Amoeba — A Distributed Operating System for the 1990s

Sape J. Mullender[*] Guido van Rossum[*] Andrew S. Tanenbaum[†]
Robbert van Renesse[†] Hans van Staveren[†]

May 1990

Abstract

Amoeba is the distributed system developed at the Free University (VU) and Centre for Mathematics and Computer Science (CWI), both in Amsterdam. Throughout the project's ten-year history, a major concern of the designers has been to combine the research themes of distributed systems, such as high availability, use of parallelism and scalability, with simplicity and high performance. Distributed systems are necessarily more complicated than centralized systems, so they have a tendency to be much slower. Amoeba was always designed to be used, so it was deemed essential to achieve extremely high performance. We are working hard to achieve this goal — Amoeba is already one of the fastest distributed systems (on its class of hardware) reported so far in the scientific literature.

The Amoeba software is based on objects. An object is a piece of data on which well-defined operations may be performed by authorized users, independent of where the user and object are located, Objects are managed by server processes and named using capabilities chosen randomly from a sparse name space.

Processes consist of a segmented address space shared by one or more threads of control. Processes can be created, managed and debugged remotely and processes may migrate at any point during their execution. Operations on objects are implemented using remote procedure calls.

Amoeba has a unique and fast file system. The file system is split into two parts — the Bullet Server, which stores immutable files contiguously on the disk, and the SOAP Directory Server, which provides a mechanism for giving capabilities symbolic names. The directory server also handles replication and atomicity, eliminating the need for a separate transaction management system.

To bridge the gap with existing systems, Amoeba provides a UNIX emulation facility. This facility contains a library of UNIX system call routines, each of which does its work by making calls to the various Amoeba server processes.

Since the original goal of the design was to build a fast system, some actual performance measurements of the current implementation are given. A remote procedure call can be performed in 1.4 msec on Sun-3/50 class machines, and the file server can deliver data continuously at a rate of 677 kbytes/sec.

This paper was published in the May 1990 Special Issue "Recent Developments in Operating Systems" of IEEE Computer.

The work described here has been supported by grants from NWO, the Netherlands Research Organization, SION, the Foundation for Computer Science Research in the Netherlands, OSF, the Open Software Foundation, and Digital Equipment Corporation.

1 Introduction

The 1970s were dominated by medium to large sized timesharing systems, typically supporting 10 to 100 on-line terminals. In the 1980s, personal computing became popular, with many

[*]CWI, the Centre for Mathematics and Computer Science
[†]Vrije Universiteit

organizations installing large numbers of PCs and engineering workstations, usually connected by a fast local area network. In the 1990s, computer prices will drop so low that it will be economically feasible to have 10, 20, or perhaps 100 powerful microprocessors *per user*. The key issue is how to organize all this computing power in a simple, efficient, fault-tolerant, and especially, easy to use way. In this paper we describe a distributed operating system that meets this challenge.

The basic problem with current networks of PCs and workstations is that they are not *transparent*, that is, the users are clearly conscious of the existence of multiple machines. One logs into a specific machine and uses that machine only, until one does a remote login to another machine. Few, if any, programs can take advantage of multiple CPUs, even if they are all idle, for example. An operating system for connecting up a number of autonomous computers is usually called a *network operating system*.

In contrast, the kind of system we envision for the 1990s appears to the users as a single, 1970s centralized timesharing system. Users of this system are not aware of which processors their jobs are using (or even how many), they are not aware of where their files are stored (or how many replicated copies are being maintained to provide fault tolerance) or how communication is taking place among the processes and machines. The whole thing just looks like a single big timesharing system. All the resource management is done completely automatically by what is called a *distributed operating system*.

Few such systems have been designed, and even fewer have been implemented. Fewer still, are actually used by anyone (yet). One of the earliest distributed systems was the Cambridge Distributed Computing System [Needham and Herbert, 1982]. Later, other systems were developed, such as Locus [Walker et al., 1983], Mach [Accetta et al., 1986] V-Kernel [Cheriton, 1988], and Chorus [Rozier et al., 1988]. Most of the classical distributed systems literature, however, describes work on parts of, or aspects of distributed systems. There are many papers on distributed file servers, distributed name servers, distributed transaction systems, and so on, but there are few on whole systems.

In this paper we will describe a research project — Amoeba — that has successfully constructed a working prototype system. We will cover most of the traditional operating system design issues, including communication, protection, the file system, and process management. We will not only explain what we did, but also why we did it.

