **PGOitem**. A PGOitem establishes the binding between a name and a set of credentials which consist of a unique identifier (UID) and a **unix id**. The unix id, preserved for compatibility with password file entries, is a small integer value used as a user id for people, a group id for groups, and an org id for organizations. Aliases, multiple names mapping to the same credential information, are allowed to exist.

PGOitems contain a *fullname* field, an *owner* field and miscellaneous properties. A PGOitem can contain a list of typed mail data. This data consists of a type code and an uninterpreted *printstring*. The printstrings may be interpreted by a system mailer and usually define the preferred delivery mechanism or mailbox to be used.

Groups and Organization PGOitems have associated membership lists. A membership list enumerates the people that have the rights and privileges of the group or organization. Organization PGOitems may also contain policy information. By establishing policy, an or-

ganization may impose stricter password and account discipline than the other organizations in the registry. The actual policy data for an organization can be retrieved for editing, but most operations that yield policy information return the **effective policy** data. To determine an organization's effective policy, the system compares the organization policy information with the base registry policy and returns the most restrictive value for each field.

Login Accounts

Accounts contain a superset of the information stored in the **/etc/passwd** file. An account entry is divided into two portions. The user portion of the account contains the home directory, login shell, password, and gecos fields. The administrative portion contains information about the creator of the account, account expiration date and other information to indicate the validity of the account. An account defines a subject identifier (SID). A SID is a UID triplet which identifies the person, group, and organization that correspond to the account.

UIDs [5], which are used extensively throughout the Apollo system, are a 64-bit concatenation of the current time and host network address. Unlike the unix ids which are assigned by the system administrator, UIDs are generated for the PGOentry by the RGY system and are guaranteed to be unique.

Accounts are keyed by login name, which is the concatenation of the person name, the group name and the organization name separated by periods (e.g., the user **smith** might have the account **smith.sys.r_d**). Login names can be abbreviated; accounts define the minimum abbreviation necessary for their selection. In the example above, the account **smith.sys.r_d** could be accessed as **smith.sys** if the associated abbreviation was person and group, or as **smith** if the associated abbreviation was simply person. Each person may have multiple accounts, either by using aliases, or by creating accounts with different abbreviations.

Decentralized Administration

Owner fields in PGOitems and registry properties allow mutually suspicious system administrators to securely partition the administration of the RGY database. An owner field defines who can update the corresponding record.

Access Controls

The RGY database maintains certain access controls when updating information in the database. Only the registry owner, stored in the registry properties, can update the registry properties or policy. The registry properties also contain owner records for each of the naming domains. Only the owner of each naming domain can create new PGOitems in that domain.

When a PGOitem is created, it is assigned an owner. All future manipulations of that PGOitem can only be performed by the owner. Group and organization membership lists can only be manipulated by the owner of the group or organization PGOitem.

Accounts can be created only by the owner of the corresponding person PGOitem. To have an account that is affiliated with a specific group and organization, a person must first be a member of the corresponding group and organization. Update of the administrative portion of accounts is reserved to the owner of the corresponding person PGOitem; updates to the user portion of an account can only be performed by the corresponding person, or by the owner of the corresponding person PGOitem. If a person is deleted from a group or organization membership list, then any accounts that may exist for that person in the group or organization are also deleted. Group and organization membership as a pre-condition for account existence is an invariant that is maintained by the RGY database.

The Representation of Owners

An owner field is represented by a SID where any constituent field may be replaced by a wildcard (represented by the character %). Owner permissions are granted to anyone who can login with an account that has a SID that matches the owner SID. If the owner SID contains a wildcard for one of the constituent fields, then any value for that field will match. In a free-wheeling system, all the owner fields could be set to %.%.%, thus allowing anyone to manipulate all the data in the RGY database. Owner records of the form %.**rgy_admin**.% would grant permission to anyone logged in with an account that had **rgy_admin** as the group portion of its SID.

If all owner fields in the RGY database are the same, then update access to the RGY database is comparable to update access to the **/etc/passwd** file. When the RGY database contains a large number of people, however, it is more likely that each user community will have the PGOitems corresponding to its members owned by an administrator from within that user community.

Example

For the purposes of this example, we will assume that the network and machines in the federation are secure. The only security risk we are concerned with is the access to or corruption of data by a person with a valid but unauthorized account.

In a large corporation, a small group of researchers are working on a sensitive project called the **manhattan project**. To protect the confidentiality of their work, they have protected their files so that access is limited to members of the **manhattan** group. The researchers could disconnect their machines from the corporate federation and ensure their security, but to do so would unduly disrupt their work. How do they guarantee that no one outside the group acquire an account with access to their data?

The first step is to assign the ownership of the **manhattan** group PGOitem to a member of the **manhattan** group (e.g., **teller.manhattan.research**). This will guarantee that only the user **teller**, who is a member of the **manhattan** group can add or delete members from the group. For most situations this will be sufficient, but for truly security conscious environments more must be done.

Assume that the user **fermi** is a member of the **manhattan** group. Further assume that the owner of **fermi**'s person PGOitem is **malicious.spy.%**. It would be a simple matter for **malicious** to change the password on **fermi**'s account and thus compromise the **manhattan** group's data. To be fully secure, the administrator of the **manhattan** group (**teller**) should allow people to be members of the group only if he is also the owner of their person PGOitems.

Structured Editing

The **edrgy** tool is used to manage the naming, account, and policy information in the RGY database. It is an interactive editor that provides users and system administrators with a structured interface to the user registration system, at once ensuring consistency, semantic correctness, and timely availability of changes.

Edrgy is aware of the semantic constraints placed upon the contents of the RGY database, and of the policy that is in effect. **Edrgy** uses this knowledge to assist the system administrator in performing semantically correct operations. For example, if the system administrator attempts to add an account for a person that does not belong to the requested group, and the administrator has rights to update the group, then **edrgy** will first add the person to the group and then add the account. Warnings are given before an operation is performed if that operation may have side effects. For example, **edrgy** will warn that the deletion of a group will also delete any accounts that exist with that group's permissions.

System Structure

The RGY user registration system is composed of two distinct portions: the database, which is an NCS replicated object, and the client agent which provides RGY access for the host environment.

RGYD: the RGY Daemon

RGYD is the NCS server (process) that exports remote interfaces to the RGY database. Three classes of interfaces are exported by the RGYD: database queries and updates, replica control, and database update propagation.

Database Operations

Database operations involve the maintenance and use of the RGY database. RGYD exports interfaces to query and update all RGY structures directly. In addition, the RGYDs maintain a set of interfaces presenting a view of the database that is equivalent to the view presented by the **getpwent**(3) and **getgrent**(3) C library functions. By extracting information from the corresponding PGOitem and account records RGYD constructs password file entries. Group and org file records are constructed from the corresponding PGOitem and membership lists.

The RGY database is kept in virtual memory as a forest of balanced binary trees [1] yielding efficient ($O(\log n)$ operations where n is the number of items in each relation) query and update access. Deleted items are marked and left in the trees until garbage collection is performed during a checkpoint. Updates are first applied to the in memory data structures and are then atomically recorded in a stable storage log. Checkpoints of the in–memory data structures are taken every few hours for each relation that has been modified since the last checkpoint. The RGYD automatically recovers the state of the RGY database after a system crash by reloading the last

checkpoint state and then re-executing each operation recorded in the stable storage log.

Not all UNIX programs access the password file structures through the procedural interfaces provided by the C library. To accommodate these programs, the RGYD maintains ASCII file versions of the password, group, and org file. These files are recreated at each checkpoint if the data in these views has been modified.

Replica Management

A collection of RGYD processes spread across a number of hosts cooperate to maintain a weakly consistent replicated database. Updates do not occur at all RGYDs simultaneously; instead, one of the RGYDs is selected to serve as the master site and becomes the only daemon that accepts database updates. The master RGYD then assumes the responsibility of propagating each update to the other cooperating (slave) RGYDs.

RGYD sites may come, go, and move around with ease. When a slave RGYD first starts running, it locates the master RGYD through the NCS Global Location Broker and announces its existence. If this RGYD is a new site, then the master RGYD will initialize the slave and record an operation to inform all other slaves of its existence. If the new RGYD is an existing site that has moved to a new address, the master RGYD will record this change of address and inform the other replicas. In this way each RGYD maintains a current copy of the replica list.

A special tool, **rgy_admin**, is used to remotely inspect and control each RGYD. All operations that affect the state of a RGYD are reserved to the owner of the registry database as recorded in the RGY properties data. With the **rgy_admin** tool, the registry owner can determine if replicas are out of date, cause a replica site to be reinitialized, select a new master site and decommission a RGYD site. When a RGYD site receives the decommission request, it purges its database and terminates execution.

Update Propagation

In addition to managing the database and replica list, the master RGYD also manages a propagation queue. The role of the propagation queue is analogous to the stable storage log. Every update operation performed at the master RGYD is recorded in the propagation queue for later application at the slave RGYDs. In practice, the propagation queue and the stable storage log are the same structure. Slave RGYDs are free to truncate the stable storage log once a checkpoint has completed, but the master RGYD must preserve the portion of the log that remains to be propagated to each slave. Update propagations, like all other remote operations, are accomplished through the use of a remote procedure calls.

A simple protocol between the master and slaves ensures that updates are processed in serial order. The master RGYD applies a monotonically increasing timestamp to each update it records. When propagating an update, the master RGYD transmits the previous update timestamp as well as the current update timestamp. Retransmitted updates are simply ignored by the slaves, but if the slave detects that it is out of date with respect to the previous update it requests to be reinitialized.

The master RGYD periodically retransmits an update to a slave which is unreachable. As the number of attempts to reach the slave increases, the time interval between retransmissions is also increased. Eventually the master will mark a slave as out of touch and and will re-initialize the slave when it finally becomes reachable. Database initialization is accomplished through bulk transfer of the database state to the target slave RGYD. Thus the master may purge updates from its propagation queue even when some slaves are unreachable for long periods of time.

The RGY Client Agent

The RGY Client Agent (RCA) is divided into two components. The most primitive level consists of the automatically generated client side RPC stubs for the registry operations and code for binding to a RGYD. The next, optional layer of the client agent (RGYC) pro-

vides registry services in the event of network failure. This layer can be used for translating credentials in a heterogeneous environment or for filtering data from the network registry. The first time a RGY operation is performed, the RCA contacts the NCS Global Location Broker to randomly select a RGYD site for the operation. Subsequent operations are directed to the same RGYD until that server becomes unavailable. To allow the client agent to remain unaware of the replication strategy chosen by the RGYDs, we have divided RGY operations into separate query and update interfaces. The RCA actually maintains two bindings, one for queries and the other for updates. With the master/slave replication currently implemented by the RGYDs, only one server will register the update interface at a time. An explicit binding operation is provided by the RCA for registry editing applications. This allows the application to force queries and updates to be delivered to the same RGYD.

