Open Software: UNIX, DCE, and Competitors

Thomas W. Doeppner Jr.

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-59
December 1993

Open Software: UNIX, DCE, and Competitors[1]

Thomas W. Doeppner Jr.[2]

Department of Computer Science
Brown University
Providence, RI 02912-1910
USA
twd@cs.brown.edu

Abstract

Open software is software contributing to a general goal of systems that do not depend on any one hardware or software vendor and are easily extensible by adding new software built upon the functionality of existing software. We discuss some of the defining characteristics of open software and *frameworks* that ease its production. These frameworks include operating systems and software supporting distributed computing. Examples are UNIX, NT, and DCE. We also examine two technological developments that are becoming very important in the development and use of these frameworks. The first is a structuring technique for operating systems—microkernels. The second is a technique for doing concurrent and parallel computation—multithreaded programming.

1 Introduction

What is open software? It is software that contributes to a general goal of systems that do not depend on any one hardware or software vendor and are easily extensible by adding new software built upon the functionality of existing software. A proprietary operating system, i.e., one that runs only on the hardware of one particular vendor, is not open software. Its users are dependent of the vendor of the hardware—they cannot easily take advantage of the hardware of other vendors. An operating system that runs on the hardware of a number of vendors but is supplied by a single software vendor is also not open software, since its users cannot take advantage of alternative implementations supplying new functionality. A document-preparation package that does not allow the importation of text and drawings from other packages or that does not produce output that can be incorporated into documents produced by other packages is not open software. It is not extensible by means of other software—it is a closed system.

Open software is not necessarily free software. People who produce software need to make a living. How can company A and company B both produce implementations of operating system X? They could both *license* the operating system from company C. Or they could both develop the system from scratch based on specifications published by some organization D. In the first case, if company C is a competitor of A and B, then A and B, by licensing C's operating system, become subservient to C and the operating system is closed software. But if C is not a competitor of A and B but is a consortium of competing companies including A and B, then operating system X is open

1. This paper appears in the proceedings of the 1993 CERN School of Computing. It is the companion to a series of lectures given by the author at this school in September 1993.
2. This work was partially supported via ARPA Order 8225, ONR grant N00014-91-J-4052.

software. Similar arguments apply in the second case: if the organization that publishes the specification produces that specification through an *open process*, i.e., a process involving all parties affected by the specification, then software adhering to the specification is open software.

What is important to software users is that it fit within a *heterogeneous environment*, so that they are not beholden to any one particular vendor for their computing needs. This means that software should be *portable*—it should run on different machines and on different operating systems.

Furthermore, software should be *interoperable*—each software package should be able to work in conjunction with other software packages as appropriate. A simple example is the ubiquity of PostScript—programs that produce PostScript can be used in conjunction with programs that consume PostScript. In the same vein, the drawings produced by a drawing program should be easily integrated into a document produced by a desktop publishing package. A publishing package that does not accept input from other sources is closed software.

What is important to software vendors is that they be able to produce open software. This means that they need to know the *specifications* of the software with which they must interoperate. Vendors of computing platforms must be able to *license* systems software, such as operating systems.

For open software to be practical, there must be industry-wide agreement on the *frameworks* for building systems. A company that produces software to run on a number of different machines will have an easier time developing and supporting such software if it can rely on a common operating-system interface on all the target machines. Similarly, developers of software that is to fit within a heterogeneous, distributed environment can greatly benefit from a standard framework for distributed computing.

In the remainder of this paper we discuss these frameworks. First we look at operating systems, briefly comparing UNIX and NT. Then we examine two technological developments that are becoming very important in the development and use of these frameworks. The first is a structuring technique for operating systems—microkernels. The second is a technique for doing concurrent and parallel computation—multithreaded programming. Finally we discuss DCE—the OSF's distributed computing environment.

2 Operating Systems

A basic framework for building systems is the operating system. UNIX is probably the first example of an operating system that comes close to being "open." MS/DOS, in conjunction with Microsoft Windows, certainly runs on more computers than UNIX is. But MS/DOS and Windows are not portable[1] and no one other than Microsoft has any say on what goes into them—they are not examples of open software. The MS/DOS operating system has been described as "functionally impaired"—in that a great number of desirable features found in "modern" operating systems that are not found in MS/DOS.

Recently Microsoft has introduced a new operating system, NT, and a new version of Windows to run in conjunction with it. Is NT a better operating system than UNIX? Is NT an open operating system? Before discussing these questions, let's first look briefly at UNIX.

There are other operating systems on the market as well, for example, OS/2 and the Macintosh operating system. We do not wish to slight either of them, but, to keep our discussion simple, we look at only UNIX and NT.

1. However, a number of companies have recently introduced the capability to emulate MS/DOS and Windows on non-Intel hardware to the extent that one can run Windows applications at speeds comparable to its performance on Intel hardware.

2.1 UNIX

The UNIX operating system has changed considerably since its birth in the late '60s and early '70s. At first it was strictly a time-sharing operating system for small computers. In the late '70s and early '80s it evolved into a time-sharing operating system for much larger computers. In the latter stages of this period it became the standard operating system on the ARPAnet and was the base operating system for much research on distributed computing. In the mid-'80s it began its most recent evolutionary stage and became a workstation operating system (and perhaps a personal-computer operating system).

A great deal of new technology has been integrated into UNIX in the past decade, much of it in the past few years. Within the kernel of the operating system, this new technology includes advanced file systems, support for multiple threads of control within a process, shared libraries and dynamic linking, support for large address spaces, enhanced security, and support for multiple processors. New technology provided on top of the kernel includes multiple GUIs (graphical user interfaces) and internationalized libraries.

2.2 NT

Is NT better than UNIX? In terms of the functionality provided by the operating systems themselves, there appears to be little difference between the two. Many current implementations of UNIX are more "mature" than the current implementation of NT, but the immaturity of NT is no doubt a transient phenomenon. The feature set of NT is certainly larger (and better) than that of "traditional" UNIX, but modern UNIX implementations support the same sorts of desirable features as does NT.

Is NT an open operating system? Unlike MS/DOS, NT is portable, both in theory and in fact. As of this writing, NT has working implementations on at least three different processors: Intel, MIPS, and Alpha. NT is certainly licensable—this is Microsoft's means for making money. There are no detailed published specifications for NT, but this also appears to be temporary. The only criterion for openness not met by NT is the open process—the design of NT is dictated strictly by Microsoft. However, just as a benevolent dictatorship may perhaps be the best form of government, this is not necessarily a bad thing.

2.3 Does the Operating System Matter?

If we agree that both UNIX and NT are acceptable operating systems, and if software vendors sell versions of their software to run on both operating systems, do software users care which operating system is used? Clearly the answer is no, it doesn't matter. One can use the operating system of one's choice and have no restrictions on the software packages that can run on the system.

There are two parts to the premise of the statement that begins the previous paragraph. That the operating systems are "acceptable" is a technical issue, that software vendors sell versions of their software to run on both operating systems is a marketing issue. The technical issue is relatively easy to decide. Some argue that the marketing issue is easy as well (i.e., that no one can compete with Microsoft). The point is that the answer to the question "Does the operating system matter?" appears now to be strictly a marketing issue.

3 Microkernels

Microkernels are a structuring technique for building operating systems. Rather than structure an OS as a single, monolithic piece of software, it is split into a *microkernel* and one or more *personalities*. The microkernel provides basic functionality, such as virtual memory, low-level I/O, and scheduling, while the personality uses the functionality of the microkernel to build the higher-level abstractions, such as processes and files, that are used by user applications. The code implementing the microkernel must run in privileged mode, i.e., it is in the "kernel." However, the code implementing the personality can run in user mode—it does nothing that would require it to run in privileged mode. The basic functionality of a microkernel can be used to build a variety of different personalities and thus provide a variety of different operating-system interfaces. Figure 1 shows a microkernel running in privileged mode and a personality and a number of user applications running in user mode. The personality and each of the user applications reside in different address spaces—thus the personality is isolated from user applications, just as it would be if it were in the kernel. The personality is a *client* of the microkernel—it obtains services from the microkernel. The user applications are clients of the personality—they obtain operating-system services from the personality (though the services of the microkernel might be used to provide communication between the applications and the personality).