2 Overview of Amoeba

The Amoeba Project [Mullender and Tanenbaum, 1986] is a joint effort of groups at the Free University (VU), and the Centre for Mathematics and Computer Science (CWI), both in Amsterdam. The VU group is led by Andrew S. Tanenbaum, the CWI group by Sape J. Mullender. The project has been underway now for nearly ten years and has gone through numerous redesigns and reimplementations as design flaws became glaringly apparent. This paper describes the Amoeba 4.0 system, which was released in 1990.

2.1 The Amoeba Hardware Architecture

The Amoeba hardware architecture is shown in Figure 1. It consists of four components: workstations, pool processors, specialized servers, and gateways. The *workstations* are intended to execute only processes that interact intensively with the user. The window manager, the command interpreter, editors, CAD/CAM graphical front-ends are examples of programs that might be run on workstations. The majority of applications do not usually interact much with the user and are run elsewhere.

Amoeba has a *processor pool* for providing most of the computing power. It typically consists of a large number of single-board computers, each with several megabytes of private memory and a network interface. The VU has 48 such machines, for example. A pile of diskless, terminalless workstations can also be used as a processor pool.

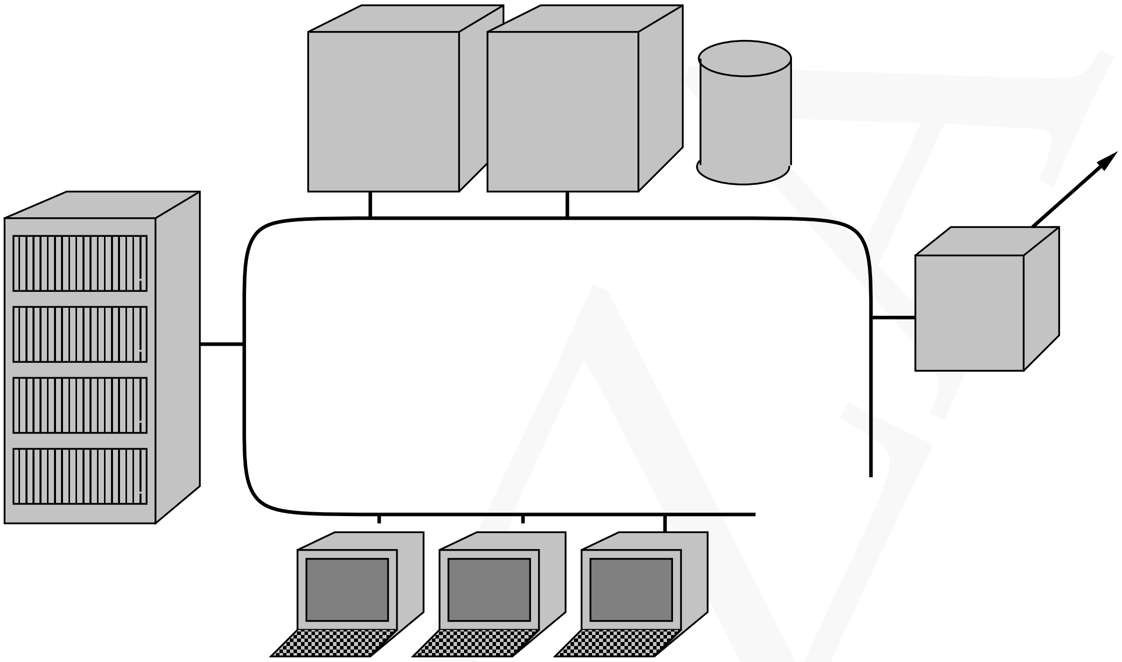

Figure 1: The four components of the Amoeba architecture.

When a user has an application to run, e.g., a *make* of a program consisting of dozens of source files, a number of processors can be allocated to run many compilations in parallel. When the user is finished, the processors are returned to the pool so they can be used for other work. Although the pool processors are all multiprogrammed, the best performance is obtained by giving each process its own processor, until the supply runs out.

It is the processor pool that allows us to build a system in which the number of processors exceeds the number of users by an order of magnitude or more, something quite impossible in the personal workstation model of the 1980s. The software has been designed to treat the number of processors dynamically, so new ones can be added as the user population grows. Furthermore, when a few processors crash, some jobs may have to be restarted, and the computing capacity is temporarily lowered, but essentially the system continues normally, providing a degree of fault tolerance.

The third system component consists of the *specialized servers*. These are machines that run dedicated processes that have unusual resource demands. For example, it is best to run file servers on machines that have disks, in order to optimize performance.

Finally, there are gateways to other Amoeba systems that can only be accessed over wide area networks. In the context of a project sponsored by the European Community, we built a distributed Amoeba system that spanned several countries. The role of the gateway is to protect the local machines from the idiosyncracies of the protocols that must be used over the wide area links.