The Local Registry

The local registry, maintained on each node by the RGYC, provides a cache of user registration data in the event that a registry server is not available. This cache of recently used accounts supports queries for login and C library calls (**getpwent**(3), **getgrent**(3)). In order to prevent the cache contents from becoming stale, each remote operation returns the timestamp of the last operation that may have invalidated the cache. If the cache is out of date, the client agent initiates a cache refresh operation.

Authentication

Ignoring well-known security holes in UNIX systems, it can be said that access to files is vigorously protected by the operating system. When deciding to grant access to a file, the kernel is free to believe the identity information that it has stored for the process. In a network computing environment, however, there is no reason for one host to believe that another host has not been compromised. An application cannot even be sure that network messages truly originated with the host listed in the message.

The traditional mechanism for proving identity is the use of a secret that is known only to the two principals, the person claiming the identity and the guardian of the resource. In a network environment where principals reside on different hosts, encryption must be used when exchanging the secret in order to maintain the secret. Our model for network authentication is inspired by Needham and Schroeder's work [8].

All RGYD update and replica administration operations that apply access controls perform authentication as the first step in those controls. To perform authentication, two encryption keys are associated with each account: a login key which is constructed from the plain text of the account's login password, and a master key which is generated by the RGYD when the account is created. When an update request is received by the RGYD, it constructs a random bit pattern and encrypts it with the requester's master key. The RGYD then makes an RPC callback to the initial requester. This callback is a challenge that requires the requester to decrypt the message, perform a function on the bit pattern, and return the encrypted result.

To successfully meet the authentication challenge posed by the RGYD, the requesting process must possess the valid master key for the claimed identity. The login (**/bin/login**) and set user id (**/bin/su**) programs have been modified to acquire the valid master key. Rather than using the standard **getpwent**(3) calls to retrieve the password file record, these programs now make a direct RCA call that retrieves the required information as well as the master key. To protect the master key, it is never transmitted in the clear. It is first encrypted with the login key for the account. In this way the master key will be useful only if the login program possesses the valid password for the account. If the password is not known, the master key will not be decrypted properly.

The mechanism described above is used to guarantee that the RGY database is never modified by unauthorized users. In a security conscious environment, it is also necessary to verify that the client is connected to a legitimate RGYD. Mechanisms to accomplish this task are inherent in the system, but are beyond the scope of this paper.

Conclusions

The RGY system is currently used by about 100 personal workstations on the Apollo corporate internet. The RGY database contains about 2500 users, 100 groups and 50 organizations. On an average day about 150,000 database operations are performed spread out over the 4 RGYDs maintaining the replicated database. While these numbers are small compared to our design goals, we are encouraged to see that we are not yet close to saturating the capacity of a single RGYD even when only one server is running.

We are currently investigating new replication algorithms that will allow us to perform updates at any RGYD site, rather than at only the master RGYD site. These algorithms maintain the semantic invariants in the database, and will improve update availability in the face of network failures.

References

[1] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading MA.

[2] Andrew D. Birrell, Roy Levin, Roger M. Needham and Michael D. Schroeder. Grapevine: an exercise in distributed computing. *Communications of the ACM*, Vol. 25, No. 4. April, 1982.

[3] Pete Cottrell. Password file management at the University of Maryland. *Proceedings of the Large Installation System Administrators Workshop*, pp. 32–33. 1987.

[4] Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, Geoffrey L. Wyant. The network computing architecture and system: an environment for developing distributed applications. *Proceedings of the USENIX Association Summer Conference*. 1987.

[5] Paul J. Leach, Bernard L.Stumpf, James A. Hamilton and Paul H. Levine. UIDs as internal names in a distributed file system. *Proceedings of the Symposium on Principles of Distributed Computing*, pp.34–41. 1982.

[6] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson and Bernard L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAL-I(5): pp. 842–857. 1983.

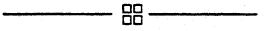
[7] Evelyn C. Leeper. Login management for large installations. *Proceedings of the Large Installation System Administrators Workshop*, 35. April, 1987.

[8] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, Vol. 21, No. 12, pp. 993–999. December, 1978.

[9] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, Kang Yueh. Remote file sharing architectural overview. *Proceedings of the USENIX Association Summer Conference*, pp. 248–259. 1986.

[10] Sun Microsystems Inc. Networking on the Sun workstation. Part no. 800–1324–03. 1986.

[11] Sun Microsystems Inc. Open network computing technical overview. DE240–0. 1987.



The Network Computing Architecture and System: An Environment for Developing Distributed Applications

by

Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin,

Joseph N. Pato, Geoffrey L. Wyant

The Network Computing Architecture (NCA) is an object-oriented framework for developing distributed applications. The Network Computing System (NCS) is a portable implementation of that architecture that runs on UNIX and other systems, including Domain/OS. By adopting an object-oriented approach, we encourage application designers to think in terms of what they want their applications to operate on, not what server they want the applications to make calls to or how those calls are implemented. This design increases robustness and flexibility in a changing environment.

Introduction

NCS currently runs under Apollo's Domain/IX [7], Domain/OS, 4.2BSD and 4.3BSD, and Sun's version of UNIX. Implementations are currently in progress for the IBM PC and VAX/VMS. Apollo Computer has placed NCA in the public domain.

In addition to its object orientation, some interesting features of the system are as follows. It supplies a transport-independent remote procedure call (RPC) facility using BSD sockets as the interface to any datagram facility. It provides at-most-once semantics over the datagram layer, with optimizations if an operation is declared to be idempotent. It is built on top of a concurrent programming support package that provides multiple threads of execution in a single address space, although versions can be made for machines that just have asynchronous timer interrupts.

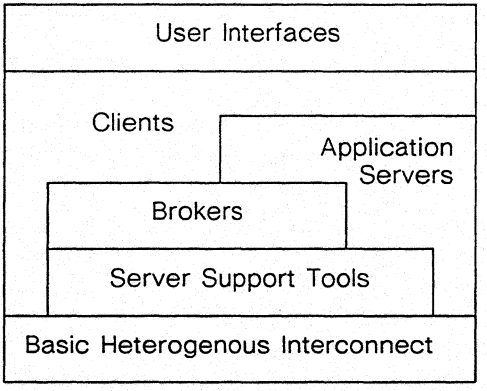
The data representation supports multiple scalar data formats, so that similar machines do not have to convert data to a canonical form, but can instead use their common data formats. The RPC interface definition compiler is extensible. Procedures to do the client/server binding can be attached to data types defined in the interface. Also, complex data types can be marshalled by user-supplied procedures which convert such types to data types the compiler understands. There is a replicated global location database: Using it, the locations of an object can be determined given its object ID, its type, or one of its supported interfaces.

There are several motivations for NCA. Large, heterogeneous networks are becoming more common. Users of systems in such networks are often frustrated by the fact that they can't get those systems to work cooperatively. Over the last few years, advances have been made in allowing data sharing to occur between the systems, but not compute sharing. Tools to allow the effective use of the aggregate compute power have not been available. The inability to share computing resources has become even more aggravating as more specialized processors (e.g., ones designed to run numerical applications fast) have become more widespread. Current "technology" obliges users of those processors to resort to FTP and Telnet. Even in an environment of systems of relatively similar power, a network computing architecture is called for: There are applications that can take advantage of many systems in parallel. (Parallel "make" is the most obvious example.) Also, replicating resources over a number of machines increases the reliability seen by users of the network.

It is important to understand that there is almost no "network application" that can't be implemented without NCA/NCS. However, the implementation is bound to be more difficult, less general, and harder to install on a variety of systems. Further, experience has shown that some obviously useful network applications simply don't get written because of these problems. The existence of NCA/NCS helps to solve these problems and as a result, expand the set of network applications.

Architecture

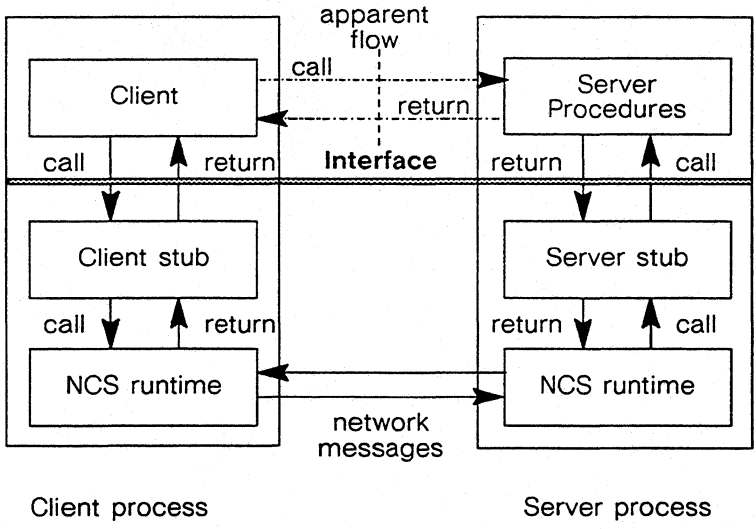
The figure illustrates NCA's overall structure.



NCA's Overall Structure

Heterogeneous Interconnect

The lowest level provides the basic interconnection to heterogenous computing systems. At this layer NCA currently defines a remote procedure call protocol (NCA/RPC), a Network Interface Definition Language (NIDL), and a Network Data Representation (NDR). RPC is a mechanism that allows programs to make calls to subroutines where the caller and the subroutine run in different processes, most commonly on different machines. The RPC approach and an implementation similar to ours is described in detail by Birrell and Nelson [2]. NIDL is a high-level language used to specify the interfaces to procedures that are to be invoked through the RPC mechanism. NCS includes a portable NIDL compiler that takes NIDL interfaces as input and produces stub procedures that, among other things, handle data representation issues and connect program calls to the NCS RPC runtime environment that implements the NCA/RPC protocol. The relationships among the client (i.e. the caller of a remoted procedure), server, stubs, and NCS runtime is shown in the following figure.



Relationships Among Client, Server, Stubs and NCS Runtime

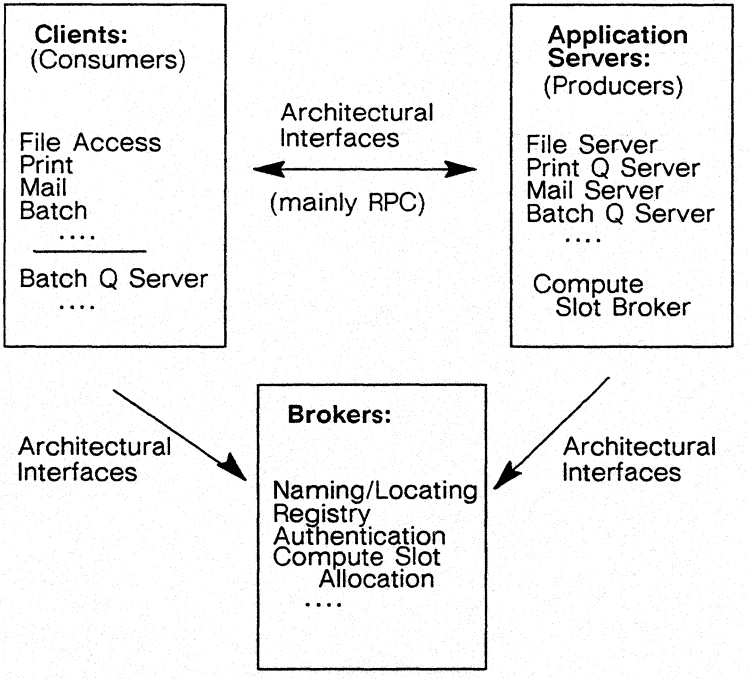
Server Support Tools

Augmenting the heterogenous interconnect layer are the server support tools. These tools simplify the writing of complex applications in a distributed environment. Currently these consist of the Data Replication Manager (DRM) and Concurrent Programming Support (CPS). DRM provides a weakly consistent, replicated database facility. It is useful for providing replicated objects when high availability is important and weak consistency can be tolerated. CPS provides integrated lightweight tasking facilities. CPS allows multithreaded servers to be written easily.