Two microkernels that have achieved some degree of success are Chorus [7], from Chorus Systems, and Mach [4], from Carnegie Mellon University. Both have been used as the basis of UNIX implementations: Chorus with UNIX System V Release 4 [1] and Mach with Berkeley UNIX and OSF/1 [5]. There have been a few experiments in building personalities for other operating systems on top of Mach [6, 9]. IBM has demonstrated an OS/2 personality for Mach. Furthermore, the architecture of NT is based on microkernel concepts [3].

3.1 Why Use Microkernels?

Why is there interest in microkernels? We argue below that execution speed is not the reason—a microkernel-based implementation of an operating system may well be inherently slower than a traditional monolithic implementation. The interest is there because of the promise of *convenience*—using microkernels should make a number of things easier to do, including *debugging*, *supporting multiple operating systems,* and *distributing functionality.*

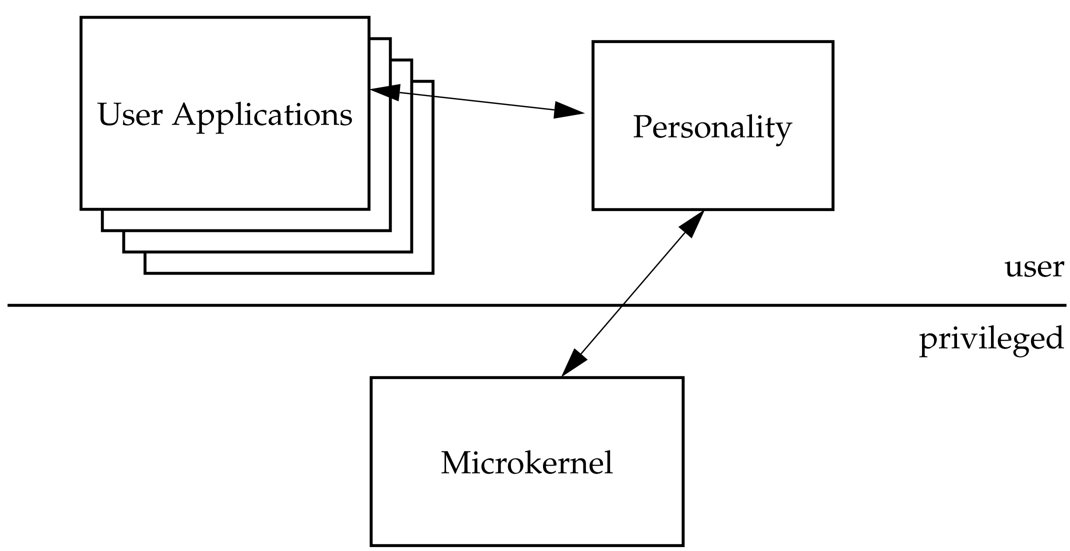

Figure 1: The Microkernel Approach

In the monolithic approach to operating system design, all components of the operating system reside in the kernel. This often makes debugging the operating system difficult, because most debugging tools are designed to debug user-level software. Kernel-level debugging tools are typically more primitive and cumbersome than user-level tools. In the microkernel approach, since the personality runs in user level, it can be debugged using the standard user-level tools. This is accomplished as shown in Figure 2: The personality being debugged, the “test personality,” is run in an environment set up by the “master personality.” The debugger, which gets its operating-system services from the master personality, has control over the test personality and all of its clients (i.e., user applications). Since the master personality has control over the test personality, it can pass this control on to the debugger, allowing the debugger to control (and monitor) the actions of the test personality and its clients.

Personality modules for different operating systems can be built on top of a single microkernel. Thus one can run two different personalities simultaneously on the same computer, for example run a DOS program while running UNIX, or debug one operating-system personality using the debugging tools of another.

What is more important is that a vendor can take advantage of the microkernel approach to ease its task of supporting multiple operating systems. Functionality common to a number of operating systems can be either put in the microkernel or made available as separate, user-level servers that can be used by the different personality modules. For example, device drivers and other machine-specific code can be shared by the various personalities—this reduces the amount of code that must be written and the number of modules that must be maintained. Any one computer might run just one personality, but some of the modules used by this personality can be used by other personalities as well.

Why is this sort of sharing not done among monolithic implementations of the operating systems, using their device drivers and other machine-specific modules? The problem is that such modules fit only into the systems for which they were designed. The internal interfaces within an operating system may be specified and well known, but they are peculiar to a particular operating

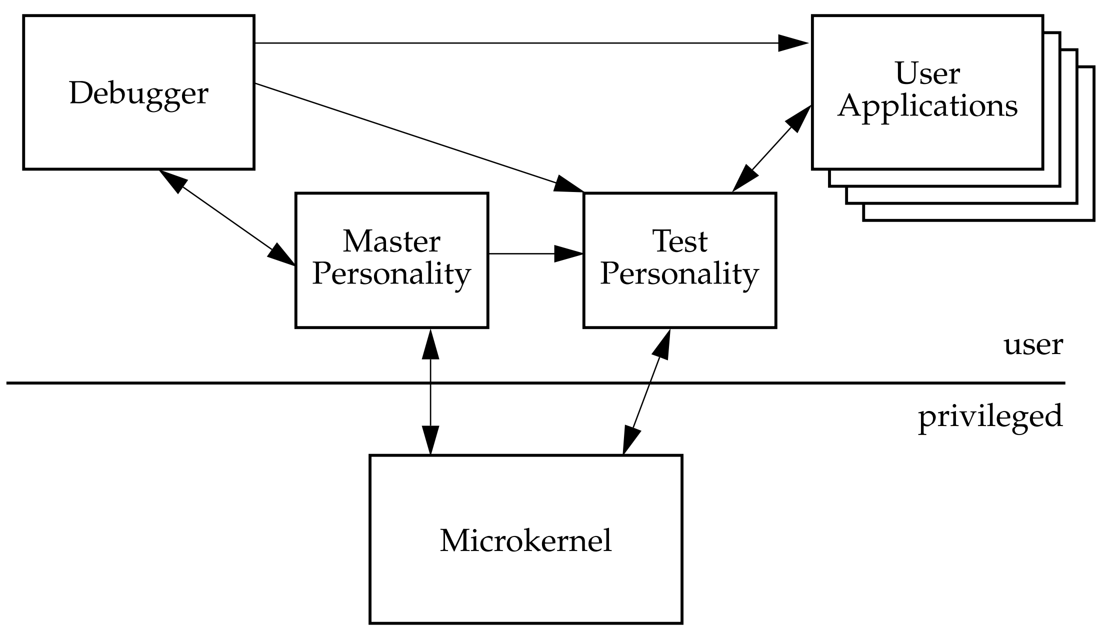

Figure 2: Debugging Personality Code

system. The modules use data structures and global variables that are defined only within their intended system. The advantage of the microkernel approach is that it forces a modular design approach that can be used to eliminate such dependencies. It is not that such a design approach cannot be used with monolithic kernels, it is that with microkernels this approach *must* be used. Furthermore, as discussed below, a lot of machinery is available in the microkernel itself to make the communication between modules, even if they reside in different address spaces, relatively efficient.

A final reason for the convenience of the microkernel approach is that it aids the development of multicomputer-based systems, i.e., collections of computers that do not share memory, but communicate via some sort of high-bandwidth communications interconnect. We would like such multicomputers to behave as much as possible as if they were a single computer system, running a single operating-system image. Even though we want to reap all the advantages of parallelism, we would like to avoid the difficulties of parallel programming.

