A Comparison of Two Distributed Systems: Amoeba and Sprite

[To appear in Computing Systems.]

Fred Douglis*
douglis@mitl.com
Matsushita Information Technology Laboratory
182 Nassau Street
Princeton, NJ 08542 USA

M. Frans Kaashoek
kaashoek@cs.vu.nl
Dept. of Math and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

John K. Ousterhout
ouster@sprite.berkeley.edu
Computer Science Division (EECS)
University of California
Berkeley, CA 94720 USA

Andrew S. Tanenbaum
ast@cs.vu.nl
Dept. of Math and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Abstract

This paper compares two distributed operating systems, Amoeba and Sprite. Although the systems share many goals, they diverged on two philosophical grounds: whether to emphasize a distributed computing model or traditional UNIX-style† applications, and whether to use a workstation-centered model of computation or a combination of terminals and a shared processor pool. Many of the most prominent features of the systems (both positive and negative) follow from the philosophical differences. For example, Amoeba provides a high-performance user-level IPC mechanism, while Sprite's RPC mechanism is only available for kernel use; Sprite's file access performance benefits from client-level caching, while Amoeba caches files only on servers; and Sprite uses a process migration model to share compute power, while Amoeba uses a centralized server to allocate processors and distribute load automatically.

1 Introduction

The shift from time-sharing computers to collections of processors connected by a local-area network has motivated the development of numerous distributed operating systems [Abrossimov et al. 1989; Cheriton 1988; Mullender et al. 1990; Ousterhout et al. 1988].

*This work was supported in part by the Netherlands Organization for Scientific Research (N.W.O.) under grant NF 62–334.

This paper compares two distributed systems, Amoeba [Mullender et al. 1990; Tanenbaum et al. 1990] and Sprite [Nelson et al. 1988; Ousterhout et al. 1988], which have taken two substantially different approaches to building distributed systems. These approaches have developed as a result of different philosophies about the role of distributed systems and the allocation of resources within them. By comparing these two systems in the context of our experiences with them, we draw conclusions about operating system organization that may aid the design of future distributed systems.

We have chosen to compare Amoeba and Sprite for three reasons. First, they take different approaches toward user applications in a distributed system. Sprite is primarily intended to run UNIX applications on a network of workstations, and it hides the distribution of the system behind a shared file system. It distributes the operating system but does not provide special support for distributed applications. Amoeba is intended as a testbed for distributed and parallel applications, as well as traditional applications. It provides a high-performance mechanism for user-to-user remote procedure calls (RPCs) [Birrell & Nelson 1984], as well as a language to support parallel programming, so applications can easily take advantage of multiple processors. At the same time, it hides the physical distribution of the system, and processes cannot even determine where they physically execute. Second, Amoeba and Sprite allocate processing resources in substantially different fashions. Amoeba users share a single "processor pool," while Sprite associates users with individual workstations. Third, we have personal experience with both systems over the course of several years. We know a good deal about the historical development of the systems and have personal knowledge of both their strengths and weaknesses. We also have access to both systems and are able to compare their performance on identical hardware.

Naturally, there are many distributed systems besides Amoeba and Sprite. It would be possible to compare several contemporary distributed systems in a survey fashion, much as Tanenbaum and van Renesse did in 1985 [Tanenbaum & van Renesse 1985]. However, with the exception of Section 4 below, we have chosen to restrict our comparison to two systems. We believe that limiting the scope of this paper permits us to consider issues in greater detail than would otherwise be possible.

The rest of this paper is organized as follows. Section 2 elaborates on the fundamental design philosophies behind the two systems. Section 3 relates these philosophies to several operating system issues: kernel architectures, communication, file systems, and process management. Section 4 discusses how these issues have been addressed by other systems. Section 5 briefly reviews the development history of Amoeba and Sprite and describes their current research directions. Finally, Section 6 draws several conclusions.

2 Design Philosophies

The Amoeba and Sprite projects began with many similar goals. Both projects recognized the trend towards large numbers of powerful but inexpensive processors connected by high-speed networks, and both projects set out to build operating systems that would enhance the power and usability of such configurations. Both design teams focussed on two key issues: shared storage and shared processing power. The first issue was how to implement a distributed file system that would allow secondary storage to be shared among all the processors without degrading performance or forcing users to worry about the distributed nature of the file system. The second issue was how to allow collections of processors to be harnessed by individual users, so that applications could benefit from the large number of available machines.

However, in spite of their similarities, the Amoeba and Sprite projects diverged on two philosophical grounds. The first philosophical difference is the expected computing model. The Amoeba designers predicted that networked systems would soon have many more processors than users, and they envisioned that future software would be designed to take advantage of massive parallelism. One of the key goals of the Amoeba project was to develop new operating system facilities that would support parallel and distributed computations, in addition to traditional applications, on a network with hundreds of processors. In contrast, Sprite assumed a more traditional model of computation, along the lines of typical UNIX applications. The goal of the Sprite designers was to develop new technologies for implementing UNIX-like facilities (particularly file systems) on networked workstations, and they assumed that the distributed nature of the system would not generally be visible outside the kernel.

The second philosophical difference is the way that processes are associated with processors. Sprite again took a more traditional approach, where each user has a (mostly private) workstation and the user's processes are normally executed on that workstation. Although active users are guaranteed exclusive access to their workstations, Sprite provides a process migration mechanism that applications can use to offload work to idle machines all around the network. In contrast, Amoeba assumed that computing power would be shared equally by all users. Users would not have personal processors; instead, computing resources would be concentrated in a processor pool containing a very large number of processors. Thus processing power is managed in a much more centralized fashion in Amoeba than in Sprite.

2.1 Application Environment

Amoeba and Sprite differ greatly in the applications they are intended to run and the resulting execution environment they provide. Amoeba provides an object-based distributed system, while Sprite runs a network operating system that is oriented around a shared file system.

In Amoeba, each entity such as a process or file is an object, and each object is identified by a *capability* [Dennis & Horn 1966]. The capability includes a *port*, which is a logical address that has no connection to the physical address of the server managing the object. Thus, the location of the server is hidden from any objects that interact with it.

In addition to providing a uniform communication model, Amoeba eases the task of writing distributed applications. It provides automatic stub generation for remote procedure calls from a procedural interface declaration [van Rossum 1989]. It also supplies a programming language, called Orca, that simplifies writing parallel applications on a distributed system [Bal et al. 1990].

By comparison, Sprite is intended to ease the transition from UNIX time-sharing systems to networked workstations. Since most of the applications running on Sprite are such things as compilations, editing, and text formatting, the design of Sprite has emphasized location-transparent file access, consistent access to shared files, and high file system performance. In particular, Sprite caches file data on client workstations in order to perform many file operations without the need for network transfers [Nelson et al. 1988]. On the other hand, because applications on UNIX typically performed little or no interprocess communication (other than pipes), little effort was made to support special protocols for communication over the network at user-level. Instead, the file system provides a simple but relatively inefficient method for location-transparent user-level IPC when it is needed.