Brokers, Clients, Servers and User Interfaces

Built on top of the server-support tools are a set of **brokers**. A broker is a third party agent that facilitates transactions between principals. In a network computing environment brokers are primarily useful in determining object locations, but can also be used for establishing secure communications (i.e., authentication), associatively selecting objects, issuing software licenses, and a variety of

other administrative chores not directly related to the operation of the principals. The role of brokers is shown in the next figure.



The Role of Brokers in NCA

Client programs and application servers make use of the three base layers. Application servers are the "producers" of services and clients the "consumers." Servers invoke brokers to make their existence known. Clients can invoke brokers to locate application servers and then use the underlying RPC mechanism to make use of the services provided. The application server may be in turn a client of other distributed services.

From user's perspective, user interfaces tie all the pieces together. However, user interfaces are not part of NCA and will not be discussed in this paper.

Unique Identifiers

An important aspect of NCA is its use of **universal unique identifiers** (UUIDs) as the most primitive means of identifying NCA entities (e.g., objects, interfaces, operations). UUIDs are an extension of the unique identifiers (UIDs) already used throughout Apollo's system [6]. Both UIDs and UUIDs are fixed-length identifiers that are guaranteed to refer to just one thing for all time. The principal advantages of using any kind of unique identifiers over using string names at the lowest level of the system include: small size, ease of embedding in data structures, location transparency, and the ability to layer various naming strategies on top of the primitive naming mechanism. Also, identifiers can be generated anywhere, without first having to contact some other agent (e.g., a special server on the network, or a human representative of a company that hands out identifiers).

UIDs are 64 bits long and are guaranteed to be unique across all Apollo systems by embedding in them the node number of the system that generated the UID and the time on that system that the UID was generated. To make it possible to generate unique identifiers on non-Apollo system we defined UUIDs to be 128 bits and made the encoding of the identity of the system that generates the UUID more flexible.

The remainder of this paper discusses several aspects of NCA and NCS: NCA's object-oriented approach; NIDL; NDR; the NIDL compiler; the Location Broker used in connecting clients with servers; and the networking model and protocol used by NCS. We conclude with a description of future directions we expect NCA and NCS to follow.

The Object-Oriented Approach

NCA is object-oriented. By this we mean that it follows a paradigm established by systems such as Smalltalk [4], Eden [1, 5], and Hydra [12, 3]. The basic entity in an object-oriented system is the **object**. An object is a container of state (i.e. data) that can be accessed and modified only through a well-defined set of **operations** (what Smalltalk calls **messages**).

The implementation of the operations is completely hidden from the client (i.e. caller) of the operations. Every object has some (what Smalltalk calls a **class**). The implementation of a set of operations is called a *manager* (what Smalltalk calls a set of **methods**). Only the manager of a type knows the internal structure of objects of the type it manages. Sets of related operations are grouped into *interfaces*. Several types may support the same interface; a single type may support multiple interfaces.

For example, consider an interface called **directory** containing the operations **add_entry**, **drop_entry**, and **list_entries**. This interface might be supported by two types: **directory_of_files** and **print_queue**. There are potentially many objects of these two types. That there are many objects of the type **directory_of_files** should be obvious. By saying that there are many **print_queue** objects we mean that a system (or a network of connected systems) might have many print queues — say, one for each department in a large organization.

Motivation

The reason for using the object-oriented approach in the context of a network architecture is that this approach lets you concentrate on what you want done, instead of where it's going to be done and how it's going to be done: objects are the units of **distribution, abstraction, extension, reconfiguration,** and **reliability**.

Distribution. Distribution addresses the question of where an operation is performed. The answer to this question is that the operation is performed where the object resides. For example, if the print queue lives on system A, then an attempt to add an entry to the queue from system B must be implemented by making a remote procedure call from system B to system A. (This implementation fact is hidden from the program attempting to add the entry.)

Abstraction. Abstraction addresses the question of how an operation is performed. In NCA, the object's type manager knows how the operation is performed. For example, a single program **list_directory** could be used to list both the contents of a file system directory and the contents of a print queue. The program simply calls the **list_entries** operation. The type managers for the two types of objects might represent their information in completely different ways (because, say, of the different performance characteristics required). However, the **list_directory** program uses only the abstract operation and is insulated from the details of a particular type's implementation.

Extension. The object-oriented approach allows extension; i.e. it specifies how the system is enhanced. In NCA, there are two kinds of extensions allowed. The first is extension by creation of new types. For example, users can create new types of objects that support the **directory** interface; programs like **list_directory** that are clients of this interface simply work on objects of the new type, without modification. The second kind of extension is extension by creation of new interfaces. A new interface is the expression of new functionality.

Reconfiguration. Because of partial failures, or for load balancing, networked systems sometimes need to be reconfigured. In object-oriented terms, this reconfiguration takes place by moving objects to new locations. For example, if the system that was the home for some print queue failed because of a hardware problem, the system would be reconfigured by moving the print queue object to a new system (and informing the network of the object's new location).

Reliability. The availability of many systems in a network should result in increased reliability. NCA's approach is to foster increased reliability by allowing objects to be replicated. Replication increases the probability that least one copy of the object will be available to users of the object. To make replication feasible, NCS provides tools to keep multiple replicas of an object in sync.

While NCA is object-oriented and we believe that applications that use the object-oriented capabilities of NCA will be more robust and general than those that don't, it is easy to use NCS as a conventional RPC system, ignoring its object-oriented features.

Network Interface Definition Language

The Network Interface Definition Language (NIDL) is the language used in the Network Computing Architecture to describe the remote interfaces called by clients and provided by servers. Interfaces described in NIDL are checked and translated by the NIDL compiler.

NIDL is strictly a **declarative** language — it has no executable constructs. NIDL contains only constructs for defining the constants, types, and operations of an interface. NIDL is more than an interface definition language however. It is also a network interface definition language and, therefore, it enforces the restrictions inherent in a distributed computing model (e.g. lack of shared memory).

NIDL Language Constructs

A NIDL interface contains an header, constant and type definitions, and operation descriptions. The header provides the interface identification: its UUID, name, and version number. The UUID is the "name" by which an interface is known within NCA. It is similiar to the program number in other RPC systems, except that it is not centrally assigned. The interface name is a string name for the interface which is used by the NIDL compiler in naming certain publicly known variables. The version number is used to support compatible enhancements of interfaces.

A standard set of programming language types is provided. Integers (signed and unsigned) come in one, two, four, and eight byte sizes. Single (four–byte) and double (eight–byte) precision floating–point numbers are available. Other scalars include signed and unsigned characters, as well as booleans and enumerations.

In addition to scalar types, NIDL provides the usual type constructors: structures, unions, pointers, and arrays. Unions must be discriminated. (Non-discriminated unions are not permitted. The actual data values must be known at runtime so that it can be correctly transmitted to the remote server.) Pointers, in general, are restricted to being "top-level." That is, pointers to other pointers, or records containing pointers are not permitted. Later, we'll see how this restriction can be relaxed. Arrays can be fixed in size or have their size determined at runtime.

Operation declarations are the heart of a remote interface definition. These define the procedures and functions that servers implement and to which clients make calls. All operations are strongly typed. This enables the NIDL compiler to generate the code to correctly copy parameters to and from the packet and to do any needed data conversions. Operation declarations can be optionally marked to have certain semantic properties, for example whether they are **idempotent**. (An idempotent procedure is one that can be executed many times with no ill-effect.)

All operations are required to have a **handle** as their first parameter. This parameter is similar to the implicit "self" argument of Smalltalk-80 or the "this" argument of C++ [9]. The handle argument is used to determine what object and server is to receive the remote call. NIDL defines a primitive handle type named **handle_t**. An argument of this type can be used as an operation's handle parameter. Clients can obtain a **handle_t** by calling the NCS runtime, providing an object UUID and network location as input arguments. Use of more abstract kinds of handles is described below.

Handle arguments can be implicit. An interface definition can declare that a single global variable should be treated as the handle argument for all operations in the interface. While this style conflicts with some of the goals of the object-oriented approach (e.g., it makes it harder to make calls on different objects using the same interface), it can be useful in cases where an existing local interface is being converted to work remotely.

NIDL Example

The following figure is a short example of an interface described in NIDL. The example is of an interface to a bank object that supports a single operation: deposit money into an account.

(1) Defines the UUID by which this interface is known. This the first version of this interface. If in the future, new operations are added, the version number should be incremented. (2) Declares the interfaces upon which this interface is dependent. The **import** statement is similiar to **#include**, except that the named interface is not textually included. The contents are made available for the importer to refer to types and constants defined in that interface. This allows factoring out a common set of types into a base interface. (3) Defines a set of types (account and account name types) that are

used by the bank operations. Finally (4) defines the operation itself.

A variant of NIDL that looks Pascal-like (as opposed to the C-like version of which the figure is an example) is also available. Regardless of the variant used as input to the NIDL compiler, the output is the same.

```
[uuid(334033030000.0d.000.00.87.84.00.00.00), version(1)]
(1)
interface bank {
    import
        "nbase.imp.idl";                                    (2)
    typedef                                                 (3)
        long int    bank$acct_t;
    typedef
        char bank$acct_name_t[32];
    void bank$deposit(                                      (4)
        [in]        handle_t            h,
        [in]        bank$acct_t     acct,
        [in]        long int                amount,
        [out]   status_$t           *status
    );
};
```

Example Interface

Object-Oriented Binding

One drawback of the language as described so far is that all operations are required to have a primitive **handle_t** as their first argument. This means clients need to embed these handles in their programs, and to manage the binding to servers themselves. We would like to achieve as much local-remote transparency as possible (i.e., to make programs insensitive to the location of the objects upon which they operate). Embedding primitive handles in client programs destroys much of this transparency. To relieve clients of the need to manage these handles, we introduced the notion of **object-oriented binding.**

Object-oriented binding comes into play when the first parameter to an operation is not a **handle_t**. In this case, the type is taken to represent some more abstract, client-oriented handle. Since to ac-

tually make remote calls, a **handle_t** is required, some way is needed to translate the abstract handle into a **handle_t**. The person who creates the abstract type is thus obliged to write a procedure to do the conversion. This procedure is assumed to have the name *type*_**bind** (where *type* is the type name of the abstract handle) and is automatically called from stubs when the remote call is made. You can view the abstract handle as an *object* (in the Smalltalk sense) which supports the **bind** operation.