The microkernel approach does not eliminate the difficulties of parallel programming, but it does give us a good start toward an implementation of an operating system for a multicomputer. How this is done is illustrated in Figure 3, which shows a multicomputer consisting of six interconnected computers. What was originally a single operating-system personality has been further subdivided along functional lines—here it is split into process management, file management, network management, database management, X/Window management, video management, and audio management components. The components reside in separate address spaces and communicate with one another using the facilities of the microkernel. If all components are on the same computer, then communication is straightforward. However, the microkernel's communication facilities allow individual components to be moved to other computers transparently—two components can communicate with each other without regard to location. Thus, in Figure 3, each component has been assigned to one or more computers, with no change to the logical structure of the composite operating system. User applications can run on any computer (or collection of computers) and have full, transparent access to all components of the operating-system personality.

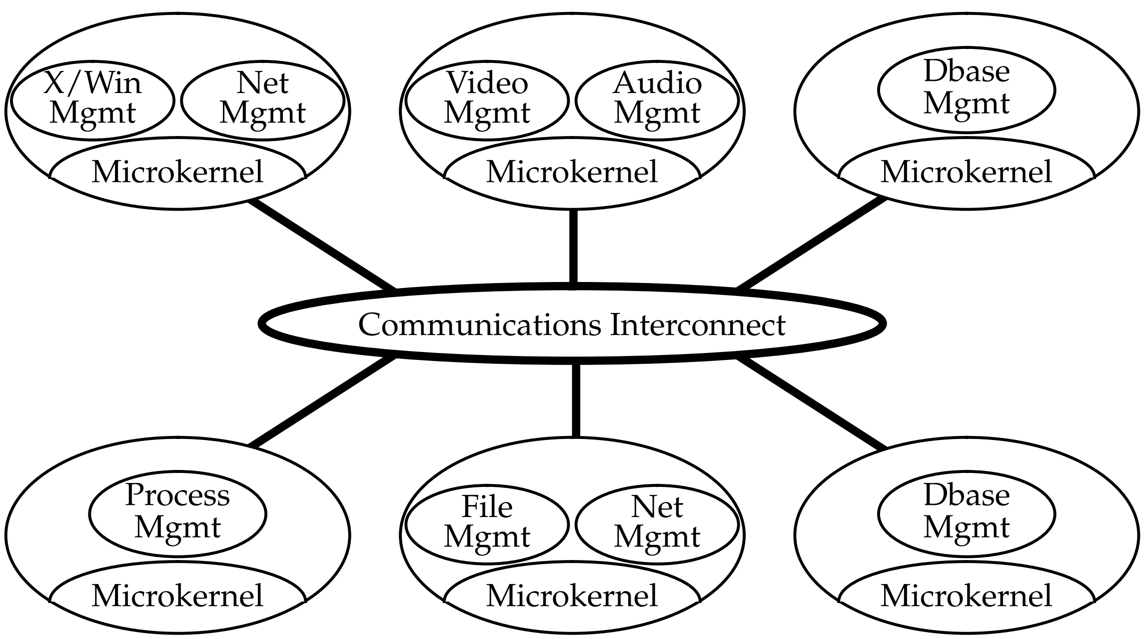

Figure 3: The Microkernel Approach on a Multicomputer

More problems must clearly be dealt with before we have a good multicomputer operating system, but many of the major pieces are in place. This approach has been used for the design of the OSF/1 AD operating system, which is used on the Intel Paragon and the Thinking Machines CM5.

3.2 How Microkernels Work

The basic concepts used by both Chorus and Mach are quite similar. We discuss Mach, but most of what we say also holds for Chorus.

Mach provides a small number of basic abstractions, of which the most important are listed below:

- *task*: an address space and a holder of capabilities.
- *thread*: the abstraction of a processor. Threads execute independently of one another; their execution is multiplexed on the available processors. Threads are restricted to execute within a task; there can be any number of threads within a task.
- *port*: a one-way communication channel. One task is the holder of *receive rights* for the port; it, and only it, can receive messages from the port. Any number of tasks have *send rights* for the port; they can send messages through the port.
- *message*: the unit of communication through ports. A message contains an arbitrary amount of data. Messages may include receive or send rights for ports. Thus the holder of the receive right for a port may pass it on to another task via a message. Similarly, a holder of a send right may give a copy of the send right to another task by including the right in a message sent through some other port.
- *memory object*: some "thing," e.g., a file, that is mapped into the address space of a task.

Ports are used not only as communication channels, but also as a means for identifying objects: An object is maintained inside of some task. A port is created in the task and the receive right for that port is associated with the object; send rights for the port are given to other tasks that as references to the object. A task sends a message through such an object-reference port, and this message is interpreted by the receiving task as pertaining to the associated object.

Tasks and memory objects form the pieces of a system. Threads are the active agents; ports and messages are the glue that holds things together. To make this a bit more precise, Figure 4 shows how UNIX might be implemented via a microkernel. The UNIX personality is implemented as a task with a number of threads executing inside of it. It maintains *process objects*, one per UNIX process (we think of these processes as being *clients* of the personality), and associates with each process object a port. These objects contain process-specific information such as the table of open files, signal-handling information, process ID, etc. The personality owns the receive rights for these ports and gives the send rights to the respective processes.

Each UNIX process is implemented as a task containing one or more threads. It is given the send rights to the port associated with the process's process object in the personality. Calls by the process to the operating system (i.e., system calls) are converted into messages sent through the port to the personality. (How are the responses to these calls handled? Ports are one-way communication channels, so the port used to send a request cannot be used to send the response. Instead, the requesting task provides send rights to a separate response port as part of the message. The receiving task uses these send rights to send its response back to the requestor.) Thus the port/message facility is the means for transferring a process's requests to the UNIX personality and for transferring the responses back to the process.

What remains to be seen is how each process's system calls are converted into message exchanges. The most straightforward way to accomplish this involves the system-call library, a library of procedures linked into each process that has one entry per system call. The body of each routine in the library performs the appropriate trap instruction to transfer control to the kernel where, in a monolithic implementation of UNIX, one would find the UNIX operating-system functionality. For the microkernel version, this library can be replaced with one in which the traps are replaced with code that invokes Mach's message-passing calls.

The only problem with this approach is that it does not work if it is not possible to replace the system-call library. This is the case if the library has been statically linked into the program executed by the process; this static linking was, until recently, the only form of linking in UNIX. Thus another technique is required if we are to support such statically linked programs.

Mach provides a system-call redirection facility in which the microkernel automatically redirects system calls back to user space. As used in OSF/1 Release 1.3, whenever a process issues a UNIX system-call trap, the microkernel converts the trap into a message and sends the message to the UNIX personality, via the process's port. Thus a program that was statically linked for a monolithic UNIX implementation can execute correctly without modification on the microkernel implementation.