Why did we choose this architecture as opposed to the traditional workstation model? Primarily because we believe that the workstation model will become inappropriate in the 1990s, as it becomes possible to give each user 10 or 100 processors. By centralizing the computing power, we allow incremental growth, fault tolerance, and the ability for a single large job to temporarily obtain a large amount of computing power.

Server port	Object number	Rights field	Check field	
48	24	8	48	Bits

Figure 2: Structure of a capability. The service port identifies the service that manages the object. The object number specifies which object (e.g., which file). The rights tell which operations are permitted. The check field provides cryptographic protection to keep users from tampering with the other fields.

2.2 The Amoeba Software Architecture

Amoeba is an *object-based* system using clients and servers. Client processes use remote procedure calls to send requests to server processes for carrying out operations on objects. Each object is both identified and protected by a *capability*, as shown in Figure 2. Capabilities have the set of operations that the holder may carry out on the object coded into them and they contain enough redundancy and cryptographic protection to make it infeasible to guess an object's capability. Thus, keeping capabilities secret by embedding them in a huge address space is the key to protection in Amoeba. Due to the cryptographic protection, capabilities are managed outside the kernel, by user processes themselves.

Objects are implemented by server processes that manage them. Capabilities have the identity of the object's server encoded into them (the Service Port) so that, given a capability, the system can easily find a server process that manages the corresponding object. The RPC system guarantees that requests and replies are delivered at most once and only to authorized processes. Communication and protection are discussed in Section 3.

Although, at the system level, objects are identified by their capabilities, at the level where most people program and do their work, objects are named using a human-sensible hierarchical naming scheme. The mapping is carried out by the *Directory Service* which maintains a mapping of ASCII path names onto capabilities. The Directory Server has mechanisms for performing atomic operations on arbitrary collections of name-to-capability mappings. The Directory Server is described in Section 4.

Amoeba has already gone through several generations of file systems. Currently, one file server is used practically to exclusion of all others. The Bullet Server, which got its name from being faster than a speeding Bullet, is a simple file server that stores immutable files as contiguous byte strings both on disk and in its cache. It is also described in Section 4.

The Amoeba kernel manages memory segments, supports processes containing multiple threads and handles interprocess communication. The process-management facilities allow remote process creation, debugging, checkpointing, and migration, all using a few simple mechanisms explained in Section 5.

All other services, (such as the directory service) are provided by user-level processes, in contrast to, say, UNIX , which has a large monolithic kernel that provides these services. By putting as much as possible in user space, we have achieved a flexible system, and have done this without sacrificing performance.

In the Amoeba design, concessions to existing operating systems and software were carefully avoided. Since it is useful to be able to run existing software on Amoeba, a UNIX emulation service, called *Ajax* has been developed. It is discussed in Section 6.

3 Communication in Amoeba

Amoeba's conceptual model is that of a client thread[1] performing operations on objects. For example, on a file object, a common operation is reading some data from it. Operations are implemented by making *remote procedure calls* [Birrell and Nelson, 1984]. A client sends a *request* message to the *service* that manages the object. A server thread accepts the message, carries

[1] *Thread of control* or *light-weight process*.

out the request, and sends a *reply* message back to the client. For reasons of performance and fault tolerance, frequently multiple server processes jointly manage a collection of objects of the same type to provide a *service*.

3.1 Remote Procedure Calls

The kernel provides three basic system calls to user processes:

- DoOperation
- GetRequest
- SendReply

The first one is used by clients to get work done. It consists of sending a message to a server and then blocking until a reply comes back. The second one is used by servers to announce their willingness to accept messages addressed to a specific port. The third one is also used by servers, to send back replies. All communication in Amoeba is of the form: a client sends a request to a server, the server accepts the request, does the work, and sends back the reply.

Although systems programmers would no doubt be content to live with only these three system calls, for most application programmers they are far to primitive. For this reason a much more user-oriented interface has been built on top of this mechanism, to allow users to think directly in terms of objects and operations on these objects.

Corresponding to each type of object is a *class*. Classes can be composed hierarchically; that is, a class may contain the operations from one or more underlying classes. This *multiple inheritance* mechanism allows many services to inherit the same interfaces for simple object manipulations, such as for changing the protection properties on an object, or deleting an object. It also allows all servers manipulating objects with file-like properties to inherit the same interface for low-level file I/O: read, write, append. The mechanism resembles the file-like properties of UNIX pipe and device I/O: the UNIX *read* and *write* system calls can be used on files, terminals, pipes, tapes and other I/O devices. But for more detailed manipulation, specialized calls are available (*ioctl*, *popen*, etc.).