To make this more concrete, we could reformulate the above bank example in terms of object-oriented binding. Instead of taking a **handle_t** as its first parameter, **bank$deposit** could take a bank name, of type **bank$name**. The NIDL compiler would generate a call to **bank$name_bind** to translate from a bank name to the primitive **handle_t**. This routine would probably call upon some sort of naming server to look up the bank location. The bind routine might also choose to cache location information to make later translations faster.

Object-oriented binding hides the details of handle binding from the client and allows interfaces to be designed in a more abstract, client-oriented fashion. This provides a higher level of local-remote transparency than other systems which always require the client to manage handles or explicitly name the remote host on each call.

Marshalling Complex Types

In the section on NIDL language constructs, we stated that pointers could not be nested. The reason is that such nesting would require the NIDL compiler to generate code to transmit general graph structures. However, permitting only top-level, non-nested pointers can be a severe limitation in the design of an interface. For example, it excludes passing tree data structures to remote procedures.

To provide an escape from this restriction, NIDL allows a type to have an associated "transmissible" type. The transmissible type is a type that the NIDL compiler does know how to marshall. Any type that has an associated transmissible type must have a set of procedures to convert that type to and from its transmissible type. In the example of the binary tree, the transmissible type could be an array. The **tree$to_xmit_rep** procedure would walk the tree to build a representation of it in the array, and the **tree$from_xmit_rep** procedure would reconstruct the binary tree from the array.

Transmissible types may be associated with any type, not just types using nested pointers. Bitmaps are an example. It may be represented internally as a fixed size array of integers. Even though the NIDL compiler is capable of marshalling this, it may be more efficient to have it transmitted in a run-length encoded (RLE) form. So the bitmap type could have an associated RLEBitmap type, and a set of procedures for converting to and from the RLE form.

Network Data Representation

Communicating typed values in a heterogenous environment requires a data representation protocol. A data representation protocol defines a mapping between typed values and **byte streams**. A byte stream is a sequence of bytes indexed by nonnegative integers. Examples of data representation protocols are Courier [13] and XDR [10]. A data representation protocol is needed because different machines represent data differently. For example, VAXes represent integers with the *least* significant byte at the low address and 68000s represent integers with the *most* significant byte at the low address. A data representation protocol defines the way data is represented so that machines with different local data representation can communicate typed values to each other.

NCA includes a data representation protocol called Network Data Representation (NDR). NDR defines a set of data types and type constructors which can be used to specify ordered sets of typed values. NDR also defines a mapping between ordered sets of values and their representations in messages.

Under NDR, the representation of a set of values consists of two items: a **format label** and a byte stream. The format label defines how scalar values are represented (e.g. VAX or IEEE floating point) in the byte stream; its representation is fixed by NDR as a data structure representable in four bytes.

NDR supports the scalar types boolean, character, signed integer, unsigned integer, and floating point. Booleans are represented in the byte stream with one byte; false is represented by a zero byte and true by a non-zero byte. Characters are represented in the byte stream with one byte; either ASCII or EBCDIC codes can be used. Four sizes of signed and unsigned integers are defined: small, short, long, and hyper. Small types are represented in the byte stream with one byte, short types with two bytes, long types with four bytes,

and hyper types with eight bytes. Either big- or little-endian representation can be used for integers; two's complement is assumed for signed integers. The two sizes of floating-point type are single and double. Single floating-point types are represented with four bytes and double floating-point types use eight bytes. The supported floating-point representations are IEEE, VAX, Cray, and IBM.

In addition to scalar types, NDR has a set of type constructors for defining aggregate types. These include fixed size arrays, open arrays, zero terminated strings, records, and variant records.

Fixed sized arrays have a known number of elements. Their values are represented in the byte stream simply as a sequence of representations of the values of the elements. Each element value is represented according to the element type of the array. Open array types have a fixed first index value and element type but their final index value is not known from their type. Therefore, it is necessary to represent the value of the index of the last element in the array immediately before the representation of the values of the array elements.

Zero terminated strings can be viewed as a special case of open arrays; they are open arrays of characters whose last index value is defined by a terminating zero byte. To support this common data type in an efficient manner, NDR represents such values with an explicit length value followed by the characters of the string including the terminating zero character.

Record values are represented in the byte stream by representations of the values of their fields in the order defined by the record type. Variant records are assumed to have an initial set of fixed fields which includes a tag field used to discriminate among the possible variants. Representations of the values of the fields of the selected variant follow the representations of the values of the fixed fields of a variant record value.

Some types may appear to be missing from NDR. NDR has no enumerated types, bit set types, or a pointer type constructor. The definition of a NIDL maps such types onto their representations in an NDR byte stream. For example, NIDL maps enumerated types and bit sets onto the NDR unsigned integer type of the appropriate size. Typed pointer values are mapped into the NDR type which represents the type that the pointer references.

NDR is abstract in that it does not define how the format label and the byte stream are represented in packets. The NIDL compiler and the NCA/RPC protocol are users of NDR: They work together to generate the format label and byte stream, encode the format label in packet headers, fragment the byte stream into packet-sized pieces, and put the fragments in packet bodies.

The important features of NDR are its flexible representation of scalar values, its use of natural alignment, and its extensibility.

By using a format label to specify an interpretation of the scalars in a byte stream NDR supports a "recipient makes it right" approach to data conversion in a heterogenous environment. A sending process can use its preferred encoding of scalars when constructing a byte stream providing that it is one of the defined options. A receiving process needs to convert data representations only when the format specified in the incoming format label differs from its own preferred format. Thus, two compatible machines can communicate efficiently without needing to convert to a conventional network format and back again on each transmission. NDR defines a broadly useful but not universal set of scalar formats. We believe that our choices are reasonable for promoting heterogenous network computing combining workstations and special purpose server machines. On the other hand, it is important to keep the space of possible formats to a reasonable size because each recipient needs to convert any incoming scalar format to its own.

NDR requires that values be natually aligned in the byte stream. Natural alignment means that all values of size $2\hat{}n$ are aligned at a byte stream index which is a multiple of $2\hat{}n$, up to some limiting value of n; NDR choses this limit to be 3. (Scalars of size up to eight bytes are naturally aligned.) This permits, but does not require, implementations of NCA to align buffers for the byte stream so that stub code can use natural operators to manipulate values in the byte stream efficiently and without alignment faults. This also helps to promote communication ease between different kinds of machines in a heterogenous environment.

By its use of a format label NDR is an extensible data representation protocol. The format label could be extended to specify other aspects of the data representation such as packing disciplines, dynamic typing schemes, new encodings of scalars, or new classes of scalars.

The NCS NIDL Compiler and Stub Functions

NCS includes a compiler which mediates between NIDL on the one hand and NDR and the NCS runtime on the other. The functions of the compiler are: checking the syntax and "semantics" of interface definitions written in NIDL; translating NIDL definitions into declarations in implementation languages such as C; and generating client and server stubs for executing the remote operations of an interface.

The NIDL compiler is organized as a front-end component and a back-end component. The front-end parses and checks an interface definition and produces an abstract syntax tree (AST) intermediate form. If the interface definition is sound, the front-end then passes this tree to the back-end which generates implementation language include files and stub code files for the interface.

NCS's NIDL compiler is implemented for portability in C using Yacc and Lex. It is available in source form to encourage its use and extension in heterogeneous networked environments.

NIDL Compiler Functions

Distributed object-oriented programming imposes certain restrictions on the semantics of interfaces. It is part of the compiler's job (along with the design of NIDL) to enforce these restrictions. We illustrate the front-end's semantic checks with some examples. All types used in a definition must be well defined. All parameters and fields whose type is an open array require the use of a **last_is** attribute to give their size at call time. Every remote interface requires a UUID. Every operation of an interface requires an implicit or explicit handle parameter to support object-oriented programming.

The second major function of the NIDL compiler is to derive files which declare the interface's constants, types, and operations in the languages in which client applications and servers are written. These files are included in client and server programs which use or implement the remote operations of an interface. For the current implementation the supported languages are C and Pascal. Generating these files is done by a fairly straightforward walk over the AST; adding the capability to generate include files in other ALGOL-like languages would be a simple exercise.

In addition to declaring the constants, types, and operations of an interface, the derived include files declare two important statically initialized variables defined for each interface. One is the **interface specification (ifspec)** which encapsulates the identity of the interface and its salient properties (number of operations, well-known ports used, etc.). The ifspec variable is used in the binding and registering operations of the NCS runtime. The second variable is the server **Entry Point Vector (EPV)** which holds pointers to the server side's stub routines. This EPV variable is used by a server process when registering as a server for an interface; it is used by the NCS runtime to dispatch incoming calls.

The third major function of the NIDL compiler is to generate files of stub code for the operations defined in an interface. There are two such files — one contains client side stub routines and the other contains server side stub routines. This emitted code is in standard C, which we use as a universal assembler to promote portability. Each operation in an interface gives rise to a client stub routine and a server stub routine. The following section discusses the functions of these routines.

Stub Functions

Client stub routines are called by clients of an interface; they have the same interface as the operation for which they stand in. Server stub routines are called by the server side NCS runtime; their interface is defined by NCS. Client stub routines call the client side NCS runtime to perform remote calls. Server stubs call the manager's implementation of an operation to provide the actual service. Thus, the first function of stubs is to hide the NCS runtime from users and implementors of remote interfaces and to create the illusion of accessing a remote procedure as though it were local.

To communicate input and output arguments and function results between callers and called routines the stub must **marshall** and **unmarshall** argument values into call and reply packets. This is done in accordance with NDR and the conventions of NCS. Unmarshalling code is also responsible for detecting and performing necessary data conversions by comparing the incoming format label with the local formats. Data conversion is done by a combination of inline code and support operations in the NCS runtime.

The stubs also need to calculate the size requirements for call and reply packets based on the dynamic size of input and output argu-

ments. The size information is used to determine whether or not a pre-declared packet on the stack is large enough. If not, the stubs need to allocate and free storage for packets. It is *not* the job of the stub to break up a large packet into pieces that can be sent over the network — the NCS runtime provides the capability of handling arbitrarily sized packets.

Client side stubs map the operations of an interface to the operation number used by the NCS runtime to identify operations; they also pass options designating the desired calling semantics and the ifspec derived from the NIDL declaration of an operation to the NCS runtime's remote call primitive.

On the server side, the stub routines are responsible for managing storage to be used as the server side surrogates for dynamically sized arguments. This is necessary to support the server's illusion of large data structures passed to it by reference.

The stubs also manage the more elaborate features of NIDL described in section 3 above. Client stubs support automatic binding by calling users' binding and unbinding routines when necessary. Implicit handles are made explicit to the NCS runtime by client stub routines. Users' marshalling routines are invoked as necessary by both client and server stubs as part of marshalling input and output arguments of the appropriate types.

