Amoeba

A Distributed Operating System for the 1990s

Sape J. Mullender and Guido van Rossum
Centre for Mathematics and Computer Science

Andrew S. Tanenbaum, Robbert van Renesse, and
Hans van Staveren
Free University of Amsterdam

In the next decade, computer prices will drop so low that 10, 20, or perhaps 100 powerful microprocessors *per user* will be feasible. All this computing power will have to be organized in a simple, efficient, and fault-tolerant system that is easy to use. The basic problem with current networks of PCs and workstations is that they are not transparent; that is, users are aware of the other machines. The user logs into one machine and uses that machine only, until doing a remote login to another machine. Few if any programs take advantage of multiple CPUs, even when all are idle.

We envision a system for the 1990s that will appear to users as a single, 1970s centralized time-sharing system. Users will not know which processors their jobs are using (or even how many), where their files are stored (or how many replicated copies are maintained to provide high availability), or how processes and machines are communicating. All resources will be managed completely and automatically by a distributed operating system.

Few such systems have been designed, and even fewer have been implemented. Fewer still are actually used by anyone yet. An early distributed system was the Cambridge system.[1] Later systems were Locus,[2] Mach,[3] the V-Kernel,[4] and Chorus.[5]

The Amoeba distributed operating system appears to users as a centralized system, but it has the speed, fault tolerance, security safeguards, and flexibility required for the 1990s.

Here we describe Amoeba, a distributed system developed at the Free University and the Centre for Mathematics and Computer Science in Amsterdam. Amoeba combines high availability, parallelism, and scalability with simplicity and high performance.

Although distributed systems are necessarily more complicated than centralized systems and tend to be much slower, we have worked hard to achieve extremely high performance: Amoeba is already one of the fastest distributed systems (on its class of hardware) reported so far, and future versions will be even faster. With the current implementation, a remote procedure call can be performed in 1.4 ms on Sun-3/50 class machines. The file server can deliver data continuously at 677 Kbytes per second.

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

0018-9162/90/0500-0044$01.00 © 1990 IEEE

A process is a segmented address space shared by one or more threads of control. Processes can be created, managed, and debugged remotely. Operations on objects are implemented using remote procedure calls.

Amoeba has a unique, fast file system split into two parts: The bullet service stores immutable files contiguously on the disk; the directory service gives capabilities symbolic names and handles replication and atomicity, eliminating the need for a separate transaction management system.

To bridge the gap with existing systems, Amoeba has a Unix emulation facility consisting of a library of Unix system call routines that make calls to the various Amoeba server processes.

Most classical distributed systems literature describes work on parts of or aspects of distributed systems: distributed file servers, distributed name servers, distributed transaction systems, and so on. Here we discuss the whole system, covering most of the traditional operating system design issues, including communication, protection, the file system, and process management. We explain not only what we did but also why we did it.

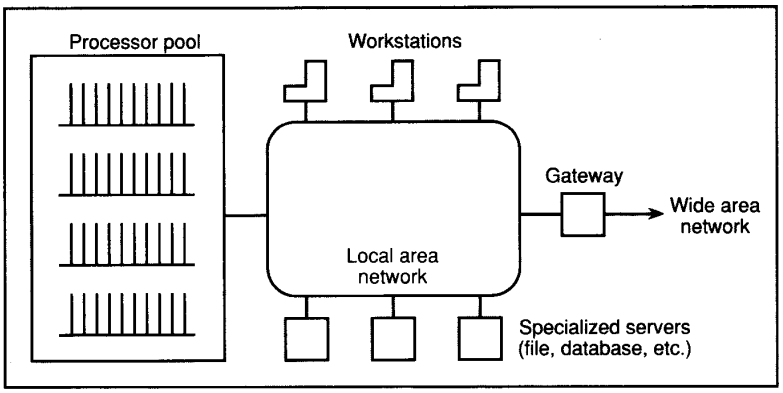


Figure 1. Four components of the Amoeba architecture.

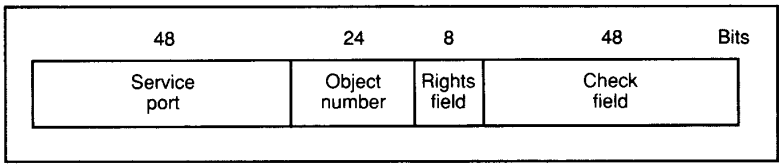


Figure 2. The structure of a capability. The service port identifies the service that manages the object. The object number specifies the object (for example, which file). The rights field determines which operations are permitted. The check field provides cryptographic protection to keep users from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

Overview of Amoeba

The Amoeba project[6] has been under way for nearly 10 years and has seen numerous system redesigns and reimplementations as design flaws became glaringly apparent. This article describes the Amoeba 4.0 system, released in 1990.