3.2 RPC Transport

The AIL compiler generates code to marshal or unmarshal the parameters of remote procedure calls into and out of message buffers and then call the Amoeba's transport mechanism for the delivery of request and reply messages. Messages consist of two parts, a *header* and a *buffer*. The header has a fixed format and contains addressing information (including the capability of the object that the RPC refers to), an operation code which selects the function to be called on the object, and some space for additional parameters. The buffer can contain data. A file read or write call, for instance, uses the message header for the operation code plus the length and offset parameters, and the buffer for the file data. With this set-up, marshalling the file data (a character array) takes zero time, because the data can be transmitted directly from and to the arguments specified by the program.

3.3 Locating Objects

Before a request for an operation on an object can be delivered to a server thread that manages the object, the location of such a thread must be found. All capabilities contain a Service Port field, which identifies the service that manages the object the capability refers to. When a server thread makes a *GetRequest* call, it provides its service port to the kernel, which records it in an internal table. When a client thread calls *DoOperation*, it is the kernel's job to find a server thread with an outstanding *GetRequest* that matches the port in the capability provided by the client.

The Amoeba Interface Language

Interfaces for object manipulation are specified in a notation, called the Amoeba Interface Language (AIL) [van Rossum, 1989], which resembles the notation for procedure headers in C with some extra syntax added. This allows automatic generation of client and server stubs. The Amoeba class for standard manipulations on file-like objects, for instance, could be specified as follows:

```
class BasicIo [1000..1199] {

const BIOSIZE = 30000;

BioRead(*, in unsigned offset, in out unsigned bytes,
out char buffer[bytes:bytes]);

BioWrite(*, in unsigned offset, in out unsigned bytes,
in char buffer[bytes:BIOSIZE]);
};
```

This AIL specification tells the stub compiler that the operation codes for *BasicIo* must be allocated in the range 1000 to 1199. A clash of the operation codes for two different classes only matters if these classes are both inherited by another, bringing them together in one interface. Currently, every group of people designing interfaces has a different range from which to allocate operation codes.

The names of the operations, *BioRead* and *BioWrite*, must be globally unique and conventionally start with an abbreviation of the name of the class they belong to. The first parameter is always a capability of the object to which the operation refers. It is indicated by an asterisk. The other parameters are labelled *in*, *out*, or *in out* to indicate whether they are input or output parameters to the operation, or both. Specifying this allows the stub compiler to generate code to transport parameters in only one direction.

The number of elements in an array parameter can be specified by [n: m], where n is the actual number of elements in the array and m is the maximum number. In an *out* array parameter, such as *buffer* in *BioRead*, the maximum size is provided by the caller. In *BioRead*, it is the value of the *in* parameter *bytes*. The actual size of an *out* array parameter is given by the callee and must be less than the maximum. In *BioRead* it is the value of the *out* parameter *bytes* — the actual number of bytes read. On an *in* array parameter, the maximum size is set by the interface designer and must be a constant, while the actual size is given by the caller. In *BioWrite*, it is the *in* value of *bytes*.

The AIL stub compiler can generate client and server stubs routines for a number of programming languages and machine architectures. For each parameter type, *marshalling code* is compiled into the stubs which converts data types of the language to data types and internal representations of AIL. Currently, AIL handles only fairly simple data types (boolean, integer, floating point, character, string) and records or arrays of them. AIL, however, can easily be extended with more data types.

Using the Transport Interface

The transport interface for the server consists of the calls *GetRequest* and *PutReply* as described above. They are generally part of a loop that accepts messages, does the work, and sends back replies, like this fragment in C:

```
/* Code for allocating a request buffer */
do {
GetRequest(&port, &reqheader, &reqbuffer, reqbuflen);
/* Code for unmarshalling the request parameters */
/* Call the implementation routine */
/* Code for marshalling the reply parameters */
PutReply(&repheader, &repbuffer, repbuflen);
} while (1);
```

GetRequest blocks until a request comes in. *PutReply* blocks until the header and buffer parameters can be reused. A client sends a request and waits for a reply by calling

```
DoOperation(reqheader, reqbuffer, reqbuflen,
repheader, repbuffer, repbuflen);
```

All of this code is generated automatically by the AIL compiler from the object and operation descriptions given to it.

We call the process of finding the address of such a server thread *locating*. It works as follows. When a *DoOperation* call comes into a kernel, a check is made to see if the port in question is already known. If not, the kernel broadcasts a special *locate* packet onto the network asking if anyone out there has an outstanding *GetRequest* for the port in question. If one or more kernels have servers with outstanding *GetRequest*s they respond by sending their network addresses. The kernel doing the broadcasting records the (port, network address) pair in a cache for future use. Only if a server dies or migrates will another broadcast be needed.