The decision to model a new system after an existing one has both positive and negative consequences. On the positive side, compatibility with UNIX has helped Sprite to develop quickly into a system that many people use for all their day-to-day computing. In particular, most UNIX applications can be run on Sprite by recompiling. On the negative side, UNIX compatibility has restricted Sprite's application domain, and it has complicated several aspects of the system (such as process migration, described below). Compatibility with UNIX was less of a goal for Amoeba; because Amoeba is only partially compatible with UNIX, it is more difficult to port existing software to it. However, it offers more flexibility in the design of new software and more opportunities to do research on distributed and parallel languages and applications.

2.2 Processor Allocation

Allocation of processors in a distributed system ranges from a pure "workstation" model, in which each user executes tasks on exactly one machine, to a pure "processor pool" model, in which all users have equal access to all processors. The workstation model makes each host essentially autonomous; for example, each host maintains its own list of processes, which may typically be viewed only from that host. To execute commands on another host, a user must normally perform an explicit remote login. With the processor pool approach, the system is more integrated. Processors are dynamically allocated to processes regardless of the location of the user running them, and users may view the state of their processes anywhere in the system. Amoeba and Sprite implement two system architectures that fall between these two extremes. Amoeba's architecture is closer to the processor pool approach, while Sprite's is closer to the workstation model.

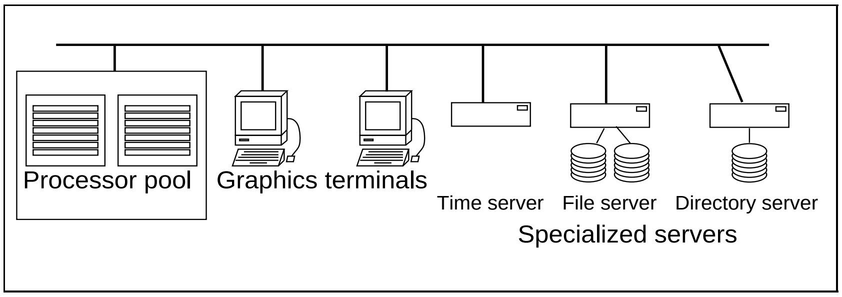

Figure 1: An Amoeba system consists of a processor pool, specialized servers, and graphics terminals.

Amoeba's system architecture is organized around a centralized processor pool, as shown in Figure 1. Each "pool processor" has a network interface and RAM associated with it, and these processors are dynamically allocated to processes as they are needed. However, unlike a system with a "pure" processor pool model, Amoeba also use processors outside the processor pool for system services. For example, the file server and directory server both run on dedicated processors. This separation avoids contention between user processes and system functions. Finally, users interact with the system using a graphics terminal, such as an "X-terminal." The terminal is essentially a cheap dedicated processor, a bit-mapped display, and a network interface. Only a display server runs on the graphics terminal; all other applications run in the processor pool.

The designers of Amoeba chose the processor pool model for three reasons. First, as we have indicated, they assumed that as processor and memory chips continue to decrease in price, the number of processors in future systems would greatly outnumber the users. In their opinion, it would be easier to place hundreds of processors in racks in a machine room than to distribute those processors equally among each user, and the addition of a new processor would benefit all users equally. Second, they assumed that the cost of adding a new pool processor would be substantially less than the cost of adding a workstation, since a pool processor would require only a processor, memory, and a network interface; a fixed amount of capital could make a larger increase in computing resources under the processor pool model. Third, they wanted to make the entire distributed system appear as a single time-sharing system. Users not only should not be concerned with the physical distribution of the hardware, they should not be aware of it at all.

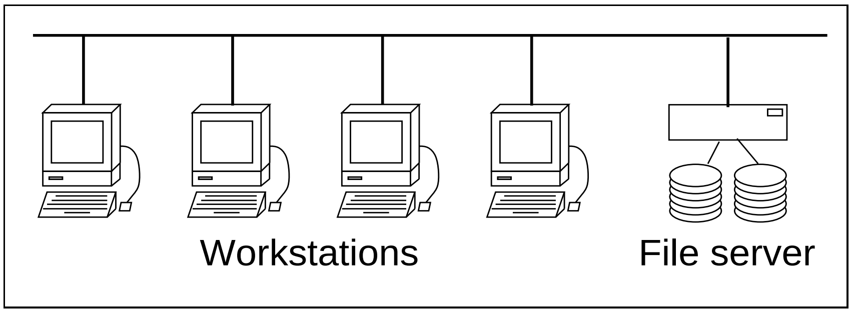

Figure 2: A Sprite system consists of workstations and file servers.

Sprite's processing power is distributed among a collection of personal workstations, as shown in Figure 2, but it does not implement a "pure" workstation model. Each user has priority over one workstation, is guaranteed the full processing power of that workstation, and executes commands on that workstation by default. However, Sprite also provides a facility to execute commands using the processing power of idle hosts. These commands appear to the user to run on the user's own workstation. In keeping with the workstation model, Sprite recognizes the preeminence of workstation owners on their own machines by migrating "foreign" processes away from a workstation if its owner returns.

In addition to workstations, Sprite provides dedicated file servers that are not normally used for application programs. It is also possible to add processing resources to the system without associating them with individual users. For example, a rack of processors could be used as a shared compute server, offering the same cost advantages as an Amoeba processor pool.

The designers of Sprite chose a workstation-based model for three reasons. First, they believed that workstations offered the opportunity to isolate system load, so that one user would not suffer a degradation in performance due to a high load on the system from another user. Second, they hypothesized that much of the power of newer and faster machines would be used to provide better user interfaces. The best way to use this power would be to put it as close to the display as possible; *i.e.*, in a workstation. Third, to the designers of Sprite, there appeared to be no difference between a graphics terminal and a diskless workstation except for more memory on the workstation; why not perform all computation on the workstations, rather than just interactive tasks?

3 Design Consequences

The decision of whether to organize processing resources into a shared pool or individual workstations has affected the design of Amoeba and Sprite in several ways. For example, Amoeba assigns processes to the most desirable processor in the system, achieving some dynamic load balancing. It does not implement client file caching, because the effectiveness of caching is decreased when the process that reads a new file is not likely to execute on the processor where the file was just written. Sprite caches files on workstations, and it implements process migration to preserve response time on workstations.

In this section, we discuss how the design philosophies described above affected operating system issues such as kernel architectures, interprocess communication, file systems, and process management. Amoeba and Sprite have made different sets of tradeoffs and differ both in the functionality they provide and the performance of many operations. While the design philosophies have affected both of these areas, in some cases performance has been affected by low-level implementation details as well. We evaluate both functionality and performance,[1]

[1]Measurements in this paper were taken on 8-Mbyte Sun 3/60 workstations (20 MHz Motorola 68020 processors, or about 3 MIPS), using Lance Ethernet controllers on a 10 megabits/second Ethernet. The file server for both systems used a SCSI-3 controller and a Wren IV SCSI disk.

distinguishing between the effects of design and implementation on performance when appropriate.