Hardware architecture. As Figure 1 shows, the Amoeba hardware consists of four components: workstations, pool processors, specialized servers, and gateways. The workstations execute only processes that require intense user interaction — for example, window managers, command interpreters, editors, and CAD/CAM graphical front ends. Most applications, however, do not interact much with the user and are run elsewhere.

Amoeba's processor pool provides most of the computing power. Typically it consists of many single-board computers, each with several megabytes of private memory and a network interface. The Free University, for example, has 48 such machines. A pile of diskless, terminalless workstations can also be used as a processor pool.

When a user has an application to run — for example, building 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 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.

The processor pool 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 processors can be added as the user population grows. When a few processors crash, some jobs may have to be restarted and the computing capacity is temporarily lowered, but otherwise the system continues normally, providing a degree of fault tolerance.

Specialized servers, the third system component, are machines for running dedicated processes with unusual resource demands. For example, it is best to run file servers on machines that have disks.

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

Why did we choose this architecture instead of the traditional workstation model? As it becomes possible to give each user 10 to 100 processors, centralizing the computing power will allow incremental growth, fault tolerance, and the ability for a large job to obtain a large amount of computing power temporarily. Current systems have file servers, so why not let them have computer servers as well?

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 Figure 2 shows. Capabilities have the set

Amoeba Interface Language

Interfaces for object manipulation are specified in a notation called the Amoeba Interface Language.[1] AIL resembles the notation for procedure headers in C, but it has some extra syntax for automatic generation of client and server stubs. The Amoeba class for standard manipulations on filelike objects, for instance, could be specified as follows:

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

        const    BIO_SIZE = 30000;

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

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

The names of the operations, bio_read and bio_write, must be globally unique. They conventionally start with an abbreviation of the name of their class. The first parameter, indicated by an asterisk, is always a capability of the object to which the operation refers. The other parameters are labeled "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 bio_read, the maximum size is provided by the caller. In bio_read, 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 bio_read 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 bio_write, it is the in value of bytes.

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

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

Reference

1. G. van Rossum, "AIL — A Class-Oriented Stub Generator for Amoeba," *Proc. Workshop on Experience with Distributed Systems*, Springer-Verlag, Berlin, to be published in 1990.

of operations that the holder may carry out on the object coded into them, and they contain enough redundancy and cryptographic protection to make guessing an object's capability infeasible. Keeping capabilities secret by embedding them in a huge address space is the key to protection in Amoeba. Because of the cryptographic protection, capabilities can be managed outside the kernel, by user processes themselves.

Objects are implemented by the 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 remote procedure call system guarantees that requests and replies are delivered only once, and only to authorized processes.

Although at the system level objects are identified by their (binary) capabilities, at the level where most people program and work, objects are named using a symbolic hierarchical naming scheme. The directory service maintains a mapping of ASCII path names onto capabilities and has mechanisms for performing atomic operations on arbitrary collections of name-to-capability mappings.

Amoeba has already gone through several generations of file systems. Currently, one file server is used almost to the exclusion of all others. The bullet service (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.

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 a later section.

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 for these services. By putting as much as possible in user space, we have achieved a flexible system without sacrificing performance.

In the Amoeba design, concessions to existing operating systems and software were carefully avoided. But a Unix emulation service was developed to run existing software on Amoeba.

Communication

Amoeba's conceptual model is that of a client thread (thread of control or lightweight process) performing operations on objects. For example, a common operation on a file object is reading data from it. Operations are implemented by making

Transport interface

The transport interface for the server consists of the calls get_request and send_reply, as described in the section on communication. They are generally part of a loop that accepts messages, does the work, and sends back replies, as in this C fragment:

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

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

```
do_operation(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.

remote procedure calls.[7] A client sends a request message to the service that manages the object. A server thread accepts the message, carries out the request, and sends the client a reply. To increase performance and fault tolerance, multiple server processes often jointly manage a collection of similar objects to provide a service.

Remote procedure calls. The kernel provides three basic system calls to user processes: do_operation, get_request, and send_reply. The first 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 is used by servers to announce their willingness to accept messages addressed to a specific port. Servers use the third call to send replies back. All communication in Amoeba takes this form: First a client sends a request to a server; then the server accepts the request, does the work, and sends back the reply.

No doubt systems programmers would be content with only these three system calls, but for most applications programmers they are far too primitive. Therefore a more user-oriented interface has been built on top of the 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 can contain 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 it. The mechanism also allows all servers manipulating objects with filelike properties to inherit the same interface for low-level file I/O (read, write, append — see sidebar on Amoeba Interface Language). The mechanism resembles the filelike 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, and so forth).

Remote procedure call transport. The Amoeba Interface Language compiler generates code to marshal or unmarshal the parameters of remote procedure calls into and out of message buffers and then call the Amoeba transport mechanism for delivery of request and reply messages (see sidebar on the transport interface). Messages consist of a header and a buffer. The header has a fixed format and contains addressing information (including the capability of the object that the remote procedure call refers to), an operation code that 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 setup, 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.

Locating objects. Before a request for an operation on an object can be delivered to a server thread that manages the object, such a thread must be located. All capabilities contain a service port field, which identifies the service that manages the object referred to by the capability. When a server thread makes a get_request call, it provides its service port to the kernel, which records it in an internal table. When a client thread calls do_operation, the kernel's job is to find a server thread with an outstanding get_request that matches the port in the capability provided by the client.

We call the process of finding the address of such a server thread *locating*. It works as follows: When a do_operation 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 has an outstanding get_request for the port in question. If one or more kernels have servers with outstanding get_requests, they respond by sending their network addresses. The kernel doing the broadcasting records the port/network address pair in a cache for future use.

Secure communication

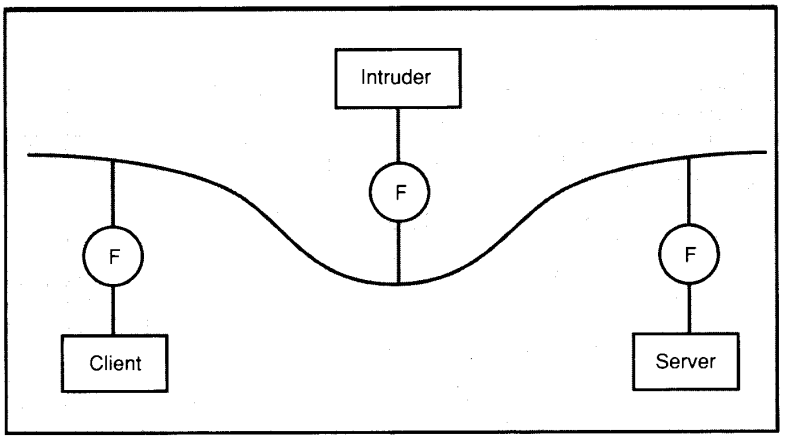
Client requests, addressed using an object's capability, are delivered to one of the servers with outstanding get_request calls on the capability's port. Ports consist of large, 48-bit numbers known only to the server processes that make up the service and to the server's clients. For a public service such as the file system, the port will be known to all users. The ports used by an ordinary user process will, in general, be kept secret. Knowledge of a port is taken by the system as prima facie evidence that the sender has a right to communicate with the service. Of course, the service is not required to carry out work for clients just because they know the port. For example, the file server will refuse to read or write files for clients lacking appropriate file capabilities. Thus Amoeba has two levels of protection: ports for protecting access to servers and capabilities for protecting access to individual objects.

Although the port mechanism conveniently handles partial authentication of clients ("if you know the port, you may at least talk to the service"), it does not authenticate servers. How do we ensure that malicious users do not make get_request calls on the file server's port and try to impersonate the file server to the rest of the system?