When Amoeba is run over a wide area network, with huge numbers of machines, a slightly different scheme is used. Each server wishing to export its service sends a special message to all the domains in which it wants its service known. (A *domain* could be a company, campus, city, country or something else.) In each such domain, a dummy process, called a *server agent* is created. This process does a *GetRequest* using the server's port and then lies dormant until a request comes in, at which time it forwards the message to the server for processing. Note that a port is just a randomly chosen 48-bit number. It in no way identifies a particular domain, network, or machine.

3.4 Performance of Amoeba RPC

To measure the speed of the Amoeba RPC, we ran some timing tests. For example, we booted the Amoeba kernel on two 16.7 MHz Motorola 68020s and created a user process on each and let them communicate over a 10 Mbps Ethernet. For a message consisting of just a header (no data), the complete RPC took 1.4 msec. With 8K of data it took 13.1 msec, and with 30K it took 44.0 msec. The latter corresponds to a throughput of 5.4 megabits/sec, which is half the theoretical capacity of the Ethernet, and much higher than most other systems achieve. Five client-server pairs together can achieve a total throughput of 8.4 megabits per second, not counting Ethernet and Amoeba packet headers. More extensive measurements are given in van Renesse, van Staveren and Tanenbaum [1989]

Why did we use objects, capabilities, and RPC as the base for the design? Objects are a natural way to program. By encapsulating information, users are forced to pay attention to precise interfaces and irrelevant information is hidden from them. Capabilities are a clean and elegant way to name and protect objects. By using an encryption scheme for protecting them, we moved the capability management out of the kernel. RPC is an obvious way to implement the request/reply nature of performing operations on objects.

4 The Amoeba File System

Capabilities form the low-level naming mechanism of Amoeba, but they are very impractical for use by human beings. Therefore an extra level of mapping is provided from human-sensible hierarchical path names to capabilities. On Amoeba, a typical user has access to literally thousands of capabilities — of the user's own private objects, but also capabilities of public objects, such as the executables of commands, pool processors, data bases, public files, and so on.

It is perhaps feasible for a user to store his own private capabilities somewhere, but it is quite impossible for a system manager, or a project co-ordinator to hand out capabilities explicitly to every user who may access a shared public object. Public places are needed where users can find capabilities of shared objects, so that when a new object is made sharable, or when a sharable object changes, its capability need be put in only one place.

4.1 The Hierarchical Directory Structure

Hierarchical directory structures are ideal for implementing partially shared name spaces. Objects that are shared between the members of a project team can be stored in a directory that only team members have access to. By implementing directories as ordinary objects with a capability that is needed to use them, members of a group can be given access by giving them the capability of the directory, while others can be withheld access by not giving them the capability. A capability of a directory is thus a capability for many other capabilities.

To a first approximation, a directory is a set of (name, capability) pairs. The basic operations on directory objects are:

- lookup(*DirCap*, *ObjectName*)
- enter(*DirCap*, *ObjectName*, *ObjectCap*)
- delete(*DirCap*, *ObjectName*)

The first one looks up an object name in a directory and returns its capability. The other two enter and delete objects from directories. Since directories themselves are objects, a directory may contain capabilities for other directories, thus potentially allowing users to build an arbitrary graph structure.

Complex sharing can be achieved by making directories more sophisticated than we have just described. In reality, a directory is an $n + 1$-column table with ASCII names in column 0 and capabilities in columns 1 through n. A capability for a directory is really a capability for a specific column of a directory. Thus, for example, a user could arrange his directories with one column for himself, a second column for members of his group, and a third column for everyone else. This scheme can provide the same protection rules as UNIX , but obviously many other schemes are also possible.

The Directory Service can be set up so that whenever a new object is entered in a directory, the Directory Service first asks the service managing the object to make n replicas, potentially physically distributed for reliability. All the capabilities are then entered into the directory.

4.2 The Bullet Server

The Bullet Server is a highly unusual file server. It supports only three principal operations:

- ReadFile
- CreateFile
- DeleteFile

When a file is created, the user normally provides all the data at once, creating the file and getting back a capability for it. In most circumstances the user will immediately give the file a name and ask the Directory Server to enter the (name, capability) pair in some directory.

All files are *immutable*, that is, once created they cannot be changed. Notice that there is no *write* operation supported. Since files cannot change, the Directory Server can replicate them at its leisure for redundancy without fear that a file may change in the meanwhile.

Since the final file size is known when a file is created, files can, and are, stored contiguously, both on the disk and in the Bullet Server's cache. The administrative information for a file is then reduced to its origin and size plus some ownership data. The complete administrative table is loaded into the Bullet Server's memory when it is booted. When a *read* operation is done, the object number in the capability is used as an index into this table, and the file is read into the cache in a single (possibly multitrack) disk operation.