3.1 Kernel Architectures

One of the greatest differences between Amoeba and Sprite is their basic kernel architectures. Sprite follows the traditional UNIX monolithic model, with all of the kernel's functionality implemented in a single privileged address space. Processes access most services by trapping into the kernel, and each kernel provides services for those processes running on its host. The only shared kernel-level service provided by Sprite is the file system. In contrast, Amoeba implements a "microkernel," with a minimal set of services (most importantly, communication and low-level process management) implemented within the kernel. Other services, such as the file system and process placement, are provided by separate processes that may be accessed directly from anywhere in the system. As a result, some services that would be provided independently on each Sprite workstation (such as the time-of-day clock) may be provided in Amoeba by a single network-wide server.

There were two principal reasons for the decision to use a monolithic kernel in Sprite. First, the performance implications of microkernels were unclear at the time (even today they are still somewhat controversial). Communicating with user-level processes is more expensive than just trapping into the kernel, since hardware registers (such as the virtual memory context) typically must be modified. Thus, although it is possible to minimize the overhead of changing protection domains [Bershad et al. 1989], there are still additional costs associated with user-level services relative to kernel-level services. Second, placing all the kernel facilities together in a single address space made it possible for them to work together and share data structures. For example, the file cache and virtual memory system work together to share the physical memory of a machine [Nelson et al. 1988], and the process migration mechanism has a close relationship with all the major parts of the system. Although such close cooperation could also have been achieved in the microkernel model, shared memory would have been precluded and additional context switches would have been incurred on each cross-module invocation.

Amoeba's microkernel approach was motivated by uniformity, modularity, and extensibility. Since services are obtained through RPC, both kernel-level and user-level services may be accessed through a uniform, location-transparent interface. Users may extend or replace standard services with their own by using different capabilities. Finally, separate services permit the functionality of the system to be distributed and replicated on multiple processors to gain performance and fault tolerance.

In light of the advantages of the microkernel approach, one may ask whether any potential overhead from separate server processes is significant enough to detract from their design. A comparison between the performance of Amoeba and Sprite offers the opportunity to answer this question, especially since Sprite's performance during system calls and context switching is similar to several commercial UNIX-based systems [Ousterhout 1990].

As one might expect, performance differences between Amoeba's microkernel and Sprite's monolithic kernel depend on service access patterns. Since a kernel call is inherently faster than a remote procedure call, obtaining a simple service from a different process can be substantially slower than obtaining it from the kernel. For example, the minimum cost of a kernel call in Sprite on a Sun 3/60 workstation is about 70 microseconds, while the minimum cost of an RPC between two distinct processes on an Amoeba processor is 500 microseconds. Furthermore, a service may be provided by each kernel in Sprite but by a single global server in Amoeba. Accessing a service over the Ethernet in Amoeba takes at least 1200 microseconds.

However, the overall performance of the system depends on many factors. For example, Amoeba's lack of swapping or paging improves performance considerably: as we describe below, process creation and context switching are both generally faster in Amoeba than in Sprite. Overall performance is more likely to be affected by system characteristics such as the speed of communications and the use of file caching than by the choice between a microkernel or monolithic kernel. If a microkernel could be tolerably efficient for trivial operations and at least as good as a monolithic kernel for more complicated operations, the advantages of the microkernel approach—most importantly, modularity and extensibility— would appear to outweigh any potential disadvantages in performance.

3.2 Communication Mechanisms

Both Amoeba and Sprite implement communication mechanisms to enable processes to communicate with each other and to hide machine boundaries. Their mechanisms for doing so, however, are different. Amoeba presents the whole system as a collection of objects, on each of which a set of operations can be performed using RPC. Like Amoeba, Sprite uses RPC for kernel-to-kernel communication. Sprite has not really addressed the problems of building distributed applications, but it does provide a mechanism that can be used to support some kinds of client-server communication.

Considering kernel communication in isolation, Amoeba and Sprite have more in common than not. Both use RPC to communicate between kernels on different machines. The implementations vary in minor ways. Sprite uses the implicit acknowledgements of the Birrell-Nelson design [Birrell & Nelson 1984] to avoid extra network messages when the same parties communicate repeatedly. On the other hand, Amoeba sends an explicit acknowledgement for the server's reply to make it possible for the server to free its state associated with the RPC. This simplifies the implementation of the RPC protocol but requires an additional packet to be built and delivered to the network. Despite this extra packet, Amoeba obtains lower latency for the null RPC (passing no data): it takes 1.1 msec to perform a null RPC in Amoeba between kernels on two Sun 3/60 workstations, compared to 1.9 msec in Sprite. The difference is largely due to the necessity to perform a context-switch in Sprite when an RPC is received. For large RPCs, Sprite uses a blast protocol to send many packets without individual acknowledgments. This compensates for the other overhead in the RPC system, resulting in a slightly higher maximum kernel-to-kernel bandwidth: 820 Kbytes/sec in Sprite compared to 814 Kbytes/sec in Amoeba. Table 1(a) summarizes the performance of kernel-to-kernel RPC in each system.

Size (Bytes)	**Kernel-level Latency** (msec)	
	Amoeba	**Sprite**
0	1.1	1.9
16384	20.0	19.5
30000	36.0	——

(a)

Size (Bytes)	**User-level Latency** (msec)	
	Amoeba	**Sprite**
0	1.2	7.9
16384	21.0	33.5
30000	36.0	62.8

(b)

Table 1: Communication latency in Amoeba and Sprite. Measurements were taken for transfer units of 0 bytes, 16 Kbytes (the largest transfer permitted for kernel-to-kernel RPC in Sprite), and 30000 bytes (the largest transfer permitted during a single RPC in Amoeba). Part (a) shows kernel-to-kernel RPC performance. Amoeba provides appreciably lower latency for small RPCs but Sprite provides better performance at its largest transfer unit. The difference in the performance of large transfers arises because individual fragments in Sprite are not acknowledged. Part (b) shows the performance of user-level IPC. Amoeba's remote procedure calls are substantially faster than Sprite pseudo-device operations for all data sizes. Measurements were made on two Sun 3/60 workstations connected by a 10-Mbit Ethernet.