In summary, the stub generation function of the NIDL compiler automates the production of a large amount of protocol code based on a routine's interface defintion. This is important because the code is complex enough to make its hand coding very error prone and tedious. Hand producing this kind of code has been a major impediment to building distributed systems in the past.

Location Broker

A highly available location service is a fundamental component of a distributed system architecture. Objects representing people, resources, or services are transient and mobile in a network environment. Consumers of these entities cannot rely on a priori knowledge of their existence or location, but must consult a dynamic registry. When consumers rely solely on a location service for accessing objects, it becomes essential that the location server remain available in the face of partial network failures.

The NCA **Location Broker** (NCA/LB) protocol is designed to provide a reliable network-wide location broker. This protocol is defined by a NIDL interface and is thereby easily used by any NCA/RPC based application.

The NCA/LB, unlike location services like Xerox SDD's Clearinghouse [8] or Berkeley's Internet Name Domain service (BIND) [11], yields location information based on UUIDs rather than on human readable string names. The advantages of using UUIDs were described earlier.

Locating

An object's type manager must first advertise its location with the Location Broker in order for that objected to locatable. A manager advertises itself by registering its location and its willingness to support some combination of specific objects, types of objects, or interfaces. A manager can choose to advertise itself as a global service available to the entire network, or limit its registration to the local system. Managers that choose the latter form of registration do not make themselves unavailable, but rather limit their visibility to clients that specifically probe their system for location information.

Clients find objects by querying the Location Broker for appropriate registrations. A client can choose to query for a specific object, type, interface, or any combination of these characteristics. When operations are externally constrained to occur at a specific location, a client can choose to query the location broker at the required system for managers supporting the appropriate object.

Location Broker Organization

The Location Broker is divided into two components. The **Global Location Database** is a replicated object containing the registration information of all globally registered managers; the processes that manage this database are called the **Global Location Broker**. The NCS runtime implementation of the Global Location Broker uses the **Data Replication Manager (DRM)** to maintain the database. DRM provides a weakly consistent replicated KSAM package. Weak consistency implies that replicas of the Global Location Database object may be inconsistent at any time, but, in the absence of updates, that all replicas will converge to a consistent state within a finite amount of time. This form of consistency provides a high degree of both read and update availability to the Global Location Database. It is not necessary to be able to communicate with all replicas of the object to affect a change in the registration database. The DRM assumes the responsibility of propagating updates to the replicas in a timely fashion.

A **Local Location Broker** supports managers that wish to limit their registration to the local system. Access to these registrations if provided in two ways. A client can directly query the Location Broker at specific node to determine the objects and managers that are registered there. Alternately, a client can simply execute a remote operation while supplying an incompletely bound handle (i.e., one which specified only an object and system, not a particular server process). Remote calls made using such a handle are delivered to the Local Location Broker, which serves as a forwarding agent if an appropriate manager has registered itself locally. This mechanism obviates the need for users of the NCA to use well-known ports.

The division of the Location Broker into two distinct entities is, to a large degree, an NCS runtime implementation decision. Logically the Local Location Database object and the Global Location Database object are a single partitioned object, and, in fact, access to these databases is provided through a common set of operations which select the target based on lookup keys.

The NCA/RPC Protocol and NCS Implementation

The NCA/RPC protocol is designed to be low cost for the common cases and independent of the underlying network protocols on top of which it is layered. The NCS runtime implementation of the NCA/RPC protocol is designed to be portable.

Protocol

The NCA/RPC protocol is designed so that a simple RPC call will result in as few network messages and have as little overhead as possible. It is well known that existing networking facilities designed to move long byte streams reliably (e.g., TCP/IP) are generally not well suited to being the underlying mechanism by which RPC runtimes exchanges messages. The primary reason for this is that the cost of setting up a connection using such facilities and the associated maintenance of that connection is quite high. Such a cost might be acceptable if, say, a client were to make 100 calls to one server. However, we don't want to preclude the possibility of one client making a call to 100 servers in turn. In general, we expect the number of calls made from a particular client to a particular server to be relatively small. The reliable connection solution is also unacceptable from the server's perspective: A popular server may need to handle calls from hundreds of clients over a relatively short period of time (say 1-2 minutes). The server does not want to bear the cost of maintaining network connections to all those clients.

The well-known way of getting around the well-known problem of using reliable network connections is to make the RPC protocol implement exactly the reliability it needs on top of an *unreliable* network service (e.g., UDP/IP). This approach has the additional advantage that some systems (e.g. embedded microprocessors) can not or do not support *any* reliable network service; however, if they're connected to a network at all, you can be sure that they'll at least supply an unreliable service. Further, unreliable services tend to be more similar across protocol suites than do reliable services. (For example, some reliable protocols might return errors immediately if the network partitions even though a virtual circuit is currently idle, while others might defer until the next time I/O is attemped.) This similarity means that the RPC protocol can be accurately implemented in more protocol suites than if it would be possible if it assumed a reliable service.

All that the NCA/RPC protocol assumes is an underlying unreliable network service. The protocol is robust in the face of lost, duplicated, and long-delayed messages, messages arriving out of order, and server crashes. When necessary, the protocol ensures that no call is ever executed more than once. (Calls may execute zero or one times and, in the face of network partitions or server crashes, the client may not know which.)

The NCA/RPC protocol operates roughly as follows. The client side sends a packet describing the call (a **request** packet) and waits for a response. The server side receives and dispatches the request for execution, and sends a packet in response that describes the results of executing the call (the **response** packet). If the client doesn't receive a response to a request within a particular amount of time, it can inquire about the status of the request by sending a **ping** packet. The server either sends back a **working** packet, indicating that execution of the request is in progress, or a **nocall** packet, which means that the request has been lost (or that the server has crashed and rebooted) and the client needs to resend it. The protocol gets slightly more complicated if the input or output arguments do not fit into one packet.

If a called procedure is non-idempotent, the protocol ensures that the server executes the call at most once. To detect old (duplicate) requests, the server keeps track of the sequence number of the previous request for each client with which it has communicated. However, the server considers this information to be discardable and it may discard it if it hasn't heard from the client in a while, i.e., there is no permanent "connection" between the client and server.) Thus, it is possible for a long-delayed duplicate request to arrive after the server has discarded the information about the requesting client. To handle this case, the server **calls back** to the client (using an idempotent remote procedure call) to ask the client for the client's current sequence number. The server then uses the returned sequence number to validate the request. Note then that for calls to non-idempotent procedures (with input and output arguments that fit in a single packet), a total of two message pairs will be exchanged between client and server for the simple case. Subsequent calls between the same client and server will require just one message pair. Note that the extra message pair in the first case could conceivably be eliminated if the server were willing to hold onto client sequence number information for long enough to ensure that all duplicate requests had been flushed from the network. We chose not to take this approach since any time interval we considered long enough (e.g., one minute or more) seemed too long to oblige the server to hold the information.

Also, for non-idempotent procedures, the server side saves and periodically retransmits the response packet until the client side has acknowledged receipt of the response. If the server side receives a retransmission of the request, it resends the saved response instead of re-executing the call. The client side acknowledges the response either implicitly, by sending a new request, or explicitly, by sending an *acknowledgment* packet. The protocol also handles the case in which the server has executed the non-idempotent call but, because of network partitions or a server crash, fails to send the response packet.

If a called procedure is idempotent, the protocol makes no guarantees about how many times the procedure is executed. On idempotent requests, the server side does not save the results of the operation once it has sent back the response packet. In addition, the client side is not required to acknowledge the receipt of responses to idempotent requests.

Runtime

The NCS RPC runtime is written in portable C and uses the BSD UNIX *socket* abstraction. (In terms of the socket abstraction, it uses SOCK_DGRAM-style sockets.) This abstraction is intended to mask the details of various *protocol families* so that one can write protocol-independent networking code. (A protocol family is a suite of related protocols; e.g. TCP and UDP are part of the DoD IP protocol family; PEP and SPP are part of the Xerox NS protocol family.) In practice, however, the socket abstraction has to be extended in several ways to make it possible to write truly protocol-independent code. We extended the socket abstraction via a set of operations implemented in a user-mode subroutine library; the NCS runtime uses these extensions so that it can be truly protocol-independent. Bringing up the NCS runtime on a new protocol family should *not* require any changes to the NCS runtime proper. All that should be required is to add some relatively trivial routines to the socket abstraction extension library.

NCS is careful about creating sockets. Sockets are a fairly scarce resource and tying lots of them up for a long period is not a good idea. NCS keeps of small private pool of sockets. One is pulled from the pool when a process makes a remote call. When the call completes, the socket is returned to the pool. The pool need contain only one socket for the entire process if the system supports

only one thread of control per process (as is the case in standard UNIX).

The use of the socket abstraction at all could be considered to be too much of a BSD-ism, thus reducing the portability of the runtime. Fortunately, two factors argue against this point of view: First, it appears that AT&T System V, Release 3 will support at least a sufficient subset of the socket calls (layered on top of their own networking model). Second, even if the target of a port doesn't have anything resembling the socket interface, NCS use of the interface is fairly simple and it wouldn't be too hard to implement the BSD calls in terms of whatever the target system supplies.

Future Directions

NCA and NCS represent the first step in a complete network computing environment. One of the guiding goals in the development of NCA has been *transparency*. This has a number of aspects: replication, failure, concurrency, location, and name transparency.

With replication transparency all copies of an object can be considered equivalent. The user of an object cannot tell whether it consists of a single copy or many. The DRM provides replication transparency in the case where some short-lived inconsistencies can be tolerated. Future versions of NCA will include support for strongly consistent replication.

Location transparency allows users to access objects without specifying where the objects are. Objects are free to be moved around the network to adapt to changing load conditions and the availability of new hardware. The Location Broker provides the ability to find the location of objects prior to their first use. We would like to be able to have objects move at any time during program execution.

Concurrency transparency supports the illusion that a given client is the sole user of an object. NCS addresses this partially through concurrent programming support which provides a simple locking facility. In the future, we would like to address this, and to some degree, failure transparency, through the use of an object-oriented atomic transaction facility.

Failure transparency, i.e., the ability of components of a distributed system to fail and recover transparently to their users, is largely a function of location and replication transparency. By replicating objects, when a given replica fails another is available to takes its place. Location transparency hides the switch from one replica to another from the user.

Neither NCA nor NCS address the issue of name transparency at this point. We anticipate building a general purpose name server in a future version of NCS. In addition, we intend to address a higher-level form of naming: In many instances, it is more convenient to find an object by attributes rather than by a text name. An *attribute broker* will provide this ability. Thus, a client will be able to query the attribute broker for a list of "26 page/sec laser printers" rather than managing the mapping between machine names and attributes itself.

Most of the focus in the NCA development so far has been on getting the basic model right. Once the object-oriented model is in place, we feel that these higher level services will evolve naturally. Had we started with a more traditional process-oriented model, the level of integration and transparency we desire would be much more difficult to achieve.