One approach is to have all ports manipulated by kernels that are presumed to be trustworthy and are supposed to know who may listen on which port. We have rejected this strategy because on some machines — for example, PCs — users might be able to tamper with the operating system kernel. Also, we believe that by making the kernel as small as possible, we can enhance system reliability as a whole. Therefore, we have chosen a different solution that can be implemented in either hardware or software.

In the hardware solution we place a small interface box, a function box or F-box, between each processor module and the network. The most logical place is on the VLSI chip used to interface to the network. Alternatively, it can be put on a small printed circuit board inside the wall socket through which PCs attach to the network. Where the processors have user mode and kernel mode, and the operating systems can be trusted, it can be put into the operating system. This is the solution in the current Amoeba implementation.

In the software solution we build the F-box with cryptographic algorithms, giving the same functional effect as the hardware box. In both cases we assume that all messages entering and leaving every processor undergo a simple transformation that users cannot bypass.



Clients, servers, intruders, and F-boxes.

The transformation works like this. Each port is really a pair of ports, P and G, related by $P = F(G)$, where F is a (publicly known) one-way function' performed by the F-box. The one-way function has the property that given G it is a straightforward computation to find P, but that given P, finding G is not feasible.

Using the one-way F-box, servers can be authenticated simply, as the figure illustrates. Each server chooses a get-port G and computes the corresponding put-port P. The get-port is kept secret; the put-port is distributed to potential clients or, in the case of public servers, is published. When the server is ready to accept client requests, it does a get_request (G, ...). The F-box then computes $P = F(G)$ and waits for messages containing P to arrive. When one arrives, it is given to the server process. To send a message to the server, the client merely does do_operation (P, ...), which sends a message containing P in a header field to the server. The F-box on the sender's side does not perform any transformation on the P field of the outgoing message.

Now consider the system from an intruder's point of view. To impersonate a server, the intruder must do get_request (G, ...). However, G is a well-kept secret and is never transmitted on the network. Since we have assumed that G cannot be deduced from P (the one-way property of F) and that the F-box cannot be circumvented, intruders cannot intercept messages not intended for them. An intruder doing get_request (P, ...) will simply cause his F-box to listen to the (useless) port $F(P)$. Replies from the server to the client are protected the same way, only with the client picking a get-port for the reply, say G', and including $P' = F(G')$ in the request message.

The F-box makes it easy to implement digital signatures for further authentication, if that is desired. Each client chooses a random signature S and publishes $F(S)$. The F-box must be designed to work as follows. Each message presented to the F-box for transmission contains three special header fields: destination (P), reply (G'), and signature (S). The F-box applies the one-way function to the second and third of these, transmitting the three ports as P, $F(G')$, and $F(S)$, respectively. The first is used by the receiver's F-box to admit only those messages for which the corresponding get has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the signature will know what number to put in the third field to ensure that the publicly known $F(S)$ comes out.

The F-box implements security and protection simply, but gives operating system designers considerable latitude in choosing policies. The mechanism is flexible and general, so putting it into hardware should not preclude yet-to-be-designed operating systems.

Reference

1. M.V. Wilkes, *Time-Sharing Computer Systems*, 2nd ed., Elsevier, New York, 1969, pp. 129-132.

Table 1. The delay in milliseconds and the bandwidth in Kbytes per second for remote procedure calls between user processes in three common cases with three different systems. For local RPCs the client and server run on the same processor. The Unix driver implements Amoeba RPCs under Sun Unix.

	Delay (ms)			Bandwidth (Kbytes per second)		
	Case 1 (4 bytes)	Case 2 (8 Kbytes)	Case 3 (30 Kbytes)	Case 1 (4 bytes)	Case 2 (8 Kbytes)	Case 3 (30 Kbytes)
Native Amoeba local	0.8	2.5	7.1	5.0	3,277	4,255
Native Amoeba remote	1.4	13.1	44.0	2.9	625	677