User-level communication, however, differs greatly between the two systems. Amoeba uses the same model for user-level as for kernel-level communications, with marginal overhead over the kernel case. Communication in Sprite is integrated into the file system name space using "pseudo-devices," which permit synchronous and asynchronous communication between user processes using the file system *read*, *write*, and I/O control kernel calls [Welch & Ousterhout 1988]. User-level communication in Sprite is more expensive than in Amoeba for four reasons: first, Sprite's user-level communication is layered on a kernel-to-kernel RPC that is significantly slower than Amoeba's for small transfers and about the same performance for large transfers; second, as a result of this layering, the Sprite calls involve additional locking and copying that Amoeba avoids; third, all buffers in Amoeba are contiguous and resident in physical memory, so no per-page checks need be performed; and fourth, Amoeba performs context switching much faster than Sprite (see Section 3.4). Thus, these differences in performance arise from both low-level implementation differences, such as contiguous buffers and context-switching speeds, and the higher-level philosophical differences that led to Sprite's layered approach. Table 1(b) demonstrates how Amoeba consistently outperforms Sprite at user level.

3.3 File System

Both Amoeba and Sprite provide a single globally shared, location-transparent file system. In either system a user can access any file system object from any location without being aware of the location of the object. The design of Sprite's file system was strongly influenced by Sprite's workstation environment and file-intensive applications. In particular, it caches data on both clients and servers to achieve high performance, and it adjusts the size of the file cache in response to demands for physical memory. Distributed applications on Amoeba are not necessarily file-intensive, and each new process is typically placed on a different processor, so client caching was not as important in Amoeba as in Sprite. Instead, Amoeba has emphasized the transparency and fault-tolerance necessary for a large distributed system.

Sprite provides a traditional UNIX open-close-read-write interface, with naming and file access performed in the kernel [Welch 1990]. Processes perform kernel calls to open files and obtain tokens they may use to perform further operations on the files. The kernel of the host running a process, known as the *client*, identifies the server for a file using an associative table based on the leading characters of the file's name. The client passes the file's full path name to the server, where name lookup and protection checking occur. The kernel of the file server returns either a *handle* that may be used to perform I/O on the file, a new path name to open (in the case of symbolic links), or an error condition. Once the client has obtained a handle for a file, it performs I/O operations by passing the handle to the server named in the handle. For ordinary files I/O is handled by the same server that looked up the name, but for devices the I/O server may be different than the server that looks up the file name (this scheme permits devices on diskless workstations to be accessed remotely). Sprite file servers support read and write operations of arbitrary size and alignment.

Sprite's file system emphasizes caching and scalability. Both clients and servers cache files in their main memories, reducing contention for network and disk bandwidth, and file-server processors [Nelson et al. 1988]. The size of the file cache varies dynamically as the demands for file data and virtual memory change: a variable cache size permits applications to perform better than in systems with a fixed partition between file data and virtual memory. The I/O server is responsible for ensuring that a process reading a file sees the most recently written data; in particular, it disables client caching for a file if one host has the file open for writing while another host is accessing it. If a server crashes, or there is a network partition, clients use an idempotent reopen protocol to reestablish the state of their open files with the server and ensure that cached file data remains valid [Baker & Ousterhout 1990]. Sprite uses a block-based file access model. Files are stored in blocks that may or may not be contiguous on disk, and not all of a file need be in memory at once. A file is transferred to and from its I/O server in blocks of 4 Kbytes.

Amoeba splits naming and access into two different servers, a directory server and a file server, in order to provide flexibility. The *directory server* translates names into capabilities, and permits processes to create new mappings of names to capabilities and sets of capabilities. It places no restrictions on the location of objects referenced by a directory, thus one directory may contain entries for files on different file servers or objects that are not files. (By comparison, this would typically not be possible in a system that provided a single combined file and directory service.) It automatically replicates directory entries as they are created, and replicates files asynchronously.

The standard Amoeba file server, known as the *Bullet Server*, emphasizes network transfer speed and simplicity [van Renesse et al. 1989]. The Bullet Server provides an immutable file store, which simplifies file replication. The server's principal operations are *read-file*, *create-file*, and *delete-file*. A process may create a new file, specifying its initial contents and receiving a capability for it. It may then modify the contents, but the file may not be read until it has been *committed*. Once the process has committed the file, it is immediately written through to disk for reliability. (Write-through may be disabled at the option of the caller, but this option is rarely used in practice.) At this point, the file may be read by anyone with the appropriate permission, but may never be modified. The only permissible operations on a committed file are reading and deletion.

In addition to its goal of simplicity, the implementation of the Bullet Server has been influenced by the distributed nature of Amoeba's software architecture. Since the Bullet Server runs on a dedicated machine, it is normally run as a collection of threads within the kernel, but it can equally well run in user space at the cost of some additional copying between the user process and the kernel thread that manages disks. All files are stored contiguously in memory and on disk. The server alleviates fragmentation problems by compacting memory and disks as needed. It is responsible for replicating files on multiple disks, while a separate "object manager" replicates files on multiple instances of the Bullet Server. Because of the distinction between the file service and the directory service, the Bullet Server provides a mechanism for garbage-collecting files that are not referenced after a period of time. It caches files, so read operations do not necessarily result in disk accesses. However, Amoeba's dynamic processor allocation suggested that new processes would be allocated to different processors over time, so client caching would be less beneficial than in a workstation-based environment. As a result, clients do not cache files, and each read must result in a network transfer. File data may be transferred in any unit up to the maximum RPC buffer size.

Although both Amoeba and Sprite have location transparent file systems, they are very different. First, Amoeba permits transparent replication of files and directory entries. Replication of files is simple because they are immutable; replication of directory entries is more complicated and trades some performance for reliability, as indicated below. Second, the Bullet Server is simpler than Sprite's file system but it enforces some restrictions. Since files are immutable, some services that can be provided by Sprite's file system have to be provided in other services. For example, Amoeba needs a logging service to manage append-only files, which currently result in entire files being copied each time data are appended. Some other UNIX file semantics are similarly hard to emulate in Amoeba without substantial overhead: for example, to emulate the *write* kernel call correctly—without buffering—a process that has a file open for reading and writing must copy the file completely each time it switches from writing the file to reading it. Furthermore, since files are required to be contiguous, the Bullet Server cannot deal with files larger than the size of its physical memory. Third, the Bullet Server does not do client caching. A file has be transferred across the network each time it is accessed. When caching would otherwise have eliminated a network transfer, the lack of caching puts more load on the network and increases latency. Fourth, unlike the Bullet Server, a Sprite file server must dedicate a significant amount of memory to maintain state about open files. The Bullet Server only keeps track of new files that have not yet been committed, and it removes any such file that is not accessed after a prolonged interval.