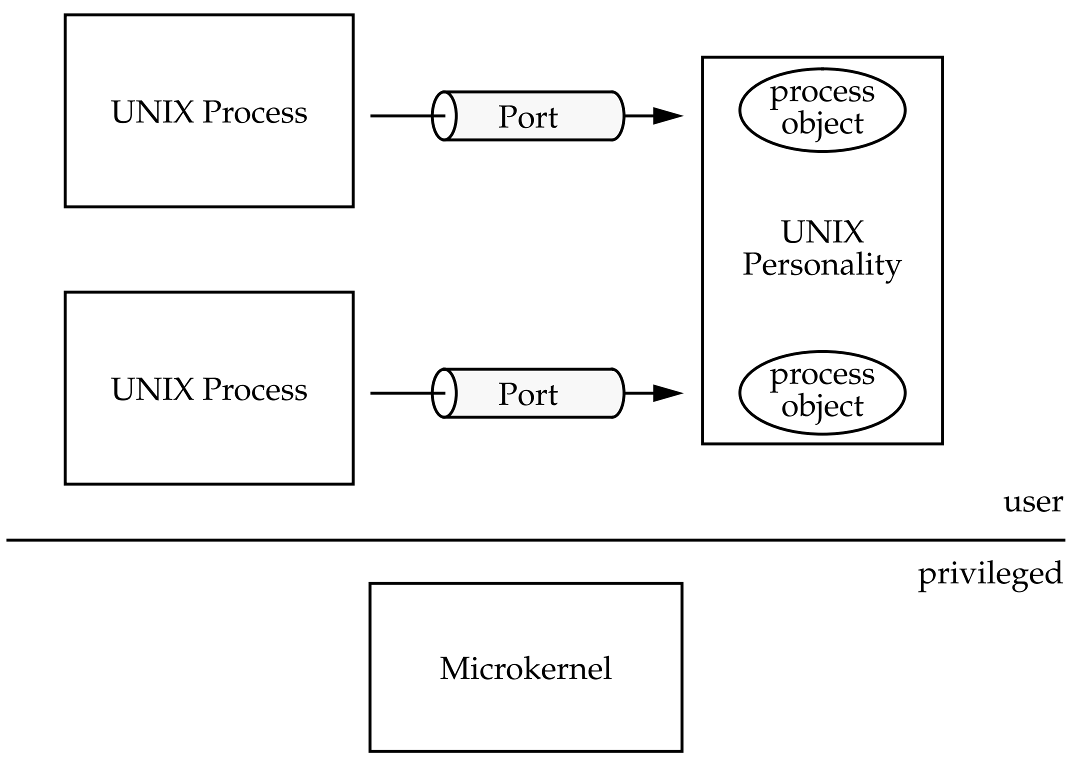

Figure 4: UNIX via the Microkernel Approach

3.2.1 Performance

There is nothing about the microkernel approach that makes microkernel-based implementations of operating systems any faster than monolithic implementations. In fact, all published studies comparing the two show that the microkernel-based implementations are always slower. Why this is so is not hard to see. With the personality code running in user mode, requests of user applications for operating-system services must first go into the kernel, then back to user space to the personality module. Furthermore, large amounts of data must be transferred among the kernel, the user application, and the personality.

A great deal of work has been done to optimize the performance of microkernels. The time to switch from user mode to kernel mode and back to user mode has been minimized and a variety of techniques are employed to reduce the costs of the data transfers. But monolithic operating systems are still faster. An approach pioneered by Chorus, and now used in Mach, is to develop and test the system with the personality running in user mode, but move the personality into the kernel (while still maintaining its modularity) for production purposes. This does speed things up, but even still the monolithic approach is faster.

This problem with performance is not, however, a serious blow to the microkernel approach. Few new operating systems are faster than their predecessors. Advances in operating-system technology makes programming easier, systems easier to maintain and easier to extend. In some cases speed is the most important criterion, but in many cases, if not most, convenience is what sells.

4 Multithreaded Programming

Multithreading is a technique for writing concurrent programs, i.e., programs in which many things are taking place at once. It is supported by most modern operating systems, including UNIX, NT, and OS/2.

A *thread of control* (or, simply, thread) is the abstraction of a processor; it is an *execution sequence*. A normal, sequential program has a single thread of control. Thus if we have two programs running independently but at the same time, we have two threads, each executing a separate program. *Concurrency* arises when a number (at least two) of threads are *in progress* at the same time; in contrast, *parallelism* arises when a number of threads are executing simultaneously.

A good example of an application that is easier to write as a multithreaded program than as a single-threaded one is a server handling queries of a simple database from a number of different clients (see Figure 5). In a single-threaded program, shown in the upper portion of the figure, the single thread either would have to serve one client at a time, resulting in excessive delays for clients with quick queries, or would explicitly need to multiplex the requests of the various clients. This latter approach would yield a complex program that would be difficult to debug and test.

On the other hand, the multithreaded program, shown in the lower portion of Figure 5, would be very simple. The programmer writes the code to deal with one client. Each client request is handled by a separate thread, but all such threads are executing the same code. There is no need to worry about multiplexing—the threads-support system takes care of that. The only thing different from a single-threaded program is that the programmer must deal with synchronizing the access to the database, but this is not difficult. Thus the resulting program does not delay clients with quick queries and is much simpler and easier to debug and test than the single-threaded version. A further advantage of the use of threads in this example is that, while one thread does I/O in response to its client's request, other threads can do other processing. Thus we get concurrency between I/O and computation.

Another advantage of multithreading is that it can be used to exploit a parallel computer. Figure 6 shows the multiplication of two matrices on a *shared-memory multiprocessor*. Rather than use some fancy parallel matrix-multiplication algorithm, we simply parallelize the simple approach: element *i,j* of the product matrix is computed directly as the inner product of *row i* of the multiplier matrix and *column j* of the multiplicand matrix. If the product matrix has *m* rows and *p* columns and if *m**p* processors are available, then each of the *m**p* threads can simply compute a single inner product. The threads-support system, in conjunction with the operating system, insures that each thread runs on a separate processor. If, for example, there are only *m* processors, then only *m* threads are created and each thread computes a row of the result. With only two processors, two threads are created and each thread computes half of the product.

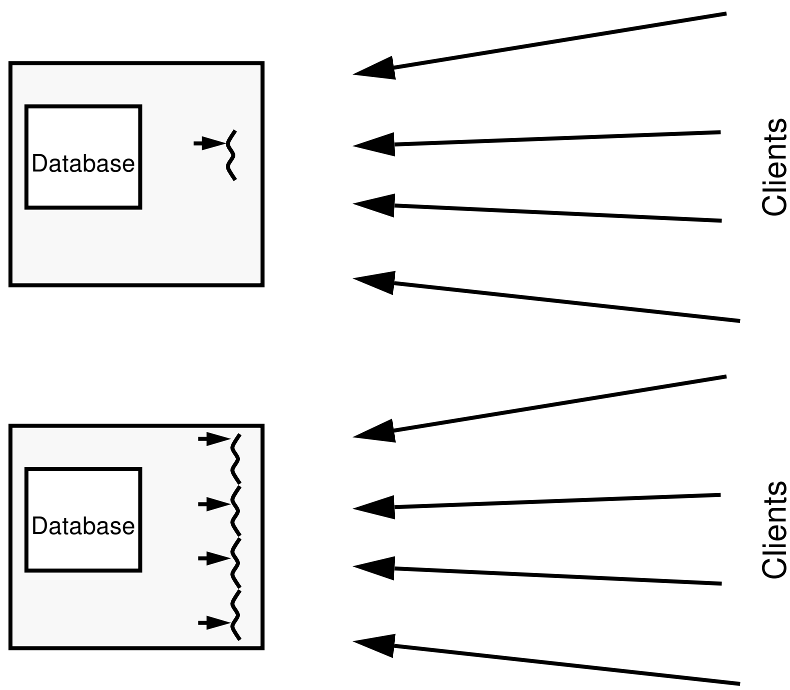

Figure 5: Two Approaches to the Design of a Database Server

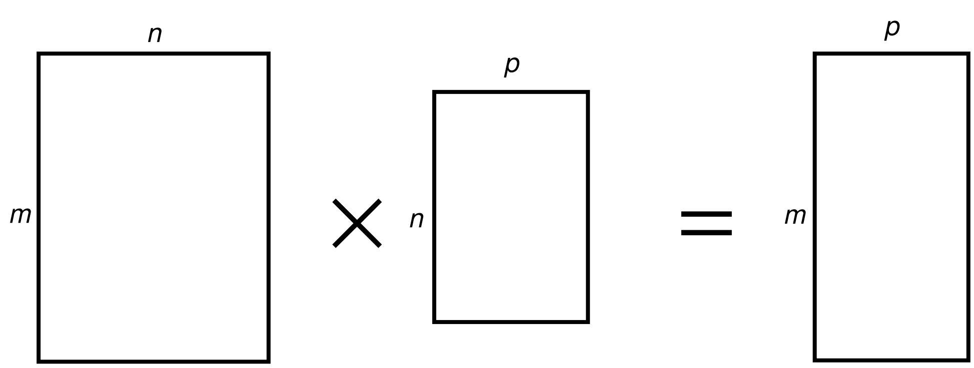

Figure 6: Multiplying Two Matrices

One might argue that traditional UNIX already supports the concept of threads—each process contains a (single) thread, so that by programming with multiple processes, one is programming with multiple threads. This is true, but misses the point of programming with threads. Creating a process is expensive; creating a thread within an existing process is cheap.