Unix driver local	4.5	10.0	32.0	0.9	819	938
Unix driver remote	7.0	36.4	134.0	0.6	225	224
Sun RPC local	10.4	23.6	imposs.	0.4	347	imposs.
Sun RPC remote	12.2	40.6	imposs.	0.3	202	imposs.

Another broadcast is needed only if a server dies or migrates.

When Amoeba is run over a wide area network with a huge number of machines, a slightly different scheme is used. Each server wishing to export its service sends a special message to all domains where it wants its service known. (A domain could be a company, campus, city, or country.) In each domain a dummy process called a server agent is created. This process does a get_request using the server's port and then lies dormant until a request comes in. Then it forwards the message to the server for processing. Note that a port is just a randomly chosen 48-bit number. It does not identify a particular domain, network, or machine (see sidebar on secure communication).

Performance. We measured the speed of the Amoeba remote procedure call with some timing tests. For example, we booted the Amoeba kernel on two 16.7-megahertz Motorola MC68020s, created a user process on each, and let them communicate over a 10-megabit-per-second Ethernet. For a message consisting of just a header (no data), the complete remote procedure call (RPC) took 1.4 ms. With 8 Kbytes of data it took 13.1 ms, and with 30 Kbytes it took 44.0 ms. The latter corresponds to a throughput of 5.4 megabits per second, which is half the theoretical capacity of the Ethernet and much faster than the speeds 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. Table 1 shows the speeds and throughput of local communication (communication between processes on the same machine) and remote communication (communication over Ethernet between processes on different machines). Remote operations were carried out with requests containing 4 bytes, 8 Kbytes, 30 Kbytes, and empty replies. Three RPC implementations were measured: RPCs on native Amoeba, the same Amoeba protocol used from a driver under Sun Unix, and Sun's own RPCs.

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

File system

Capabilities form Amoeba's low-level naming mechanism, but they are hard for people to use. Therefore an extra level of mapping is provided from symbolic hierarchical path names to capabilities. A typical user has access to literally thousands of capabilities — those of the user's own private objects, but also capabilities of public objects, such as the executables of commands, pool processors, databases, and public files.

While a user could perhaps store his own private capabilities somewhere, a system manager or project coordinator cannot 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 shareable, or when a shareable object changes, its capability need be put in only one place.

Hierarchical directory structure. Hierarchical directory structures are ideal for implementing partially shared name spaces. Objects shared among members of a project team can be stored in a directory that only team members have access to. When directories are implemented as ordinary objects with a capability that is needed to use them, group members can be given access by giving them the capability of the directory, while others are denied access by withholding the capability. A directory capability 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, enter, and delete. The first operation looks up an object name in a directory and returns its capability. The other two operations enter and delete objects from directo-

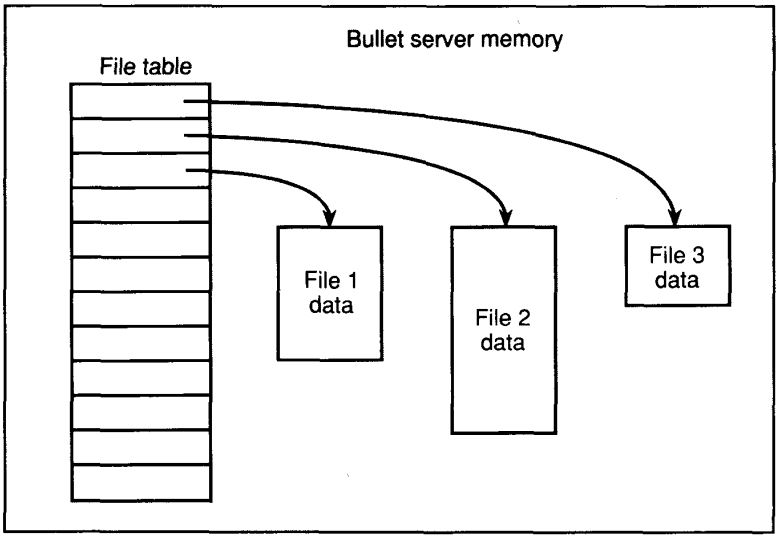


Figure 3. Bullet server file representation.

ries. Since directories themselves are objects, a directory can contain capabilities for other directories, thus 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, users could arrange their directories with one column for themselves, a second column for members of their group, and a third column for everyone else. This scheme provides the same protection rules as Unix, but obviously many other schemes are 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, which can be physically distributed for reliability. All the capabilities are then entered into the directory.

Bullet service. The bullet service is a highly unusual file server. Each bullet server supports only three principal operations: read file, create file, and delete file.

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 service to enter the name/capability pair in some directory.

All files are immutable; once created, they cannot be changed. No write operation is supported. Since files cannot change, the directory service can replicate them at its leisure for redundancy.