Operation		**Delay** (msec)			
		Amoeba		**Sprite**	
open-close	*foo*	7.2		9.7	
	a/b/c/foo	7.6		10.4	
read				CACHE	NOCACHE
	10 Kbytes	14.0		2.8	18.6
	100 Kbytes	123.0		21.7	167.4
create-delete		BULLET	BULLET/DIR	CACHE	NOCACHE
	no data	33.0	288.0	50.9	50.9
	10 Kbytes	86.0	312.0	67.1	84.9
	100 Kbytes	367.0	617.0	101.4	411.1

Table 2: File system performance of Amoeba and Sprite. Subheadings indicate multiple measurements for the purpose of distinguishing between factors affecting performance. The "open-close" benchmark measures the time to open and close a file in Sprite, or obtain a capability for a file in Amoeba. The "read" benchmark measures the time to read a file on a client. The file was not cached on the client in Amoeba; for Sprite, the measurement shows the measurement with client caching allowed (CACHE), followed by the measurement without client caching (NOCACHE). The "create-delete" benchmark simulates the use of a temporary file, creating and later deleting a file that it transfers data to and from. For Amoeba, the measurement shows the costs of communication only with the Bullet server (BULLET) and also with the directory server (BULLET/DIR). Both measurements include the cost of writing files through to disk. For Sprite, the measurement again shows the performance with and without client caching. Measurements were made on Sun 3/60 workstations connected by a 10-Mbit Ethernet.

We compared the performance of the file systems of Amoeba and Sprite, using three file system benchmarks from Ousterhout's operating system performance analysis [Ousterhout 1990]. The results of these benchmarks appear in Table 2. The "open-close" benchmark, on Sprite, measures the elapsed time to open a file and then close it again. In Amoeba, this measures the time to lookup the capability in the directory server. Table 2 shows the time to open and close a file with a name containing one element, *foo*, and the time for a name containing four elements, *a/b/c/foo*.

The "read" benchmark measures the time to read 10 Kbytes and 100 Kbytes from a file server. The measurements for Sprite show two numbers, corresponding to measurements with and without client caching enabled, respectively. The benchmark demonstrates the effects of client caching and file system overhead: with client caching enabled, Sprite outperforms Amoeba, but without client caching, Sprite is slower. The latter difference arises because Sprite transfers data only in 4-Kbyte units, and it performs additional copying that Amoeba's RPC system avoids.

Finally, the "create-delete" benchmark simulates the use of a temporary file. It measures the time to create a file, write a fixed amount of data to it, and close it; then open the file, read the data from it, and close it; and finally delete it. Like Ousterhout, we varied the amount of data, transferring no data, 10 Kbytes, and 100 Kbytes. Amoeba applications can use capabilities for temporary files without registering the capabilities in a directory, so the measurements for Amoeba show first the cost of creating and deleting a file without registering a capability for the file with the directory server, and then the cost including the additional overhead of registering a directory entry, replicating it, and removing it. In each case, the file is written through to disk for reliability. For Sprite, in the case of non-empty files, the measurement again shows the performance with and without client caching.

Table 2 shows that Sprite's file system is slower than Amoeba's for opening files, but is much faster than Amoeba's when client caching obviates the need for network transfers. The benefits of client caching on machines with large physical memory have been shown before [Nelson et al. 1988], and this comparison further illustrates the point: despite optimizations to store files contiguously in memory and transfer them in a single operation, Amoeba's file system would benefit from caching files in the memory of each processor.[2] Client caching of immutable files could be implemented in a natural fashion in Amoeba, as in the Cedar File System [Gifford et al. 1988], but caching of newly-created files would be more difficult.

3.4 Process Management

The final area of comparison is process management. Amoeba's process model was influenced by both the distributed nature of Amoeba applications and the use of a centralized

[2] A higher-level comparison of the systems, such as the modified Andrew benchmark [Ousterhout 1990], would provide additional insights into performance differences. Unfortunately, however, any comparison involving UNIX-based programs would be affected more by overhead in Amoeba's UNIX emulation than by differences in their file systems. In particular, as the next section indicates, native-Amoeba process creation is faster than Sprite's, but process creation that is compatible with UNIX is extremely slow.

processor pool. Sprite provides facilities comparable to BSD UNIX, combined with a mechanism to use idle workstations.

PROCESS MODEL

Amoeba is designed to provide high performance communication between clients and servers, and it has a fairly simple and efficient process model. It provides virtual memory, allowing processes to use the full addressing range available on the hardware, but it does not perform swapping or demand-paging: *i.e.*, a process is resident in memory at all times during its lifetime. The lack of paging helps to improve the performance of user-level RPC, because there is no need to verify that each page of a buffer is physically in memory. Amoeba provides threads as a method for structuring servers. A server process can inexpensively create a new thread of control within its address space. Multiple threads can service multiple RPCs in parallel, and can share resources (such as the buffer cache of a file server).

Process creation in Amoeba is designed to work efficiently in an environment with a processor pool. As described below, each new process is likely to run on a new processor, so Amoeba is tailored for remote program invocation. A process starts a new program using the *exec_file* library call, specifying the name of an executable file and a set of capabilities with which to execute the program. This sequence avoids the need to copy the state of the creating process, as in a UNIX *fork* call. (The Amoeba *exec_file* call is comparable to the *run* call in LOCUS [Popek & Walker 1985]).

Sprite's process model is nearly identical to that of BSD UNIX. Sprite supports demand-paging, but it uses a regular file rather than a separate paging area. This permits the system to use the main memory on a file server to cache pages for clients. To execute a new program in Sprite, as in UNIX, a process *fork*s a copy of itself and then issues a second kernel call (*exec*) to replace its virtual image. In addition, Sprite's version of the *fork* kernel call optionally permits the newly created child process to share the data segment of its parent. This option is not commonly used in Sprite; however, it provides semantics that are similar to lightweight threads in Amoeba, so a comparison of the two can demonstrate the performance advantage of threads for server processes.

Table 3 shows the costs associated with process management. It shows the speed of context switching, the time to create a shared-memory thread or process, the time to create an identical process that does not share memory, and the time to invoke a program that immediately exits. The context-switch benchmark measured the fastest possible round-trip context switch in each system: a null RPC in Amoeba and synchronization using shared memory and kernel-level wakeup calls in Sprite. Context switching is significantly faster in Amoeba than in Sprite. The difference in performance is largely a function of the overhead of a highly layered mechanism for synchronization and scheduling in Sprite, as well as the overhead of supporting virtual memory. The table next gives the time to create a new entity that shares memory with its parent—a thread in Amoeba or a process in Sprite. Thread creation is faster than process creation, as one might expect, because the kernel performs substantially less bookkeeping. By comparison, Amoeba is much slower at creating a new process with an unshared copy of the state of its parent. This operation is expensive in Amoeba because the only way to perform the equivalent of a UNIX *fork* is to communicate

Operation	**Time** (msec)	
	Amoeba	**Sprite**
Context switch	0.5	1.6
Thread creation	2.4	(12.5)
fork	(169.5)	13.6
Program invocation	58.0	71.6

Table 3: Performance of context switching and process creation on Sun 3/60 workstations. Parenthesized numbers indicate operations that are not performed under normal circumstances: shared memory *fork*s in Sprite and UNIX-like *fork*s in Amoeba. The "context switch" benchmark measures the cost of round-trip communication (*i.e.*, two context switches). Amoeba outperforms Sprite in all areas but a UNIX-like *fork*. The high cost of creating a new Amoeba process from an existing one is attributable to overhead relating to UNIX compatibility; normally, this cost is avoided because processes in Amoeba invoke programs without an intervening *fork*.

with a special server that will suspend the forking process and copy its state from user-level. Finally, Table 3 shows the performance of creating a new process from an executable image, and waiting for it to exit. The combination of process creation and termination in Sprite is moderately slower than in Amoeba. The additional overhead in Sprite is due to the wasted effort of creating a new address space for a child process that immediately replaces its image.