The Bullet file server can deliver large files from its cache, or consume large files into its cache at maximum RPC speeds, that is, at 677 kilobytes per second. Reading a 4 kilobyte file from the Bullet Server's cache by a remote client (over the Ethernet) takes 7 msec; a 1 megabyte file takes 1.6 sec. More detailed performance numbers and comparisons with other systems can be found in van Renesse, van Staveren and Tanenbaum [1989].

Although the Bullet Server wastes some space due to fragmentation, its enormous performance easily compensates for having to buy an 800M disk to store, say, 500M worth of data.

4.3 Atomicity

Ideally, names always refer to consistent objects and sets of names always refer to mutually consistent sets of objects. In practice, this is seldom the case and it is, in fact, not always necessary or desirable. But there are many cases where it is necessary to have consistency.

Atomic actions form a useful tool for achieving consistent updates to sets of objects. Protocols for atomic updates are well understood and it is possible to provide a toolkit which allows independently implemented services can collaborate in atomic updates of multiple objects managed by several services.

In Amoeba, a different approach to atomic updates has been chosen. The Directory Service, takes care of atomic updates by allowing the mapping of arbitrary sets of names onto arbitrary sets of capabilities to be changed atomically. The objects referred to by these capabilities, must to be immutable, either because the services that manage them refuse to change them (e.g., the Bullet Service) or because the users refrain from changing them.

The atomic transactions provided by the Directory Server are not particularly useful for dedicated transaction-processing applications (e.g., banking, or airline-reservation systems), but they are enormously useful in preventing the glitches that sometimes result from users using an application just when a new version is installed, or two people simultaneously updating a file resulting in one lost update.

4.4 Reliability

The Directory Server plays a crucial role in the system. Nearly every application depends on it for finding the capabilities it needs. If the Directory Server stops, everything else will come to a grinding halt as well. Thus the Directory Server must never stop.

The Directory Service replicates all its internal tables on multiple disks so that no single-site failure will bring it down. The techniques used to achieve this are essentially the same techniques used in fault-tolerant data base systems.

The Directory Server is not only relied on to be up and running; it is also trusted to work correctly and never divulge a capability to an entity that is not entitled to see it. Security is an important aspect of the reliability of the directory service.

Even a perfect design of the Directory Server may lead to unauthorized users catching glimpses of the data stored in it. Hardware diagnostic software, for example, has access to the Directory Server's disk storage. Bugs in the operating system kernel might allow users to read portions of the disk.

Directories may be encrypted in order to prevent bugs in the directory server, in the operating system or other idiosyncrasies from laying bare the confidential information stored in them. The encryption key may be exclusive-or'ed with a random number and the result may be stored alongside the directory, while the random number is put in the directory's capability. After giving the capability to the owner, the Directory Server itself can forget the random number. It only needs it when the directory has to be decrypted in order to carry out operations on it, and will always receive the random number in the capability which comes with every client's request.

Why did we design such an unconventional file system? Partly to achieve great speed and partly for simplicity in design and implementation. The use of immutable files (and some other objects) makes it possible to centralize the replication mechanism in one place — the Directory Server. Immutable files are also easy to cache, an important issue when Amoeba is run over wide area networks.

5 Process Management

Amoeba processes can have multiple threads of control. A process consists of a segmented virtual address space and one or more threads. Processes can be remotely created, destroyed, checkpointed, migrated and debugged.

On a uniprocessor, threads run in quasi-parallel; on a shared-memory multiprocessor, as many threads can run simultaneously as there are processors. Processes can not be split up over more than one machine.

Processes have explicit control over their address space. They can add new segments to their address space by *mapping them in* and remove segments by *mapping them out*. Besides virtual address and length, a capability can be specified in a map operation. This capability must belong to a file-like object which is read by the kernel to initialize the new segment. This allows processes to do mapped-file I/O.

When a segment is mapped out, it remains in memory, although no longer as part of any process' address space. The unmap operation returns a capability for the segment which can then be read and written like a file. One process can thus map a segment out and pass the capability to another process; the other process can then map the segment in again. If the processes are on different machines, the contents of the segment are copied (by one kernel doing read operations and the other kernel servicing them); on the same machine, the kernel can use shortcuts for the same effect.

A process is created by sending a *process descriptor* to a kernel in an *execute process* request. A process descriptor consists of four parts as shown in Figure 3. The host descriptor describes on what machine the process may run, e.g., its instruction set, extended instruction sets (when required), memory needs, etc., but also it can specify a class of machines, a group of machines or a particular machine. A kernel that does not match the host descriptor will refuse to execute the process.