Consider Figure 7. The operating system represents a process with the help of some sort of control block, the *process control block (PCB)*. But a process is also an address space. Creating a process involves creating a new address space. To do this, the operating system must create some sort of memory map (the form of which depends upon the hardware). This new address space must be filled from somewhere. In UNIX, the address space is created as a result of the `fork` system call and is filled with a copy of the parent process. Of course, the parent's address space is not really copied into the child's; it is a highly optimized "logical copy." But, even so, making it is still a lot more expensive than doing nothing. If the `fork` is followed by an `exec` system call, then there is the additional expense of loading the new image from the file system (this too is optimized, but, again, is more expensive than doing nothing).

Associated with each processor is some sort of address-translation cache, commonly called a *translation lookaside buffer (TLB)*. This TLB contains address translations for the current address space. References to pages whose translations are not in the cache cause a cache miss and force a lookup of the translation in the memory map, which involves a number of extra memory cycles. If the processor switches from one address space to another, the previous contents of the TLB are now useless, and it must suffer a high proportion of cache misses until enough translations for the new address space are in the cache.

Communicating between processes involves transferring data from one address space to another, perhaps via a pipe. This typically requires that the data being transferred be copied, often twice. Some systems provide optimizations for even this, involving tricks with virtual memory, but, once again, doing them is still more expensive than doing nothing.

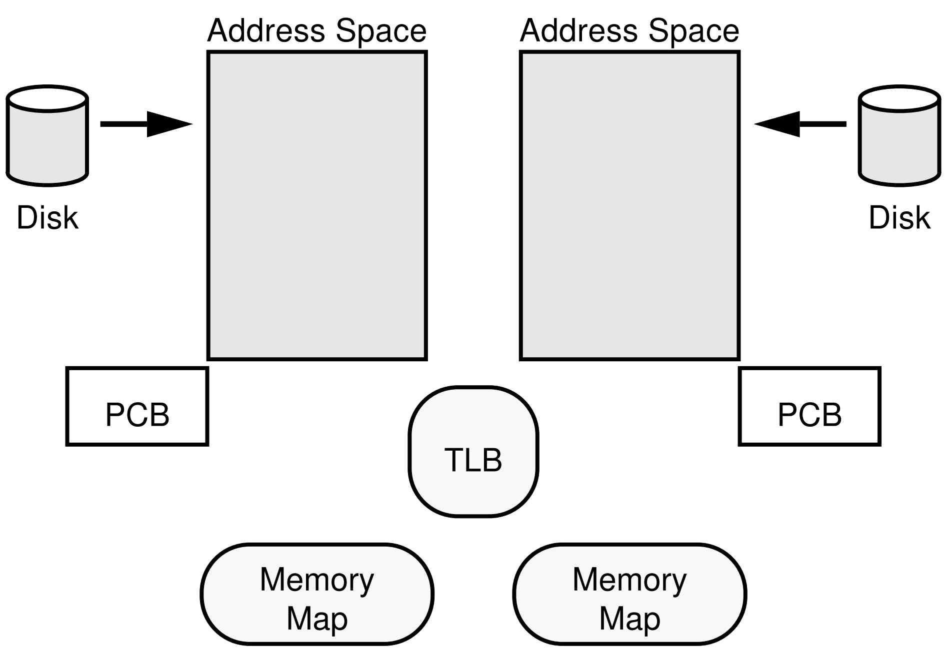

Figure 7: Architectural Support for Processes

How is a thread represented? As shown in Figure 8, a thread consists of a *thread control block* (*TCB*), a stack, and perhaps some operating system control state, a *kernel thread control block* (*KTCB*). Creating a thread is easy—the process already exists, so all we have to do is create a new TCB, allocate a new stack, and possibly create the KTCB. The time required to create a thread depends on the operating system and the hardware, but it is on the order of a thousand times less than the time to create a process.

Switching between threads does not involve switching between address spaces; thus we do not have the problem with refilling the TLB that arises in switching between processes.

Communicating between threads of one process is simple because the threads share everything—in particular, they share their address space. Thus data produced by one thread is immediately available to all the other threads.

One can get the benefits of multithreading in a number of ways. One approach is to build it into the programming language. Not very many languages were designed with multithreading in mind, but one that was is *Ada*, in which threads are known as *tasks*. Another approach is to augment an existing language by adding thread support as a new, grafted-on language feature. There are a number of *parallel Fortrans*, all based on this approach. The canonical parallel Fortran feature is the "parallel do-loop," in which multiple threads execute a do-loop in parallel.

A final approach is to supply multithreading capabilities via a subroutine library. (The subroutine library does not implement all the thread support, but the interface to the thread support is via the subroutine library.) We can use this library with C and, perhaps soon, with C++. (There is no conceptual or technical problem with using threads with the C++ language; the only issue is standardization—what interface to threads should the C++ programmer see?)

Not all applications can benefit from multithreading. A good example of one that cannot is a compute-bound application on a uniprocessor system: the use of threads would slow down such an application. But a number of types of applications do benefit.

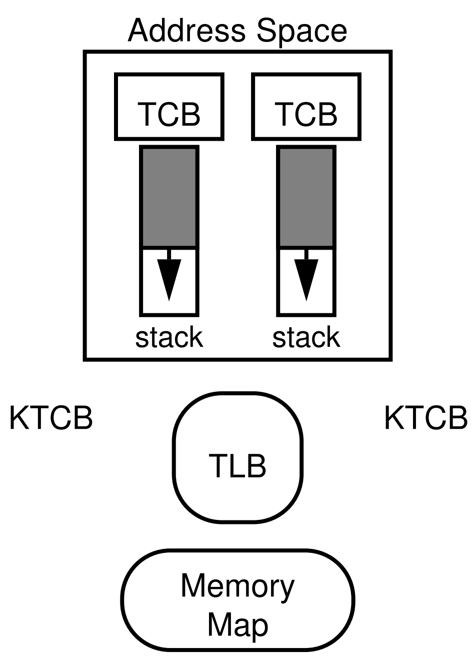

Figure 8: Architectural Support for Threads

As we have already seen, servers that handle multiple clients concurrently can benefit immensely from the use of threads. Such applications do not necessarily run faster with threads, but they are easier to write, debug, and test when written with threads.

Applications that are slowed down because of *synchronous waiting* can be speeded up with threads—one thread can wait while another thread continues with normal execution. This class of applications includes both *client* applications that use the services of some server and applications making synchronous I/O requests.

Finally, as we saw in the matrix multiplication example, applications involving parallelizable algorithms running on parallel processors can benefit from threads.

Consider an application that is making a request of a server. If there is something else to do in the application and the request will take a while to be handled, then it makes no sense to have the entire application wait for the response. With threads, one can simply create one thread to wait for the response while the original thread goes on to do other work.

Note that the server need not be on a different machine, or even in a different process. For example, consider a document-formatting system (such as the one used to produce this paper). At times typed (or moused) input is not responded to, because the program is busy doing something else, such as checkpointing the document. If this program had been written with the use of threads, then such waits would be unnecessary, since input processing and checkpointing could take place concurrently.

We have discussed how to use threads to exploit a shared-memory multiprocessor, but what about nonshared-memory multiprocessors? A common example of such a system is a collection of workstations. By using threads in conjunction with a remote procedure call (*RPC*) package, we can distribute our application relatively easily and treat the collection of workstations as a multiprocessor.

One thread might create a number of other *child threads*. Each of these children could then place a remote procedure call, invoking a procedure on another workstation. The original thread has merely created a number of threads that are now running in parallel; however, this parallelism involves other computers.

The history of multithreaded programming goes back to at least the ’60s. Its development on UNIX systems began in the mid-’80s. Perhaps surprisingly, there is fair agreement about the features necessary to support multithreaded programming. Even so, a number of different thread packages are available today, each with a different interface. However, for several years a group, known as POSIX 1003.4a, has been working on a standard for multithreaded programming. At the time of this writing, this standard is still in draft form, but it is widely believed that the final form will be agreed upon early in 1994. Once this happens, most vendors of systems supporting multithreaded programming will support the POSIX interface. This will have the extremely important benefit of allowing multithreaded programs to be portable, and thus bring them into the world of open software.