All in all, these comparisons suggest that UNIX compatibility has had a great impact on the performance of process management in the two systems. The desire to support a wide range of UNIX applications resulted in Sprite's providing virtual memory, which slows context switching, and a *fork/exec* paradigm, which slows process creation. In contrast, Amoeba's poor performance for UNIX-compatible *fork*s arises more from an inefficient UNIX emulation than from a particular design decision.

Processor allocation

Since the designers of Amoeba assumed that a system would contain many processors per user, they arranged for the system to assign processes to processors transparently. The *run server* selects a processor for a new process based on factors such as processor load and memory usage. (The only exceptions to automatic host placement are dedicated server processes, which are explicitly placed on the specialized servers shown in Figure 1.) Because of the assumption of many processors, Amoeba makes no provisions for associating individual users with specific processing resources, and instead relies on automatic distribution of load. There is no mechanism to migrate a process atomically to a new processor once it has started execution, though there is a facility to checkpoint the state of a process and create a new process elsewhere with the same state.

Sprite's basic model assumes a one-to-one mapping between users and workstations, and it assumes that Sprite would be used mostly for traditional applications. It further assumes that users want a guaranteed response time for interactive processes, and that most processes are either interactive or short-lived. As a result, Sprite gives each user priority on one workstation and run all processes there by default. Nevertheless, there are often many idle machines in a collection of personal workstations, so Sprite provides a mechanism to take advantage of idle hosts transparently using process migration [Douglis & Ousterhout 1991].

Logically, a process in Sprite executes on the host of the user that invoked it (known as its "home machine"), though it may physically migrate between machines at any time. The *fork* kernel call creates a new process that physically executes on the same host as its parent, wherever that may be, while logically executing on the parent's home machine. The *exec* call permits a process to specify a new execution site, so that the address space of the process need not be transferred when the process migrates. Alternatively, a process may migrate at some other time, in which case any modified pages in its address space are flushed to a shared file server and paged in by the process's new host. Transparency is assured by forwarding location-dependent operations to and from a process's home machine, using kernel-to-kernel RPC. For example, a request by a remote process to get the time of day would be forwarded home; the call would take about two milliseconds, compared to 210 microseconds in the local case.

Though Sprite could make remote execution the default case, by starting all new programs on idle hosts, it currently starts a new program on the same host as its parent unless specified otherwise. A few system programs, such as a parallel *make* [Feldman 1979] facility, take advantage of remote execution by default. A centralized daemon process called *migd* keeps track of idle hosts and allocates them to processes when needed. A process such as *make* can request an arbitrary number of hosts and start a command, such as a compilation, on each host. The process can continue to use the host until it is notified by the daemon that the host has been reclaimed. A workstation is reclaimed when its owner returns, or if no additional hosts are available and one process is using more than its fair share of hosts [Douglis & Ousterhout 1991].

Operation	**Time** (msec)	
	Amoeba	**Sprite**
Local	58	72
Remote (specified)	84	116
Remote (unspecified)	95	131

Table 4: Performance of program invocation. Local program invocation is faster in Amoeba than Sprite, as is remote invocation if a new processor must be selected. Sprite normally executes locally or reuses the same host multiple times for remote invocation, with minimal costs of 72 and 116 milliseconds respectively. Amoeba normally selects a processor each time a program is invoked, for a minimal cost of 95 milliseconds. Measurements were made on Sun 3/60 workstations connected by a 10-Mbit Ethernet.

Table 4 shows the costs of creating a new process to execute a small program that immediately exits. The first entry in the table corresponds to the cost of creating a local program, from Table 3. The second entry shows the cost of running the same program on a remote host known in advance, while the third shows the cost of running it on a remote host determined at invocation time. The normal case in Amoeba is to select a remote host at invocation time, while in Sprite process creation is usually local or on a predetermined remote host. The cost of remote invocation in Sprite is additionally affected by the time to transfer open files [Douglis & Ousterhout 1991], which in Amoeba are capabilities that require no additional processing overhead.

In both systems, centralized scheduling has its drawbacks. Amoeba provides no support for multiple parallel applications to cooperate and scale their parallelism to use the system efficiently; instead, it will let each application create as many processes as processors, and then time-share each processor among all processes in a round-robin fashion. In Sprite, the default of local execution means that users can overload their own workstation if they run programs that do not execute remotely—the system will not automatically spread load. Also, an application may use another workstation only if it is idle and no other application is already using it. This rule is based on the assumption that processes that run remotely will be processor-bound and will not operate as efficiently if they are multiprogrammed. As a result, interactive applications may not use the remote execution facility without monopolizing resources they do not fully utilize.

4 Related Work

In the introduction, we noted that there are many other distributed systems, and several of them have similar goals and functionality to Amoeba and Sprite. We briefly describe these systems in the context of the design philosophies we have discussed throughout this paper.

The V System [Cheriton 1988], like Amoeba, provides most system services at user-level via messages. Those services that are internal to the kernel, such as one that provides the current time, are accessed via a message interface as well. Unlike Amoeba, V implements conventional files, using paged virtual memory to access the files from process address spaces. File I/O is based on block transfers rather than whole file transfers or byte streams. Finally, V implements a workstation model similar to Sprite. It uses process migration to execute new tasks on lightly loaded workstations, but it runs "guest" tasks at a lower priority than local ones in order to reduce their impact on interactive response. V provides multicast communication to support distributed applications.

Chorus [Rozier et al. 1988] is based on a microkernel and message passing as well. Like Amoeba, it implements capabilities and ports, and it runs system services in both kernel mode and user mode. It permits the execution of multiple operating system interfaces layered on a kernel; in particular, it supports a binary-compatible UNIX interface through the use of user-level managers for processes, pipes, and devices. It also provides support for real-time facilities, but provides no special support for distributed applications or load leveling.

Locus [Popek & Walker 1985] has more similarities to Sprite than to Amoeba, as it is a UNIX-compatible system based on a monolithic kernel. It supports a transparent network-wide file system with provisions for redundant data storage. It also supports remote execution with automatic load leveling [Kiser 1990]. However, as it was designed for a small collection of time-sharing mainframes, it has only limited support for distributed applications.