Then come the capabilities. One is the capability of the process which every client that manipulates the process needs. Another is the capability of a *handler*, a service that deals with process exit, exceptions, signals and other anomalies of the process.

The memory map has an entry for each segment in the address space of the process to be. An entry gives virtual address, segment length, how the segment should be mapped (read

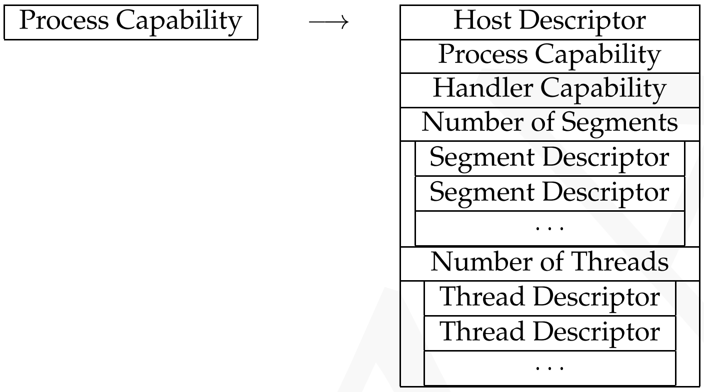

Figure 3: Layout of a process descriptor.

only, read/write, execute only, etc.), and the capability of a file or segment from which the new segment should be initialized.

The thread map describes the initial state of each of the threads in the new process, processor status word, program counter, stack pointer, stack base, register values, and system call state. This rather elaborate notion of thread state allows the use of process descriptors not only for the representation of executable files, but also for processes being migrated debugged or being checkpointed.

In most operating systems, system call state is large and complicated to represent outside an operating system kernel. In Amoeba, fortunately, there are very few system calls that can block in the kernel. The most complicated ones are those for communication: *DoOperation* and *GetRequest*.

Processes can be in two states, *running*, or *stunned*. In the stunned state, a process exists, but does not execute instructions. A process being debugged is in the stunned state, for example. The low-level communication protocols in the operating system kernel respond with 'this-process-is-stunned' messages to attempts to communicate with the process. The sending kernel will keep trying to communicate until the process becomes running again or until it is killed. Thus, communication with a process being interactively debugged continues to work.

A running process can be stunned by a stun request directed to it from the outside world (this requires the stunner to have the capability of the process as evidence of ownership), or by an uncaught exception. When the process becomes stunned, the kernel sends its state in a process descriptor to a *handler* whose identity is a capability which is part of the process' state. After examining the process descriptor, and possibly modifying it or the stunned process' memory, the handler can reply either with a *resume* or *kill* command.

Debugging processes is done with this mechanism. The debugger takes the role of the handler. Migration is also done through stunning. First, the candidate process is stunned; then, the handler gives the process descriptor to the new host. The new host fetches memory contents from the old host in a series of file read requests, starts the process and returns the capability of the new process to the handler. Finally, the handler returns a *kill* reply to the old host. Processes communicating with a process being migrated will receive 'process-is-stunned' replies to their attempts until the process on the old host is killed. Then they will get a 'process-not-here' reaction. After locating the process again, communication will resume with the process on the new host.

The mechanism allows command interpreters to cache process descriptors of the programs they start and it allows kernels to cache code segments of the processes they run. Combined, these caching techniques make process start-up times very short.

Our process management mechanisms are unusual, but they are intended for an unusual environment: one where remote execution is the normal case and local execution is the exception. The boundary conditions for our design were the creation of a few simple mechanisms that allowed us to do process execution, migration, debugging and checkpointing in such a way that a very efficient implementation is possible.

6 UNIX Emulation

Amoeba is a new operating system with a system interface that is quite different from that of the popular operating systems of today. Since we had no intention of writing hundreds of utility programs for Amoeba from scratch, it was quickly decided to write a UNIX emulation package, to allow most UNIX utilities to work on Amoeba, sometimes with small changes. Binary compatibility was considered as a possibility, but was rejected for an initial emulation package on grounds that it is more complicated and less useful (first, one has to choose a very particular version of Unix; second, one usually has binaries for only one machine architecture, while sources can be compiled for any machine archtecture; and, third, binary emulation is bound to be slow). Experiments with binary emulation have, however, shown that it is quite feasible to provide it in the future.

The emulation facility started out as a library of UNIX routines that have the standard UNIX interface and semantics, but do their work by calling the Bullet Server, the Directory Server and the Amoeba process management facilities. The system calls implemented initially were those for file I/O (*open, close, dup, read, write, lseek*) and a few of the *ioctl* calls for ttys. These were very easy to implement under Amoeba (about two week's work) and were enough to get a surprising number of UNIX utilities to run.