References

[1] Guy T. Almes. Integration and distribution in the Eden system. Technical Report 83-01-02, Department of Computer Science, University of Washington. 1983.

[2] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, II(1): pp. 39-59. 1984.

[3] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. *Proceedings of the Fifth Symposium on Operating Systems Principles*, pp. 141-160. ACM Special Interest Group on Operating Systems. 1975.

[4] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley. 1983.

[5] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pp. 148-159. ACM Special Interest Group on Operating Systems. 1981.

[6] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton and Paul H. Levine. UIDs as Internal Names in a Distributed File System. In *Proceedings of the Symposium on Principles of Distributed Computing*, pp. 34-41. Association for Computing Machinery. 1982.

[7] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAL-I(5): pp. 842-857. 1983.

[8] D. C. Oppen and Y. K. Dalal. The Clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems* I(3): pp. 230-253. 1983.

[9] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley. 1986.

[10] Sun Microsystems. Networking on the Sun workstation. Part no. 800-1324-03. 1986.

[11] D. B. Terry, M. Painter, D. Riggle and S. Zhou. The Berkeley Internet Name Domain Server. *Proceedings of the USENIX Association Summer Conference*, pp. 21-31. 1984.

[12] W. Wulf, R. Levin, C. Pierson. Overview of the Hydra operating system development. *Proceedings of the Fifth Symposium on Operating Systems Principles*, pp. 122-131. ACM Special Interest Group on Operating Systems. 1975.

[13] Xerox Corporation. Xerox System Integration Bulletin, OPD B018112. 1981.

An Extensible I/O System

by

Jim Rees, Paul H. Levine, Nathaniel Mishkin, Paul J. Leach

Introduction

For years, programming environments have provided **device independent** program I/O. The programmer normally codes file I/O requests using a standard set of procedure calls, such as the UNIX **open**, **close**, **read**, and **write** system calls, or language specific I/O calls. This model enables a program written primarily to perform I/O to simple files to also read from keyboards or IPC channels, and to write to display windows or IPC channels without any modification. The intent is to unburden the programmer from the necessity of either binding the program to a specific target for its I/O or enabling the program to adjust to the vagaries of different I/O targets at program run-time; that is, to make the applications program I/O independent of target type.

While this concept has been around for a long time, the systems that implemented the concept have generally had one major shortcoming. The only way to add a new type of I/O target to the system was to modify the system source. In the case of UNIX operating systems, for example, it is necessary to modify and rebuild the operating system kernel and to have all of the software that implements the management of the new I/O target permanently wired into physical memory. Most schemes for adding new file types to the UNIX kernel operate at the file system level, so that within a given file system, all files have the same type. Further, whenever a new type is added, various pieces of the system have to be modified to behave correctly with respect to the new type. Because of this sizable burden, programmers are discouraged from defining numerous I/O target types.

Our goal was to create a framework in which file I/O could be truly extensible — to allow users to define new types without modification to the basic system. Our work consisted of building a general framework for extensibility and then applying those techniques to stream I/O. We call the framework a **typed object management system**; and the associated file I/O facility **Extensible Streams** (ES). The combination of these two is called the Domain Open System Toolkit.

The system resulting from our work is novel because it:

- Supports (relatively large) typed, permanent, sharable objects in a distributed file system.
- Allows users to define new types of objects.
- Allows users to associate generic procedures (operations) with types; the procedures are dynamically loaded into the address space of processes when the procedure is invoked.

The Open System Toolkit allows users to extend the Domain file system by inventing new file types and writing managers for these types. The current implementation allows dynamic creation of new types, and dynamic binding of typed objects to the **managers** which implement their behavior. Type managers are written and debugged as user programs and require no kernel modifications for installation. This system has been used successfully to write and debug new device drivers, to add new types of files, and to provide remote file system interconnects to foreign file systems.

Domain Architecture

The Domain system [3] is an architecture for networks of personal workstations and server computers that creates an integrated distributed computing environment. A major component of this distributed system is a distributed file system [4] which consists of four major components: the object storage system, mapped file management, concurrency control and naming service.

The Domain distributed object storage system (OSS) provides location transparent typed object management across a network of loosely coupled machines. We say "object" rather than file to specifically include all of the named non-disk objects in a computing

environment, such as devices (serial I/O lines, magtapes, null, etc.), IPC facilites (sockets, etc.) and processes. While a naming service manages a network-wide hierarchical name space, at the OSS level objects are named by a 64-bit **unique identifier** (UID). The UID consists of a timestamp and a unique node ID. This guarantees that the UID is unique across all Domain nodes for all time.

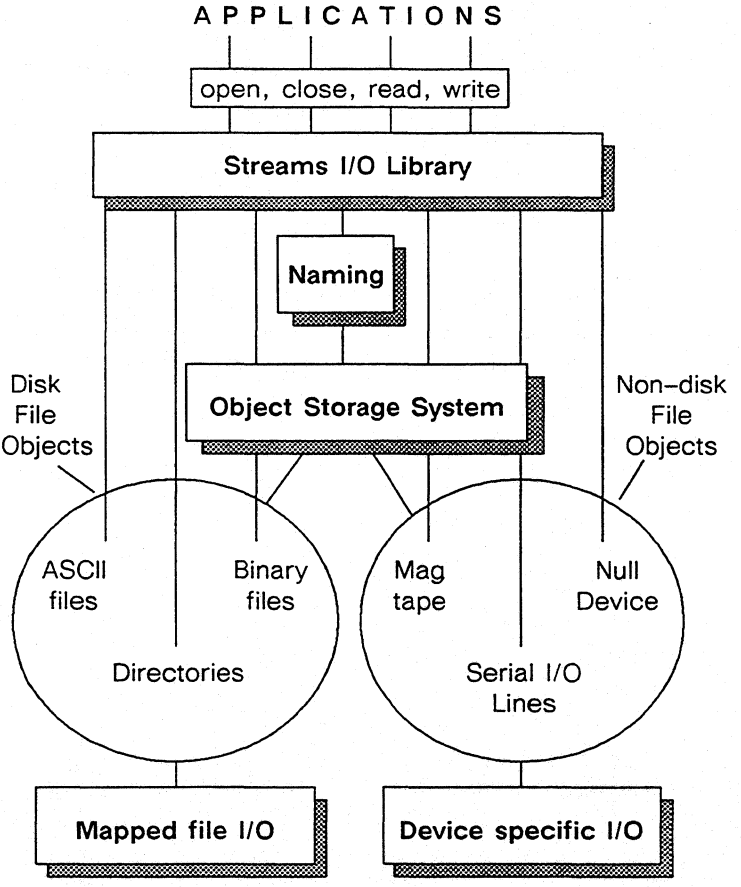
A 64-bit **object type UID** is associated with every object. This type is used to divide the set of all objects into classes of like objects; all of the objects in a class have common properties and must be operated upon by a single set of procedures. We use a UID (rather than any other kind of type identifier) because a system facility supports the unique creation of these 64-bit numbers across all Apollo products. In the basic Domain system there are several types, including ASCII text, binary, directory, and record. This strong typing allows the creator of an object to explicitly specify its intended use and interpretation, rather than depending on the conventions and cooperation of other users and programs.

The Domain OSS supports a consistent set of facilites for naming, locating, creating, deleting, and providing access control and administration over all objects. Each object has an inode, which we have extended to contain (among other things) the type UID of the described object.

For disk-based objects OSS also provides storage containers (arrays of pages) for uninterpreted data. A process accesses this data by handing the kernel the object's UID and asking for it to be **mapped** into its address space. The process then uses ordinary machine instructions to directly manipulate the contents of the object — the single-level store (SLS) concept of Multics [6], Pilot [7], and System/38 [1].

Layered on top of the file system is the **Streams** library, a user state library mapped into every process's address space, which provides a traditional I/O environment for programs. The Streams library implements the standard I/O interfaces and so provides equal access to both disk and non-disk resident objects. The Domain Stream operations form a superset of the UNIX file I/O operations, as they include record-oriented operations and more inquiry operations but are all based on a file descriptor returned to callers of **open**. Streams is an object-oriented facility in that its behavior is determined by the type of object to which its operations are applied. When a stream operation is invoked, Streams calls the manager that handles operations for the type of the object being operated

on. The following figure diagrams the relationships among the various pieces of the Domain object management system and Streams.



The relationships among the various pieces of the Domain object management system and Streams.

Typed Object Management

The fundamental concept underlying the object–oriented part of our system is the notion that every object is strongly typed and that for each object type there is a set of executable routines that implement a well–specified group of **operations** on that type. This section describes the object typing strategy, defines operations and describes their partitioning into traits. It also explains the management facilities necessary to associate typed objects with the code that implements the operations defined for them.

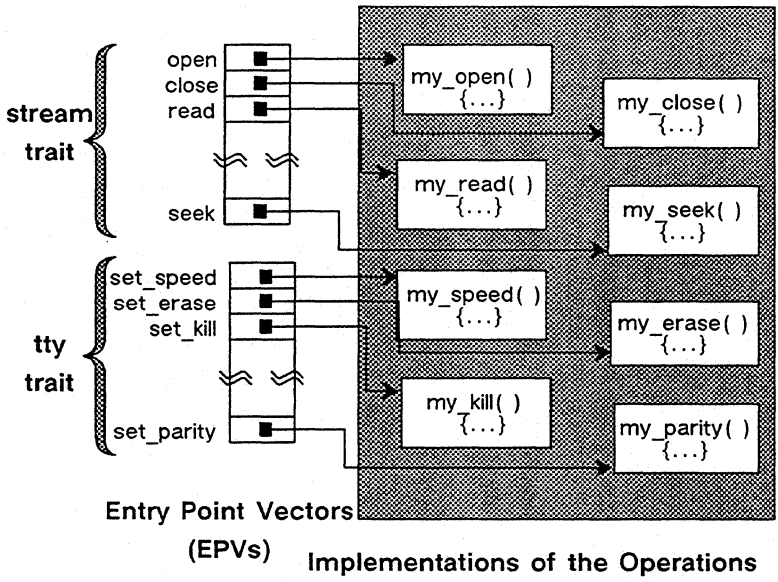
UNIX file system objects do not have an explicit type tag, but do keep a form of type information in several different places. The *mode* field in a file's inode contains some bits that distinguish among ordinary files, directories, character and block special files (devices), and depending on the version of UNIX system, FIFOs, sockets, textual links, and other types of file system objects. There may also be type information coded into the major and minor device numbers to, for example, distinguish between tape drives and disk drives. In some cases, type information is encoded in the first few bits of the file data itself. For instance, there may be "magic numbers" for tagging various flavors of executable (**a.out**) files.

In the Domain system, the type tag is a UID which is explicitly attached to the object at the time that the object is created. This provides the advantage of a single, common mechanism to distinguish among all types. The use of a UID (rather than a small integer) allows the arbitrary creation of new types without appealing to a central authority.