Mach [Accetta et al. 1986] is similar to both Amoeba and Sprite in various ways. Mach integrates virtual memory with its message-based communication system, using memory mapping techniques and copy-on-write semantics to improve performance. It allows user-level processes to service requests to read and write memory segments. Mach is compatible with BSD UNIX and was initially implemented as a modification of the BSD UNIX monolithic kernel. Mach was later separated into a Mach microkernel and a separate user-level UNIX server process, which offered comparable performance to previous monolithic versions of Mach [Golub et al. 1990]. Mach is organized around the workstation model: each host is autonomous, with its own processes and file system. However, Mach's network-transparent communication is used by other facilities, such as Avalon [Detlefs et al. 1988], to support distributed applications.

Finally, Plan 9 [Pike et al. 1990] offers an interesting perspective on the subject of processor allocation. Like Amoeba, it distinguishes between graphics terminals (with a small amount of processing capacity) and computation-intensive processors. However, rather than providing a large number of independent processors, Plan 9 centralizes its processing power in a small number of multiprocessors. The designers of Plan 9 argue that this centralization is the most cost-effective way to provide a large amount of processing power. Though Plan 9 does not provide process migration—which offers less benefits in a system with a small number of shared processors than one with a larger number of "independently owned" workstations—the execution environment on a graphics terminal, relative to a CPU server, is similar to Sprite's "home machine."

5 Project Evolution

Both Amoeba and Sprite have been under development for several years. In this section we summarize the development history of the two projects, describe the ways in which the systems are currently used, and discuss the current directions of research for Sprite and Amoeba.

5.1 Amoeba

The initial work on Amoeba began in 1981. By 1984 a working prototype existed and was selected as the basis for a European-wide distributed system as part of the EEC sponsored COST-11 Mandis project. The Mandis project involved connecting sites in Holland, England, and Norway in a transparent distributed system based on Amoeba. This experience led to the discovery of various problems [Tanenbaum et al. 1990] and a major redesign, leading to the current version, Amoeba 5.0.

Amoeba is currently being used in the European space industry for the transmission of of real-time digital video over LANs, as well as other applications where high performance and parallelism are important. Amoeba has evolved from a one student's PhD research to a system in in daily use by about a dozen people at the Vrije Universiteit (faculty members, students, and staff) for a wide variety of projects involving distributed and parallel computing. It is also available to universities (on an "as is" basis) and to companies (on a commercial basis).

Current research is concentrated in the following areas:

Parallel applications. The Amoeba group has designed and implemented a language for parallel programming called Orca, which runs on Amoeba, and eases the task of write applications that use massive parallelism, such as playing chess. Research is continuing on the language, runtime system, and parallel applications.

Group communication. Current distributed systems, are based on a point-to-point communication paradigm, usually using RPC. One project is looking at the use of group communication in distributed computing, for example, to support replicated services [Kaashoek & Tanenbaum 1991].

Distributed shared memory. An object-based distributed shared memory system based on Amoeba allows programs to share data objects on machines that do not have physically shared memory, as though they did. This system attains a high degree of speedup on certain classes of problems. Work is continuing in improving and using the distributed shared memory.

Wide-area transparent systems. With the current system, it is possible to have Amoeba machines in different countries work together completely transparently. An authorized user logged into Amoeba at Cornell, for example, can use the processor pool and file server in Amsterdam as though it were local. Research into transparent distributed computing is continuing, to better understand the interaction between wide-area computing and transparent computing.

5.2 Sprite

The design of Sprite began in the Fall of 1984, and implementation began in 1985. By the Fall of 1987 the system had sufficient functionality to support its own development, and members of the Sprite project began using Sprite for all their day-to-day computing. Additional users began using Sprite in 1988. As of the Fall of 1991 the Sprite user community numbers more than 50, of which 20–30 do all their day-to-day computing on Sprite. Sprite currently supports research in operating systems, computer-aided design, and computer architecture, plus a number of administative functions. Most people use Sprite as though it were UNIX, though they implicitly take advantage of Sprite's process migration and file caching. At least one person has used Sprite to run large numbers of simulations in parallel on 10–15 idle machines, obtaining the equivalent of over 800% effective utilization relative to a single machine [Douglis & Ousterhout 1991].

The original Sprite research on network file systems and process migration is now complete, but a number of new research projects are underway. Most of the new projects concern high-performance file systems and are being carried out as part of the RAID project (Redundant Arrays of Inexpensive Disks) [Patterson et al. 1988]. Current research includes the following topics:

Log-structured file systems (LFS). LFS is a new approach to disk storage management where the only structure on disk is an append-only log. This structure allows information to be written to disk an order of magnitude more efficiently than previous approaches, but it introduces interesting problems with garbage collection [?].

Striping files. Techniques are being investigated for improving the bandwidth of large-file accesses by spreading the files across multiple disks and even multiple file servers.

Buffering techniques. For sequential accesses to large files, buffering may make more sense than caching, particularly with disk arrays to provide high bandwidth. The Sprite project is studying how best to use buffer/cache memory and how reconcile the buffering and caching approaches.

Reliability. Another project is investigating the recovery of file system state after server crashes. One of the project's goals is to reduce server recovery time to only a few seconds, so that crashes are almost invisible to the rest of the system [Baker & Ousterhout 1990].

Mach interoperability. Micro-kernel approaches are being explored by porting the Sprite kernel to run as a user-level server process on the Mach operating system.

6 Conclusions

This paper has compared two distributed systems that share many goals but diverge on two philosophical grounds. Their approaches toward distributed applications and resource allocation account for many differences in their designs, and in their performance. The issues addressed in this paper lead to several conclusions.

First, Amoeba helps to disprove the notion that the performance of microkernels need be inferior to monolithic kernels. Although the cost of simple operations can be higher if a service is delivered via RPC, many other operations are faster in Amoeba than in Sprite. (Golub, et al., provide even stronger support for this hypothesis, since they were able to compare two versions of the same system rather than two distinct systems [Golub et al. 1990].) By providing services as separate processes, accessed via RPC, the system offers several advantages over a monolithic kernel: simple location transparency, extensibility, and modularity. With a microkernel, it is possible to develop new services at user-level, test them, and then possibly incorporate them into the kernel to obtain higher performance. Given these advantages, we think that microkernels will be the implementation method of choice for future distributed systems.

Second, along the same lines, Amoeba demonstrates the desirability of a uniform communication model. Whether a service is provided at user-level or within the kernel, it is accessed via the same high-performance RPC interface. Services are completely location-transparent, without the need for explicit forwarding of operations (as in Sprite). Applications may take advantage of the distributed nature of the system explicitly, using RPC, or implicitly, using Orca. In contrast, Sprite's organization is restrictive. Sprite does not export its relatively fast kernel-to-kernel RPC to user-level, and it lacks flexibility in replacing system services.

As systems become more and more distributed, fast and simple communication at user level will be even more important.