5 DCE

The *distributed computing environment* (*DCE*) is an enormous package of software from OSF (built from software provided by a number of its member companies). It provides support for a model of distributed computing known as *client/server computing*: certain computers, servers, provide services to other computers, clients. An important aspect of DCE is that it is operating-system independent (though its early implementations were all UNIX implementations). Thus it provides a computing model that may differ from that provided by the operating system.

The most important part of this model is the interaction between client and server. A server provides some sort of service, something as simple as printing or a bit more complicated such as database management. DCE provides the following features to support this sort of interaction:

- *remote procedure call* (*RPC*): this facility extends the notion of procedure calling across machine boundaries. One can place a call to a procedure, even though the code of the procedure and all of its data reside on a different computer. Thus the model of computing is that servers provide procedures that are called by clients.
- *naming and directory services*: servers can put information about themselves into a general directory system so that clients can obtain information about how to access the services provided. This includes support for a number of *name spaces* so that services can be referred to via convenient names (much like file names)
- *security*: one establishes an identity which is valid throughout the distributed environment. Clients and servers can *authenticate* each other; i.e., each can prove to the other that it really is who it claims to be. Furthermore, the client and server can protect the data be passed between them to prevent other parties from modifying it and, if necessary, from reading it. A standard means is provided for *authorization*: objects managed by servers can be tagged with *access control lists* (*ACL*s) indicating who is allowed to access the object and what they are allowed to do to it.

In addition, DCE provides the following three facilities:

- *distributed time services*: for a variety of reasons, it is important that the notion of time on all computers within a distributed environment be fairly consistent with one another.
- *distributed file system*: though the NFS distributed file system is already used widely, DCE includes another one, *DFS*, that is integrated with the rest of the DCE facilities. Also included is a local file system, known simply as *LFS*, which is also integrated with the other DCE facilities (including ACLs) and which allows quick recovery from crashes.
- *threads*: the use of threads is essential for many aspects of DCE. Since not all operating systems support multithreaded programming, included with DCE is a threads package that supports multithreaded programming using strictly user-mode facilities. It is not a replacement for a good, operating-system-supported threads package, but a stand-in so that DCE can be used while such a good threads package is being developed.

In the following paragraphs we first discuss RPC and how it makes use of some of the other pieces of DCE. We then say a bit more about these other pieces.

5.1 RPC

To understand remote procedure calls, we must first understand local procedure calls. Let's assume that the context of each procedure is stored on the stack in an area known as a *stack frame*. When a procedure is called, the first step is to prepare the current stack frame: the arguments of the call and the address of the next instruction (the return address) are pushed onto the stack. Next, a jump is made to the target procedure, which sets up its own stack frame: it pushes onto the stack the values of any important registers it will be modifying (i.e., it sets up the *register save area*) and allocates space on the stack for any local variables it uses. The target procedure can now find the arguments being passed to it, since they are at a known location at the end of the previous (i.e., the caller's) stack frame. When this procedure returns, it restores the caller's context (the saved registers), passes any return value back to it, and transfers control to the return address.

Now consider what happens if we try the same thing when the caller and the callee are on two different machines (or at least in two different address spaces). A naive way of handling this would be to make no changes at all to the code of the caller and the callee, so that the caller's stack frame is on the caller's machine and the callee's stack frame is on the callee's machine. Assume that somehow we arrange for the transfer of control between the caller and callee machines.

We have an immediate problem once we get to the callee procedure: it is running on one machine, but its arguments, inside the caller's stack frame, are on another. Furthermore, when the callee is about to return, it puts its return value in a register, but this register cannot be seen by the caller. We could provide special code inside the callee so that, whenever it attempts to access its arguments in the caller's stack frame, a request is sent to the caller for these arguments. However, among the problems with this approach is that the code produced for the remote version of the procedure would have to be different from that produced for the local version. A better approach would be to transfer the arguments to the remote (i.e. callee's) machine as part of the call, so that they are there when the callee needs them. Furthermore, we can do this so that the code for the remote version of the procedure is identical to the code for the local version.

The technique for doing this is to make use of additional code known as *stub procedures*. These come in pairs—the client-side (or caller-side) stub and the server-side (or callee-side) stub. The client-side stub procedure appears to the caller as if it were the actual callee procedure. The caller calls it, and eventually it returns with the desired result. However, what this stub actually does is to put together a message containing the arguments. It transmits this message through the communication medium to the callee's machine, where the message is received by the server-side stub. This stub pulls the arguments from the message and calls the callee procedure with these received arguments. Thus, from the point of view of the callee procedure, the server-side stub is playing the role of the caller procedure. When the callee returns, it returns to the server-side stub, which puts the value being returned into a message and transmits this message back to the client-side stub. This stub pulls the value out of the message and returns it to its caller, the original caller.

A number of details are omitted from the above description:

- *Binding*—where is the callee? I.e., to which machine should the RPC request be sent?
- *Stub generation*—how are the stubs created? How can this be automated?
- *Data transfer*—how are input parameters, output parameters, and return values transferred between machines that may have different ways of representing the various data types?
- *Identification*—how are procedures identified? E.g., how do the client and the server know that they are referring to the same thing?
- *Transport*—what is the transport? Is it reliable? If not, how do we cope with it?
- *Exceptions*—how are runtime problems dealt with?
- *Failures*—what happens if the caller, the server, or both crash?
- *Security*—who is the caller? Who is the server? Who is listening?

5.1.1 Binding

In the case of a local procedure call, one must somehow bind the name supplied by the caller to an instance of a procedure with that name, the callee. This is normally a fairly straightforward operation (at least as far as the application programmer is concerned)—one uses a linker that finds an appropriate routine in a library and arranges so that when a call is made to the callee procedure, control is transferred to the procedure obtained from the library.

This must be done in the remote case as well, but here there is the additional problem of identifying the computer to which the call is to be placed. A simple approach is that the programmer explicitly specifies the computer, perhaps as part of the call. More generally, it would be nice to leave this open, to be determined at runtime. Furthermore, this determination of the server machine should be done as transparently as possible to the programmer—ideally, the programmer should not need to be aware of the process at all.

For example, the purpose of the remote procedure call might be to retrieve information from a database (perhaps to read one's mail). Though normally this database might be expected to reside on a single, designated machine, the machine might be down and a backup machine used instead. Or the database might be replicated and any one of a number of different sites could be used.

While the selection of the server could be explicitly coded in the client program, we would like to treat this aspect of writing a distributed program as a separate issue, so as not to interfere with the logic of the program, which should be the same regardless of the machine to which the call is bound. A nondistributed program can be linked to different instances of the procedures it calls merely by supplying it with different libraries. In the same vein, we should be able to bind calls from a distributed program to different servers without having to change the code of the program itself.

In general, we have the name of a desired procedure and pass this name to a name server, which returns to us the identity of the server or servers we may use for the call. Such name servers are conceptually very simple: potential servers store information about themselves in the name server's database; prospective clients query the database, getting names of suitable servers. If there are a number of possible servers, then in most cases some simple technique, such as random selection, is used to determine which server is chosen, though more sophisticated techniques, such as attempting to choose the most lightly loaded server, might be used (though there is no support for this in the current implementation of DCE).

5.1.2 Stub Generation

Stubs make remote procedures look like local procedures: the client-side stub looks to the caller as if it were the callee; the server-side stub looks to the callee as if it were the caller. The stubs are responsible for moving the arguments and return value from one side to the other. Creation of the stubs thus involves determining what is being moved.