The fundamental concept underlying the object–oriented part of our system is the notion of an object type as a set of legal states together with a collection of **operations** that implement the state transitions. Operations can be viewed in two ways: as a specification of how to invoke a transformation on the state of an object, or as the executable code that performs the transformation. The collection of code that implements the set of operations for an object type is known as that object type's **type manager**.

A **trait** is an ordered set of operations. It represents a kind of behavior that a client desires from an object. For example, the operations **open**, **close**, **read** and **write** could be a "stream-like" trait, and the operations **set speed** and **echo input** could be part of a

"tty-like" trait. An object supports a trait if its type manager implements the operations of the trait. For every trait that a type manager supports, the manager provides an **entry point vector (EPV)**, that is an ordered list of pointers to the procedures that implement the operations in the trait.



Object Type Manager

The type manager consists of the routines that implement one or more sets of operations (traits) and the entry point vectors (EPVs) that map the supported operations to the routines that implement them.

The implementation of the typed object management system has two main components: the **type system** and the **trait system**. We use the name **Trait/Types/Managers (TTM)** to refer to these two components plus the set of all type managers.

The type system is responsible for maintaining a data base containing mappings between type UID, type manager, and type name. New types can be created at will. For convenience, there is a name for every type, but a type UID rather than a type name is actually

attached to the file system objects. This guarantees that all types are unique, even if two different implementors independently choose the same name. The type system provides procedures that can be used to create new types, associate a name with a type, and look up type UID of a given type name. It can also find the manager for a given type.

The role of the trait system is to bind *<object, trait>* pairs to type manager EPVs. It provides the **trait_$bind** call for this purpose. This call looks up the object's type UID and then asks the type system for the corresponding type manager. Object code libraries containing managers are not pre-linked with client object code. Rather, the trait system is responsible for dynamically loading them into the address space of clients as necessary. To perform this task, the trait manager uses the type system to locate the manager object code file. It then loads the manager into the address space of the client. The type manager is linked as an autonomous program whose main entry point is called when the manager is loaded. The code at this entry point registers all supported *<trait, EPV>* pairs with the trait system. Once the manager is loaded, the trait system returns the requested EPV to its client.

The type definition for an EPV corresponding to a trait that describes operations on stacks might look like:

```
typedef struct {
            void          (*push)(uid_$t, stack_$elem_t);
            stack_$elem_t (*pop) (uid_$t);
} stack_$epv;
```

The actual EPV for a type manager that supported the stack trait would be declared as:

```
stack_$epv my_stack_epv = {
            my_push,
            my_pop,
};
```

where **my_push** and **my_pop** are the names of real procedures that implement the push and pop operations:

```
void my_push(obj, elem)
uid_$t      obj;
stack_$elem_t elem;
{
```

```
...
}

stack_$elem_t my_pop(obj)
uid_$t      obj;
{
...
}
```

The client uses **trait_$bind** to get a pointer to an EPV from the trait system:

```
trait_$epv *trait_$bind(obj, trait, typuidp, statusp)

uid_$t      obj; /*IN: object we want to operate on */
trait_$t    trait; /* IN: trait we want to use */
uid_$t      *typuidp; /* OUT: type of object */
status_$t   *statusp; /* OUT: status */
```

Once a client has called **trait_$bind** and received an EPV, it can invoke operations on the object. For example, to call the push and pop operations in the sample trait above:

```
epv = (stack_$epv *) trait_$bind(my_obj, stack_$trait,
&type_uid, &status);
(*(epv->push))(my_obj, an_elem);
an_elem = (*(epv->pop))(my_obj);
```

The Domain system provides a set of programs for creating and installing new types and their managers. A user who creates a new type will also typically write a type manager for that type. The manager is written as a set of subroutines, each implementing an operation for the traits that the manager supports. The programmer can use the standard debugging tools on the type manager. The manager is installed by running a program that puts the executable code in a well–known place and registers the new manager with the type system data base. No kernel modifications are required, and the machine does not have to be rebooted. There is no limit on the number of object types a single system may support since their managers are only loaded when needed.

Extensible Streams

Extensible Streams is a client of TTM. ES defines three basic traits: IO, IO_OC, and IO_XOC. The IO trait contains the traditional I/O operations — **get (read)**, **put (write)**, **seek**, etc. The IO_OC trait contains the operations **open** and **initialize**. (The IO_XOC trait is similar to IO_OC except that it supports **extended naming**, a facility that allows non-standard pathnames, described below.) ES also defines a set of **auxiliary traits** containing operations that only some type managers will choose to implement. The current set of auxiliary traits include: SIO (operations for manipulating serial I/O lines), SOCKET (operations corresponding to the 4.2BSD UNIX "socket" system calls), PAD (operations for manipulating windows), and DIRECTORY (operations for reading and manipulating directories).

ES introduces a layer of abstraction on top of the basic operations. This layer — called the **I/O Switch** — supports the notion of an **open stream** and isolates the user of file system I/O from the TTM. An open stream is created by calling the I/O Switch procedure **ios_$open** which:

- Calls **trait_$bind** to get the IO and IO_OC EPVs for the object being opened.
- Calls the manager's **open** operation. This operation returns a **handle** — a virtual address of a descriptor that is meaningful only to the manager. The manager stores in the handle whatever information it needs in order to maintain the semantics of an open stream (e.g., position in stream, buffers).
- Allocates an entry in the **stream table** — a table of open streams. Each entry in this table contains the EPVs for the IO and IO_OC traits, and the handle returned by the **open** operation.
- Returns the small integer — the **file descriptor** — that identifies the table entry allocated in the previous step. This file descriptor is used by the application program on subsequent calls.

Another I/O Switch procedure, **ios_$create**, is similar to **ios_$open** except that it creates a new object and calls the manager's **initialize**

operation. In addition to returning a handle, the **initialize** operation stores any information it needs to in the newly-created object.

For each operation in the IO trait, a trivial I/O Switch procedure takes a file descriptor as its first argument, converts the descriptor to a handle (by consulting the stream table), and calls the appropriate procedure from the EPV (also obtained from the stream table). The various forms of I/O (e.g., UNIX I/O system calls, FORTRAN and Pascal language I/O primitives) are implemented in terms of these I/O Switch procedures.

Extended Naming

Extended naming is a facility that allows the pathname of an object being opened to be augmented with additional text to be interpreted by the Streams manager of the object to which the pathname refers. This additional text is called the **residual pathname**.

If an application calls the I/O Switch's open procedure with a pathname containing a residual, and the non-residual part of the pathname names an object whose type manager implements the IO_XOC trait (as opposed to the IO_OC trait), then the I/O Switch passes the residual to the manager as one of the arguments in the IO_XOC **open** operation. The manager is free to interpret the residual in any way it chooses.

Program–level I/O based on a simple system naming facility allows an application program to pass the name of a file system object into the **open** call, for the I/O Switch to locate the specified object, and for the manager of that type of object to then do its job. For example, the pathname **/usr/fonts/classic** refers to the object whose name is **classic**, which is catalogued in the directory whose name is **fonts**, which in turn is catalogued in a directory object whose name is **usr**. The I/O Switch resolves the entire pathname down into the single target object, and passes a shorthand identifier for that object to the manager.

The intent of extended naming is to allow the object managers themselves to take over part of the pathname–walking responsibility so that they can manage a collection of objects that can be distinguished by the remainder of the pathname. To clarify this notion, consider the following.

The pathname **/jim/test.c** would normally be interpreted as a file named **test.c** catalogued in the directory named **jim**. The name also suggests that the file is a C language source file and that all operations that would need to work on such a file (e.g., compiling, printing, editing) could be requested by specifying this name.

Now let's suppose that file **test.c** is of type history. The actual file system object contains the entire change history of the file, much the same way that a SCCS [9] file does. Programs that do not care about the change history can open this file and read from it. The **open** and **read** requests are passed on by the I/O Switch to the history type manager, and the manager can be written so that the program always reads the latest version of the file.

Extended naming takes the concept one step further by allowing the manger writer for the history object type to allow the specification of additional pathname text. Where the simply specified pathname results in the reading of data from the *latest* version of the file **test.c**, the manager writer might wish to allow a naming syntax of the form **/jim/test.c/-1** to indicate that the application wishes to use the penultimate version of the file instead of the newest. The I/O Switch allows this additional specification to be issued at the application program layer and passed through to the manager for the target object.

The application passes the pathname (with the extended name) to the I/O Switch **open** routine. The **open** routine evaluates the pathname one pathname component at a time walking from left to right. In the current example, **jim** is a directory where the name **test.c** is located. **test.c** is discovered to be a history file (not a directory), and because the original pathname still has remaining text ('-1') that the I/O Switch cannot resolve, it passes that remainder to the history object manager's IO_XOC **open** routine. The history manager is then able to decide what text to provide to subsequent **read** requests and the intended result occurs. In this case, the application program is not affected by the apparent peculiarity of the original pathname. The I/O Switch avoids confusion by only walking the pathname through objects that support the directory trait and the manager is able to get whatever information it needs to do the job it was written to do.

Other examples of extended names a *history* manager might be willing to accept are:

/jim/test.c/03.02.85
/jim/test.c/original
/jim/test.c/yesterday

Another example of the application of extended naming is a gateway to a non-Domain file system. For example, imagine an object whose name is **THEM** and whose type is UNIX_gate. A pathname of the form **/gateways/THEM/usr/jan/test.c** could be passed by an application program to the I/O Switch. The Switch would see that the object named **gateways** was a directory and would look the name **THEM** up in that directory. **THEM** would be found to be a UNIX_gate object, and since the Switch cannot walk the pathname through objects that are not directories, it would call the UNIX_gate object manager's **open** routine. That routine is passed the UID for the object whose name is **THEM** and the remaining pathname (**/usr/jan/test.c**). The **UNIX_gate** manager then has the information it needs to contact a remote file service for the data it needs to meet the demands of the requesting application program. The protocol that the manager uses to access the remote files is entirely up to the manager writer, and because the manager runs in user space, it is not restricted to kernel services but can use any service available at the user level. This scheme has been used to build a type manager that interconnects the Domain file system with a generic Berkeley 4.2 UNIX file system, and another that connects to a VAX/VMS file system.

Underlying Facilities

Many facilities provided in the Domain environment made the implementation of TTM and Extensible Streams possible. These facilities make it possible to write OS-like functions in user space.

The underlying virtual memory system — which allows objects to be mapped into the virtual address space — is needed to give type managers low-level, yet controlled, access to the raw data in objects. The virtual memory system allows more flexible access to the address space than that allowed by **sbrk***(2)*. These calls take the name of an object, map the object into the address space, and return a pointer to (i.e., the virtual address of) the mapped object. The address space of a process can be characterized solely in terms

of what objects are mapped where. Processes are not allowed to make memory references to parts of the address space to which no object is mapped.

The read/write storage (**RWS**) facility is a flexible and efficient storage allocation mechanism. It is implemented in user space in terms of the virtual memory primitives; it maps temporary objects into the address space and allocates storage from that part of the address space. It allows storage to be allocated from multiple **pools**. One pool corresponds exactly to the type of storage allocated by **malloc**. Another pool is similar, except its state is not obliterated by **exec** calls. Type managers must use storage from this pool to hold per-process state information since open streams must survive calls to **exec**.