Subsequently, a *Session* server was developed to allocate UNIX PIDs, PPIDs, and assist in the handling of system calls involving them (*fork, exec, signal, kill*). The Session Server is also used for dealing with UNIX *pipes*. With the help of the Session Server many other UNIX utilities are now usable on Amoeba.

Currently, about 100 utilities have been made to run on Amoeba without any changes to the source code. The Bourne shell needed a two-line modification because of the extraordinary way it allocates memory. We have not attempted to port some of the more esoteric UNIX programs; In some cases, we cannot even determine what they do. Work is in progress to make our UNIX interface compatible with the emerging standards (e.g., IEEE POSIX).

The X window system has been ported to Amoeba and supports the use of both TCP/IP and Amoeba RPC, so that an X client on Amoeba can converse with an X server on Amoeba and vice versa.

We have found that the availability of the UNIX utilities has made the transition to Amoeba much easier. Slowly, however, many of the UNIX utilities will be replaced by utilities that are better adapted to the Amoeba distributed environment. Our new parallel *make* is an obvious example.

Why did we emulate UNIX in a library instead of making the system binary compatible? Because any system that is binary compatible with UNIX cannot be much of a step forward beyond the ideas of the early 1970s. We wanted to design a new system from the ground up for the 1990s. If the UNIX designers had constrained themselves to be binary compatible with the then-popular RT-11 operating system, it would not be now where it is.

7 Conclusions

We are pleased with most of the design decisions of the Amoeba project. The decision, especially, to design a distributed operating system without attempting to restrict ourselves to existing operating systems or operating system interfaces has been a good one. UNIX is an excellent operating system, but it is not a distributed one and was not designed as such. We do not believe we would have made such a balanced design had we decided to build a distributed system with a UNIX interface.

In spite of our design-independence from UNIX , we found it remarkably easy to port all the UNIX software we wanted to use to Amoeba. The programs that are hard to port are mostly those we have no need for in Amoeba anyway (programs for network access and for system maintenance and management, for example).

The use of objects and capabilities has also given us some very important advantages. When a service is being designed, the protection of its objects usually does not require any though; the use of capabilities automatically provides enough of a protection mechanism. It also gave us a very uniform and decentralized object-naming and -access mechanism.

The decision not to build on top of an existing operating system, but to build directly on the hardware has been absolutely essential to the success of Amoeba. One of the primary goals of the project was to design and build a high-performance system and this can hardly be done on top of another system. As far as we can tell, only systems with custom-built hardware or special microcode can outperform Amoeba's RPC and file system on comparable hardware.

The Amoeba kernel is small and simple. It implements only a few operations for process management, and interprocess communication, but they are versatile and easy to use. The performance of its interprocess communication has already been mentioned. The kernel is easy to port between hardware platforms. It now runs on VAX and Motorola 68020 and 68030 processors, and is currently being ported to the Intel 80386. Amoeba is now available. For information about how to obtain a copy, please contact the authors.

8 References

Accetta et al. [1986]
M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, Mach: A New Kernel Foundation for UNIX Development, *Proceedings of the Summer Usenix Conference*, Atlanta, GA, July 1986.

Birrell and Nelson [1984]
A. D. Birrell and B. J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems* **2**, February 1984, 39–59.

Cheriton [1988]
D. R. Cheriton, The V Distributed System, *Communications of the ACM* **31**, March 1988, 314–333.

Mullender and Tanenbaum [1986]
S. J. Mullender and A. S. Tanenbaum, The Design of a Capability-Based Distributed Operating System, *The Computer Journal* **29**(**4**), 1986, 289–300.

Needham and Herbert [1982]
R. M. Needham and A. J. Herbert, *The Cambridge Distributed Computing System*, Addison-Wesley, Reading, MA, 1982.

van Renesse, van Staveren and Tanenbaum [1989]
R. van Renesse, J. M. van Staveren and A. S. Tanenbaum, Performance of the Amoeba Distributed Operating System, *Software–Practice and Experience* **19**, March 1989, 223–234.

van Rossum [1989]
G. van Rossum, AIL – A Class-Oriented Stub Generator for Amoeba, *Workshop on Progress in Distributed Operating Systems and Distributed Systems Management* **433**, 1989.

Rozier et al. [1988]
M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser, *CHORUS Distributed Operating Systems*, Chorus Systèmes, Report CS/TR-88-7.6, Paris, November 1988.

Walker et al. [1983]
B. Walker, G. Popek, R. English, C. Kline and G. Thiel, The LOCUS Distributed Operating System, *Proceedings of the Ninth Symposium on Operating Systems Principles*, Bretton Woods, NH, October 1983, 49–70, In *ACM Operating Systems Review* **17**(5).