The big problem is making this determination automatically—we want to generate the stubs from some sort of description of the actual remote procedure. Aside from the complete source code, the only description of a remote procedure commonly available is its declaration (or *signature*). But, certainly in C programs, this declaration is not all that helpful. To produce the stubs, we need to know how much data is being sent as arguments to the procedure. If an argument is an integer (and is declared unambiguously in C as a *long*), then we would know that four bytes are being transferred[1]. But what if the argument is a *char* *? Clearly we do not want to pass just the pointer

1. Note, however, that even this is not always true. The DEC C compiler for the Alpha architecture uses *long* to mean an eight-byte integer.

to the remote procedure; we would want to pass what it points to as well. But how big is the item pointed to? It is probably not intended to refer to a single character. It might to refer to a null-terminated character string, or perhaps to an array of bytes of some particular, but specified, size. Sometimes *char* * is used by C programmers to refer to "generic storage," i.e., storage of unspecified size for which, for some reason, it is inconvenient to provide a proper type. In any event, it is, in general, not possible to determine from the procedure's declaration the size of the data that should be passed to the remote procedure.

Clearly, in some languages enough information is provided in procedure declarations to determine the size of data to be transferred. Unfortunately this is not true of C and C++. Rather than redesign these languages completely, what is done is to provide additional information, outside the language, that completely specifies the data being passed to and returned from a remote procedure. Various specification languages have been designed for this purpose, such as the MIG language (used in Mach), the RPC *Language* (used in Sun RPC), and the *Network Interface Definition Language* (NIDL, used in Apollo's Network Computing System; DCE uses an enhancement of NIDL known as *IDL* (*interface definition language*).

5.1.3 Data Transfer

Besides knowing *what* is to be transmitted between caller and callee, we also need to know the *form* in which data should be transmitted. There are two issues here: the order in which the various pieces of complex data items (such as arrays and structures) are transmitted, and the representation of the primitive data items, such as integers and floating-point numbers.

Two approaches have been used for dealing with these issues. One is to include with the data being transmitted an indication of what is being transmitted. The other is for the caller and callee to agree upon what is to be done ahead of time.

In the Xerox Courier protocol, the first issue (order of transmission) is handled by the first approach: a description of each argument is supplied as part of the call and the return. Most other protocols use the second approach: the sender transmits the data in an order expected by the receiver. This order is determined from the datatypes of the arguments and return value, using some standard scheme. For Sun RPC, this scheme is given by the *external data representation* protocol (XDR); in DCE, it is given by the *network data representation* protocol (NDR).

The representation issue has also been dealt with via both approaches. In Courier and Sun RPC, a single data representation is used by all parties. So, for example, if an integer is being transmitted, then it is always transmitted in *big-endian*[1] form. This avoids all confusion about the representation. However, if a machine's native representation for integers is *little-endian*, it has to convert. Thus if two little-endian machines are communicating with each other, they each must convert all outgoing data to big-endian and all incoming data to little-endian.

The alternative approach is used in DCE RPC: the sender transmits the data in its own format and supplies a tag indicating what that format is. If the receiver's format differs from the sender's, then the receiver must convert. But if the formats are the same, no conversion is necessary. This approach is practical because the number of data formats in use is not large.

1. The "big-endian" vs. "little-indian" distinction, an allusion to Swift's *Gulliver's Travels*, refers to whether the byte addressed by the address of an integer contains the most significant bits (big-endian) or the least significant bits (little-endian). This allusion was first made by D. Cohen in [2].

5.1.4 Identification

We ordinarily think of a remote procedure as being a service or part of a service that is being made available to potential clients. If you have developed such a service, how do you arrange so that your service can be unambiguously identified? Putting this another way, how does a caller determine that it has contacted the appropriate callee, and *vice versa*? Note that this is a different problem from that of locating a service. There may be many instances of a remote procedure (or, more generally, of a remote service, supplying multiple entry points), all at different sites. Each of these instances should have the same identity, however such an identity is specified.

What all approaches for specifying the identity boil down to is associating with each service a unique integer. In DCE RPC such integers are known as UUIDs and are generated automatically. In other RPC approaches, such as Sun RPC, there is no such generator; identifiers are either assigned by a central authority (e.g., Sun) or supplied by the programmer (with the programmer guaranteeing that the integer is really unique).

5.1.5 Transport

The transport protocol is responsible for the actual transmission of data between the caller and the callee. Some protocols, such as the *transmission control protocol* (TCP), provide *reliable* transmission of data. This means that, as long as communication is not hindered by line outages, etc., the protocol makes certain that what is transmitted by the sending application is received by the receiving application unmodified (i.e., no bytes are changed, added, or deleted, and all bytes are received in the order in which they were sent). Other protocols, such as the *user datagram protocol* (UDP), do not guarantee reliable transmission. Since such reliability is essential for the desired RPC semantics, the RPC implementation itself must provide it.

5.1.6 Exceptions

As with local procedure calls, something may go wrong while a remote procedure is being executed. The problem might be reported to the caller via a result parameter or the return value. However, a number of potential problems are typically handled via some sort of *exception mechanism*. For example, on UNIX systems, an arithmetic problem is reported to the program via a *signal*. If the signal is generated on a remote machine, something must be done to propagate it (or its equivalent) back to the calling machine. The intent is to make the behavior in response to exceptions for the remote case identical to what it would be in the local case.

Signals may or may not be the desired exception-handling mechanism. Even on UNIX systems, we might want an exception mechanism whose semantics differ from those of the signal mechanism.

5.1.7 Failures

Situations arise with remote procedure calls that do not arise with local procedure calls. In particular, with local procedure calls, if the host computer crashes, then we have lost everything and there is no thought of recovering—nothing remains that can perform any recovery actions. (While some state information may be left on nonvolatile storage, say on disk, that can used once the computer comes back to life, this is beyond the scope of an RPC mechanism.) In the remote case there are three independent types of failures: the remote computer might crash, the local computer might crash, or the communication path between the two machines might be lost. In addition, of course, the remote procedure may itself make remote procedure calls, complicating the possibilities for failures even further.

If the local computer crashes, the effect should be analogous to the case in which the only computer crashes during a local procedure call—all activity related to the call ceases. But for this to happen, some sort of notification must be sent to the callee telling it to abort. We could simply terminate the callee, but we might want enable the callee to quit what it is doing gracefully, perhaps continuing its execution until it gets to a convenient stopping point or calling a special cleanup routine.

If the remote computer crashes, we need to inform the caller, perhaps through an exception mechanism. What is relevant to the caller is how far the call progressed before the crash. Perhaps the crash occurred before the call even started. In this case, the caller may simply retry the call, either on the original remote computer when it restarts or on some other computer. On the other hand, the call may have started before the crash took place. The call may have even completed before the crash—the results were just about to be transmitted back to the caller. Unfortunately, it is not possible for the caller to determine when the crash took place. This information is known only to the callee, and the assumption is that everything was lost when the computer crashed. (Information may of course be stored in non-volatile storage that can be used to determine what happened, but such use of application-specific information is beyond the scope of a general RPC mechanism; it could be used, however, by the application itself.)

What to do about this uncertainty becomes an issue of semantics. What we would like is what is called *exactly-once* semantics—if the caller places a call, that call executes exactly once on the remote machine; if there is a crash, then the RPC implementation figures out whether the call should be repeated. As we have just argued, however, this is generally not possible. So we must weaken the semantics so that we have something that is possible to implement. One such weakened semantics is known as *at-least-once* semantics—the caller continually repeats the call until it gets a sure indication it was completed. Another semantics is known as *at-most-once* semantics—if the callee machine crashes, the call is aborted and the callee is notified. It is then up to the callee to determine what to do next.

At-least-once semantics has the advantage that the calling program is not complicated by considerations of failure. However, there is of course the possible disadvantage that the call may actually take place on the remote machine more than one time. In many situations this presents no problem, but in some situations it is most definitely a problem (consider a banking application in which the remote procedure transfers money from your account to mine). With at-most-once semantics, the caller is protected from unintended repeats of procedure calls, but it must provide the code that determines what to do in the event of a crash.

The third potential failure is the loss of the communication path while the caller and callee remain up. Neither the caller nor the callee can communicate with the other, so both consider the other to be down. As long as the caller does not repeat the call on another machine, then when it finally regains communication with the callee machine and repeats the call, there is (or may be) information on the callee machine indicating the state of the original call. Thus if the call had completed the first time, the callee should be able to (finally) transmit the results to the caller, while if the call had aborted, the callee can supply the appropriate notification to the caller.