RWS also provides a global storage pool. The global pool is a place where storage that can be viewed from all processes' address spaces can be allocated. The allocation call returns a pointer to the allocated storage, and this pointer is valid in all processes. Type managers must use storage from the global pool to maintain things like the current position (i.e., offset from beginning-of-file) of an open stream. If a process opens a stream to an object, forks, and then the child does I/O to the stream it was passed, the parent sees the position of *its* stream change too. Thus, position information must be in storage accessible to both parent and child. Because type managers run in user space, they need a user space global storage allocater for this purpose.

The dynamic program loader allows the system to load managers as they are needed. Managers for types that are not used by a given process do not take up any virtual address space in that process. The loader is implemented in user space in terms of the RWS facility (to allocate space for static data) and the mapping calls. The pure parts of executable images are simply mapped into the address space before execution, because the compilers produce position-independent code. In 4.2BSD, only the kernel can be dynamically linked to; all other subroutines must be statically bound to the program which uses them.

The **eventcount** [8] (**EC2**) facility is the basic process synchronization mechanism. Eventcounts are similar to semaphores: eventcounts are associated with significant events, and processes can advance an eventcount to notify another process that an event has occurred, or wait on a list of eventcounts until the first event happens.

A design principle for all Domain interfaces is that for every potentially blocking procedure in an interface, there is an associated eventcount that can be obtained through the interface and that is advanced when the blocking procedure would have unblocked. This always allows programs to wait for multiple events (say, input on a TTY line and arrival of a network message) simultaneously. The 4.2BSD **select***(2)* system call is implemented in terms of eventcounts. However, unlike **select**, eventcounts can also be used to wait on non–I/O events, such as process death.

The **mutual exclusion** (**MUTEX**) facility is a user-state library that contains calls that allow multiple processes to synchronize their access to shared data (i.e., data in objects that are mapped into multiple processes). MUTEX is implemented in terms of EC2. MUTEX defines a lock record that consists of a lock byte and an eventcount. Typically, applications embed a record of this type in a data structure over which mutual exclusion must be maintained. A MUTEX lock is set by calling **mutex_$lock**, which attempts to set the lock byte (using the hardware test-and-set instruction). If it fails to set the lock byte, it waits on the eventcount; when the wait returns, **mutex_$lock** repeats the attempt to set the lock byte. **mutex_$unlock** unlocks a MUTEX lock by clearing the lock byte and advancing the eventcount. Type managers use shared storage to maintain various kinds of information. To control access to this data, managers use the MUTEX facility.

The **shared file control block** (**SFCB**) facility allows multiple processes to coordinate their access to the same object. There is various dynamic information that processes might want to keep about an object. For example, type managers need to maintain information about the object's current length, whether the object is being accessed for read or write, and whether other processes should be allowed to concurrently access the object. Since this information must be accessed by multiple processes, it must reside in global storage.

The first process to access the object can allocate the storage, but how are other processes to find the virtual address of that storage? The SFCB facility addresses this problem by maintaining a table translating object UID into global virtual address. (The table is in global storage at a well-known location.) The **sfcb_$get** call takes an object UID and returns a pointer to a piece of global storage (called the **SFCB**). If no storage was "registered" with SFCB prior to the call, an SFCB is allocated and registered under the specified UID; otherwise, a pointer to the existing storage associated with

that UID is returned and a use count field in the storage is incremented to reflect the additional "user" of the storage. **sfcb_$free** decrements the use count and, if it reaches zero, frees the storage.

Examples

Extensible Streams allows a number of special-purpose types to be defined. For example:

- History objects: objects that contain many logical versions, only one of which is presented through the open stream at a time. The residual text is used to specify a particular version; if omitted, the most recent version is presented. Useful for source control systems.

- Circular objects: objects that grow to a certain size and then have their "oldest" data discarded when more data is written to them. Useful for maintaining bounded log output from long-running programs.

- Structured documents: objects that contain document control (e.g., font and sectioning) information but which can be read through an open stream as if they were simple ASCII text. Useful for using conventional text processing tools (e.g., UNIX **grep**) [10].

- Gateways to non-Domain file systems: objects that are placeholders for entire remote file systems. The residual is used to specify a particular file on the remote system. The manager implements whatever network protocol it chooses to access the remote system's data.

- Distributed, replicated data bases: objects that, for reliability reasons, are distributed across a network of machines. A Yellow Pages [5] manager would eliminate the need for the **ypcat** command, and allow any ordinary user to access a Yellow Pages data base without modification and without having to bind to a special library (the type manager, in effect, is the library).

TTM can be used independently of Extensible Streams. For example, the Domain graphics library may be converted to use TTM. Currently, the graphics library has code for all the display hardware

types it must support. A TTM-based implementation would define multiple types, one for each type of display hardware, a trait that contains graphics operations (e.g., **move**, **draw**, **trapezoid_fill**), and a set of managers, one per type. This approach would make it possible for only the code necessary for a particular display hardware type to be loaded into the system, and for the graphics library to be easily extensible to new hardware types.

Experience

While the original Streams library was written with the idea of types and type managers in mind, the actual implementation had to be restructured substantially to take advantage of TTM. We took this opportunity to redesign the interface to managers and the interface presented to applications that use the Streams library.

The decision to implement the Berkeley socket calls in terms of a trait turned out to be a good one. On a standard Berkeley UNIX system, defining and implementing a new domain (address family) is a fairly difficult task — it requires working inside the kernel. With Extensible Streams, you need only create a new type and implement the SOCKET trait in the manager for that type. We have already implemented a manager for "Domain domain sockets." Currently, this domain supports only datagram-oriented sockets (SOCK_DGRAM) because our short-term goal was merely to allow access to specific, low-level Domain networking primitives using the generic, high-level socket calls.

The nature of the address family space made our task a bit more complicated. Address families are identified by small integers in a space over which there is no central authority. As a result, one has to simply pick an address family out of thin air and hope no one else has picked it too. It is interesting to contrast this state of affairs with the type UID approach we took in TTM, since the small integer address families are essentially type tags. The type UID approach does not have the problem of more than one person picking the same type tag. We did not have the option to change the way address families are identified, so we used a scheme in which address families are translated into type UIDs.

The socket creation primitive is called **socket_$create_type**. This calls takes a type UID (and a socket type) and returns a stream to a socket of that type. (**socket_$create_type** is analogous to **ios_$cre-**

ate except that it calls the **create** operation in the SOCKET trait instead of the **initialize** operation in the IO_OC trait.) The **socket** system call converts its address family argument into a type UID by consulting an object in the file system that contains a table translating address families into type UID. It then calls **socket_$create_type**. Note that we could have simply hardcoded a "switch" statement on address family into the implemention of *socket*, but this would have meant that **socket** would not have been as extensible as we would like. (User-defined sockets could have been created via **socket_$create_type**, but not by **socket**). The scheme we implemented is less than ideal in that it requires both that the type be created and that the address-family-to-type-UID object be updated, but it was the best we could do.

One difficult problem that we have not adequately addressed is that of expanding wildcards in an extended name. For example, using our VMS gateway type manager, one would like to type the name:

```
/gateways/my_vms_sys/dra0:[rees.*]mail.txt
```

If **my_vms_sys** is a gateway object to a VMS system, and **dra0:[rees.*]mail.txt** is a VMS file specification, this specification should be expanded to include files named **mail.txt** in all subdirectories of **dra0:[rees]**. Unfortunately, the agent doing the wildcard expansion (typically the UNIX shell) has no knowledge of the syntax of the extended part of the name, and so has no way to expand the wildcard. We considered implementing a "wildcard trait," but this is difficult to specify in a general way, and every program that does wildcard expansion would have to be modified to use this trait. Instead, we require that standard UNIX hierarchical names with / separators be used whenever wildcards are being expanded, but we also allow non-standard syntax (as in the example above) if there are no wildcards.

The semantics of certain UNIX operations turned out to be fairly obscure. For example, suppose a program sets the FAPPEND flag (via **fcntl***(2)*) to "true," then forks, then the child sets the flag to "false." Is the change to the stream state seen by the parent as well? We were frequently obliged to look at UNIX kernel source or to write sample programs and run them on a standard UNIX system to answer our questions. As we discuss below, we are led to believe that the task of producing exact semantic specification is a forbidding one. The various UNIX standards committees have their work cut out for them if they intend to do a complete job.

Another interesting experience gained during the implemention of TTM and Extensible Streams relates to the problem of documentation. The goal of Extensible Streams is to make it possible for people who are not employees of Apollo Computer to write new type managers without having access to Apollo source code. This means that the specification of the semantics of the operations must be very precise — it must completely characterize the expectations of application programs that do I/O. The creation of this specification turned out to be a non-trivial task.

Acknowledgments

In addition to the authors, James Hamilton, David Jabs, and Eric Shienbrood worked on the implementation of TTM and Extensible Streams. John Yates was involved in some of the early design work. Elizabeth O'Connell wrote most of the documentation.

References

[1] R. E. French, R. W. Collins, L. W. Loen. System/38 Machine Storage Management. *IBM System/38 Technical Developments,* IBM General Systems Division. 1978.

[2] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, Paul H. Levine. UIDs as Internal Names in a Distributed File System. *Proceedings of the 1st Symposium on Principles of Distributed Computing,* Ottawa, Canada. 1982.

[3] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, Bernard L. Stumpf. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communication,* SAC-1, 5. 1983.

[4] Paul J. Leach, Paul H. Levine, James A. Hamilton, Bernard L. Stumpf. The File System of an Integrated Local Network. *Proceedings of the ACM Computer Science Conference,* New Orleans, LA. 1985.

[5] B. Lyon and G. Sager. Overview of the Sun Network File System. Sun Microsystems, Inc. 1985.

[6] E. I. Organick. *The Multics System: An Examination of Its Structure,* M.I.T. Press. 1972.

[7] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, S. C. Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM,* Vol. 23. 1980.

[8] David P. Reed and Rajendra K. Kanodia. Synchronization with Eventcounts and Sequencers. *Communications of the ACM.* 1979.

[9] M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*. 1975.

[10] J. Waldo. Modelling Text as a Hierarchical Object. *USENIX Conference Proceedings*. 1986.

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Domain/OS Design Principles*
Order No.: 014962-A00
Date of Publication: January, 1989

What type of user are you?

___ System programmer; language _____
___ Applications programmer; language _____
___ System maintenance person
___ System Administrator
___ Manager/Professional
___ Technical Professional
___ Student
___ Novice
___ Other

How often do you use the Apollo system? _____

What additional information would you like the manual to include? _____

_____

Please list any errors, omissions, or problem areas in the manual by page, section, figure, etc. _____

_____

_____

_____

Your Name

Date

_____

Organization

_____

Street Address

_____

City State

Zip

No postage necessary if mailed in the U.S.

ut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold


014962-A00