Third, Sprite demonstrates the benefits of client caching. Just as communication-intensive applications can take advantage of high-performance IPC, file-intensive applications obtain significantly better performance if network transfers can be avoided. Client caching also helps to alleviate contention for networks and file servers [Nelson et al. 1988; Satyanarayanan et al. 1985]. It has not been implemented in Amoeba because of Amoeba's processor pool model. However, if it were combined with more sophisticated processor allocation, using the same processor repeatedly for related but sequential applications, the performance and scalability of Amoeba's file system should improve. Like communication, client caching will become more important as distributed systems grow larger.

Fourth, the comparison between Amoeba and Sprite shows the advantages of a hybrid system containing both workstations *and* a processor pool. Dedicated personal workstations guarantee fast interactive response: in a distributed system, it should be unacceptable for a small number of users to monopolize the resources of the system in a way that degrades the performance of other users beyond some threshold. Once each user has a workstation, additional processing capacity can be shared by all, providing cost-effective power for parallel, computation-intensive applications. The flexibility offered by this hybrid approach will be necessary as hardware becomes cheaper and parallel programming becomes more common.

Fifth, compatibility with UNIX has been a double-edged sword. On the one hand, the decision to make Sprite mostly compatible with BSD UNIX has helped Sprite to mature to a "real system" in a relatively short time. Though Amoeba is easily used for some applications—distributed programs using Orca, and simple UNIX-based programs—it is not yet ready to serve as a replacement for a system like UNIX on a day-to-day basis—nor was it intended for that use. On the other hand, UNIX compatibility is not necessarily a bed of roses. The UNIX model of performing interprocess communication through the file system has hurt performance and complicated the kernel implementation. Support for UNIX file system semantics, such as shared file descriptors, has complicated the implementation of process migration [Douglis & Ousterhout 1991]. Supporting the UNIX process model at the lowest level of the system can detract from the performance of normal operation (witness the cost of context switching and program invocation in Sprite), while supporting full UNIX semantics only with a user-level emulation layer can be unacceptably inefficient (for example, a *fork* in Amoeba). Given the impact of UNIX compatibility on both the performance and the application domain of a system, one must make a conscious decision about whether to be compatible, and how.

Finally, one should consider the performance differences between Amoeba and Sprite in light of their development. While some of the differences are attributable to fundamental differences in their designs, such as the mechanism for user-level interprocess communication, other differences are due at least in part to inefficiencies in implementation. Though Amoeba has been programmed with an eye toward high performance throughout its history, and has undergone several substantial rewrites, its UNIX-compatibility library is especially inefficient. Some of its inefficiency results from the imperfect mapping between UNIX and Amoeba operations, but the performance of the compatibility library could be significantly improved, given time. Similarly, Sprite has several important components (especially with respect to context-switching and scheduling) that have barely changed since its inception. Thus, we have used performance as an obvious metric for comparison, but differences in performance should be considered in the context of design versus implementation.

Amoeba and Sprite continue to evolve. We hope that the issues addressed in this paper will result in positive changes to the implementation of these two systems and the design of future distributed systems.

Availability

Amoeba and Sprite are both available. For information about Amoeba, please contact Andrew S. Tanenbaum (email: ast@cs.vu.nl or FAX +31 20 6427705). To get more information about Sprite, please contact the Sprite group by email (sprite-request@sprite.berkeley.edu).

Acknowledgments

Erik Baalbergen, Henri Bal, Arnold Geels, Dick Grune, Mike Kupfer, Darrell Long, Sape Mullender, Mike Nelson, Robbert van Renesse, Guido van Rossum, Greg Sharp, Kees Verstoep, and Brent Welch provided comments on early drafts of this paper, which improved its content and presentation substantially. We also wish to thank the referees for their input, which further helped to improve the paper.

References

V. Abrossimov, M. Rozier, & M. Shapiro. Generic virtual memory management for operating system kernels. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 123–136, December 1989.

M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, & M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX 1986 Summer Conference*, July 1986.

M. Baker & J. Ousterhout. Availability in the Sprite distributed file system. In *Proceedings of the Fourth ACM SIGOPS European Workshop*, Bologna, Italy, September 1990.

H.E. Bal, M.F. Kaashoek, & A.S. Tanenbaum. Experience with distributed programming in Orca. *IEEE CS Int. Conf. on Computer Languages*, pages 79–89, March 1990.

B. N. Bershad, T. E. Anderson, E. D. Lazowska, & H. M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 102–113, December 1989.

A. D. Birrell & B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

J. Dennis & E. Van Horn. Programming semantics for multiprogrammed computation. *Communications of the ACM*, 9:143–155, March 1966.

D.L. Detlefs, M.P. Herlihy, & J.M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, 21(12), December 1988.

F. Douglis & J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience*, 21(8):757–785, August 1991.

S. I. Feldman. Make — a program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–265, April 1979.

D. Gifford, R. Needham, & M. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, March 1988.

D. Golub, R. Dean, A. Forin, & R. Rashid. Unix as an application program. In *Usenix 1990 Summer Conference*, pages 87–95, June 1990.

M.F. Kaashoek & A.S. Tanenbaum. Group communication in the Amoeba distributed operating systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, Arlington, TX, May 1991. To appear.

S. Kiser. Personal communication, 1990.

S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, & H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

M. Nelson, B. Welch, & J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, & B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Usenix 1990 Summer Conference*, pages 247–256, June 1990.

D. Patterson, G. Gibson, & R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD 88*, pages 109–116, Chicago, June 1988.

R. Pike, D. Presotto, K. Thompson, & H. Trickey. Plan 9 from Bell Labs. In *UKUUG Summer 1990 Conference Proceedings*, pages 1–9, London, England, July 1990.

G. J. Popek & B. J. Walker, editors. *The LOCUS Distributed System Architecture*. Computer Systems Series. The MIT Press, 1985.

M. Rozier et al. Chorus distributed operating systems. *Computing Systems*, 1(4), 1988.

M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, & M. West. The ITC distributed file system: Principles and design. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 35–50, Orcas Island, WA, December 1985. ACM.

A. S. Tanenbaum & R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, A. Jansen, & G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

R. van Renesse, A. S. Tanenbaum, & A. Wilschut. The design of a high-performance file server. *Proc. of the 9th Int. Conf. on Distr. Computing Systems*, pages 22–27, June 1989.

G. van Rossum. AIL – a class-oriented stub generator for Amoeba. In *Proceedings of the Workshop on Experience with Distributed Systems*. Springer Verlag, 1989.

B. B. Welch & J. K. Ousterhout. Pseudo devices: User-level extensions to the Sprite file system. In *USENIX 1988 Summer Conference*, pages 37–49, San Francisco, CA, June 1988.

B. B. Welch. *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System*. PhD thesis, University of California, Berkeley, CA 94720, February 1990. Available as Technical Report UCB/CSD 90/567.