5.1.8 Security

A final general concern is security. For the local case, security, at least in the interaction between the caller and the callee, is not an issue—the security identity of the executing thread of control is of course the same in both caller and callee, and we assume that no third party can interfere in any way. But the situation is very different in the remote case. If the remote procedure is performing an action that requires some sort of privilege, the remote machine must have a reliable means of determining the identity of the caller. Similarly, if the remote procedure is supposed to perform some important action, the callee must be assured that it has actually contacted the correct server and not some impostor. Furthermore, a third party (the "enemy") may be trying to learn sensitive information by eavesdropping, or may be attempting to interfere actively by modifying the data being transmitted.

Providing security is certainly possible but can be expensive, in terms of both the compute time required and the extra data that may have to be transmitted. The application writer should be allowed to make a tradeoff between expense and security through the provision of levels of security. Authentication, i.e., proving one's identity to another party, is relatively cheap to accomplish, but the other forms of security are not. Thus, if authentication is considered a sufficient level of security, it can be provided without the cost of the other levels.

5.2 Naming and Directory Services

As discussed above, we need some sort of support for a *name service*—a database that maps service names to collections of servers providing the service. In DCE, such a name service is provided as part of a more general *directory service* which establishes a hierarchical name space and associates a variety of information with each name. DCE actually provides two such services: *CDS* (*cell directory service*), handling the special needs of DCE itself, and *GDS* (*global directory service*), an implementation of the international X.500 standard.

For administrative and naming purposes, DCE is organized into large units known as *cells*. A cell is a collection of clients and servers and is intended to mirror organizational structure: a cell might be an entire company or university, or perhaps a division, site, or campus. Each cell is managed by a single administration that is responsible for cell-wide policies and security.

CDS provides a hierarchical naming scheme within each cell. A name represents a path through a tree, in much the same way as files are named in most systems. Any sort of entity can be named in this way, for example, services, servers, hosts, etc. CDS is implemented as a replicated, distributed database, providing complete service even if some of the sites implementing it are down.

The collection of all cells is a forest of naming trees. Individual cells can be named in a number of ways:

- via the Internet name service—known as the *domain name service* (*DNS*)
- via X.500
- via CDS (this feature will not be implemented until Release 1.1 of DCE, due out in mid- to late 1994)

Within a cell, the paths defined by the CDS naming tree can lead into other name spaces. For example, both DCE Security and DFS have their own private name spaces. A file within DFS is identified by supplying the name of the cell, the name of the DFS namespace within the cell, and the name of the file within the DFS namespace. This is much easier than it sounds, because the syntaxes of the various names are "blended" with one another. Thus such a file might be identified as /.../osf.org/fs/u/twd/file. In this name, "/.../osf.org" identifies a cell within the Internet name service, "/fs" identifies the root of the DFS namespace within the cell, and "/u/twd/file" identifies a file within the DFS namespace.

5.3 DCE Security

DCE security consists of three related services:

- An *accounts database* that maintains information about user accounts, etc. for use throughout the cell.
- An *authentication service* that provides the means for a client and a server mutually to authenticate each other and to insure that data is transmitted securely.
- An *authorization service* that manages the use of ACLs (access control lists), allowing servers to determine if clients should be allowed to perform the operations they have requested.

The authentication service is based upon MIT's Kerberos system [8]. This is a *private key* approach in which one or more trusted servers store a copy of each *principal's* private encryption key. (A principal is some entity, such as a user or a server, than can enter in security negotiations. The encryption key, at least for human principals (users), is derived from the password). Kerberos provides a means by which two principals can prove to each other that each knows its own encryption key. It goes on to provide to the pair of principals a new encryption key, known only to the two principals, that can be used for further protection, such as *data integrity* (assurance that transmitted data is not being surreptitiously modified), and *data privacy* (assurance that transmitted data is not being surreptitiously read).

The accounts database stores account information (including encryption keys) for all the principals in a cell, along with information describing protection groups. One can "log in" to DCE—this process involves both the authentication service and the accounts database: one establishes one's credentials, i.e., proves his or her identity, using information provided by the user and information obtained from the accounts database. At no time is the unencrypted version of one's password ever transmitted over the communications network.

When one contacts a server, i.e., places an RPC to it, the RPC package automatically invokes DCE security and includes with the request an copy of one's credentials, encrypted by the security service in the encryption key of the server. The RPC runtime package on the server then decrypts these credentials and thus learns who the client is. The server can then invoke the DCE authorization service (which is linked into the server application) to determine, via access control lists, whether the client should be allowed to perform the requested operation.

5.4 DFS

DFS, the DCE *distributed file system*, is based upon *AFS* (the *Andrew file system*) from Transarc. Like AFS and Sun's NFS, DFS supports the transparent access to files within a distributed environment. Like AFS, it takes advantage of local disk space on client computers to form a cache of recently accessed file data. For read-only and private files, this can dramatically reduce the use of the network and the load on the servers. Furthermore, unlike both AFS and NFS, the semantics of operations on files over DFS are nearly identical to the semantics of such operations on local file systems. DFS fully supports DCE ACLs for files, assuming that the underlying local file system also supports ACLs.

In addition, DFS supports a number of features that are not so visible to users. Read-only collections of files may be *replicated*, i.e., redundantly stored on a number of different servers. If the server currently being accessed by a client crashes, then the client's file accesses are transparently switched to another server. Techniques for automatic and semiautomatic replication are provided.

The DCE local file system (*LFS*) is a nondistributed file system that can be used in conjunction with DFS. It provides full ACL support for files and directories. It supports a fast duplication technique, known as *cloning*, in which a logical (but not physical) copy is made of a collection of files. This collection can then be safely copied, perhaps to a backup medium, while the original copy goes back to normal use.

LFS also is a *log-based* file system, so that little time is required to recover it after a crash. This is accomplished by maintaining a log of all changes to what is known as *meta-data*, i.e., data for use by the system only that describes the contents of a file system. If a crash occurs, rather than perform lengthy consistency checking as in standard UNIX file systems, the contents of the log are "played back," a procedure which takes a few seconds rather than many minutes.

5.5 Whither DCE?

DCE contains an enormous amount of code—it is many times the size of the kernel of many modern operating systems, even monolithic implementations. Its first releases have been what are euphemistically called "functional" releases—meaning that they demonstrate the functionality of the product but are neither fast enough nor reliable enough to be used in production. Will it ever become mature enough to be used in production?

A number of companies are betting that it will be. It is a required piece of *COSE*, the Common Operating System Environment—a coalition of UNIX vendors formed in response to Microsoft's NT. Even Microsoft appears to be giving at least portions of DCE its de facto support—for example, the RPC package used by NT is a extension of DCE RPC and is interoperable with it (as long as no Microsoft extensions are used).

OSF has released its fourth minor release of DCE (1.0.3) and is scheduled to release the next major release, 1.1, in mid- to late 1994. Whether this release will be commercially successful remains to be seen. If so, it will represent a major step towards making open software in a distributed environment a reality.

6 References

[1] N. Batlivala et al., *Experience with SVR4 Over Chorus*, in *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April '92.

[2] D. Cohen, *On Holy Wars and a Plea for Peace*, *Computer* magazine, IEEE, October '81.

[3] H. Custer, *Inside Window NT*, Microsoft Press, '93.

[4] R. Draves et al., *Microkernel Operating System Architecture and Mach*, in *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April '92.

[5] Open Software Foundation, *Design of the OSF/1 Operating System*, Prentice Hall, '93.

[6] R. Rashid et al., *DOS as a Mach 3.0 Application,* in *Proceedings of the Usenix Mach Symposium,* November '91.

[7] M. Rozier et al., *Overview of the Chorus Distributed Operating System*, in *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April '92.

[8] J. Steiner et al., *Kerberos: An Authentication Service for Open Network Systems*, in *Proceedings of the Winter Usenix Conference*, February '88.

[9] C. Wiecek, *A Model and Prototype of VMS Using the Mach 3.0 Kernel*, in *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April '92.