Since the final file size is known when a file is created, files can be and are stored contiguously, both on the disk and in bullet servers' caches, as Figure 3 illustrates. Administrative information for a file is thus 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. For a read operation 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 service can deliver large files from its cache or accept large files into its cache at maximum RPC speeds, that is, at 677 Kbytes per second. A remote client can read a 4-Kbyte file from a bullet server's cache (over Ethernet) in 7 ms; a 1-Mbyte file takes 1.6 seconds.[8]

Although the bullet service wastes some space because of fragmentation, its performance easily compensates for having to buy an 800-Mbyte disk to store, say, 500 Mbytes of data.

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 is, in fact, not always necessary or desirable. But in many cases consistency is necessary.

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

For Amoeba we chose a different approach. The directory service handles atomic updates by allowing atomic changes in the mapping of arbitrary name sets onto arbitrary capability sets. The objects referred to by these capabilities must be immutable, either because the services that manage them refuse to change them (for example, the bullet service) or because the users refrain from changing them.

The atomic transactions provided by the directory service are not particularly useful for dedicated transaction-processing applications (for example, banking and airline reservation systems), but they do prevent the glitches that sometimes result when people use an application just as a new version is installed, or the lost update that results when two people simultaneously update a file.

Reliability and security. The directory service is crucial to the system: Nearly every application depends on it for finding the capabilities it needs. If the directory service stops, everything else will come to a halt as well. So that no single-site failure can bring it down, the directory service uses techniques similar to those used in fault-tolerant database systems to replicate all its internal tables on multiple disks.

The directory service must also work correctly and should never divulge a capability to an entity not entitled to see it. Yet even a perfectly designed directory service might allow unauthorized users to catch glimpses of data. 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 can be encrypted so that bugs in the directory server and the operating system (or other idiosyncrasies) will not reveal confidential information. The encryption key can be exclusive-ORed with a random number and the result stored alongside the directory, while the random

number is put in the directory's capability. After giving the capability to the owner, the directory service itself can forget the random number. It needs the number only when the directory has to be decrypted to carry out operations on the directory, and will always receive the number in the capability that 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) allows the replication mechanism to be centralized in the directory service. Immutable files are also easy to cache (because a cached immutable file can never become stale), an important issue when Amoeba is run over wide area networks.

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

Processes have explicit control over their address space. They can add new segments to it by mapping them in and remove segments by mapping them out. Along with virtual address and length, a capability can be specified in a map operation. This capability must belong to a filelike 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 the address space of any process. 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. Figure 4 shows the four parts of a process descriptor. The host descriptor describes the machine on which the process can run — for example, its instruction set, extended instruction sets (when required), and memory needs — but it can also 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.

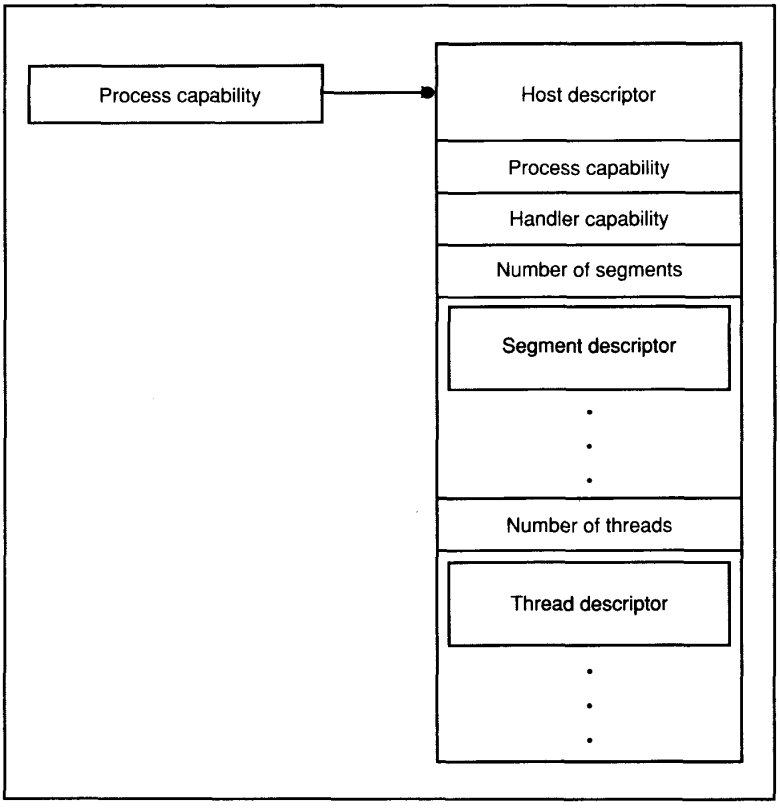


Figure 4. Layout of a process descriptor.

The capabilities are next. One is the process capability that every client manipulating the process needs. Another is the capability of a handler, a service that deals with process exits, 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 only, read/write, execute only, and so forth), 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 thread in the new process: the processor status word, the program counter, the stack pointer, the stack base, the register values, and the system call state. This rather elaborate notion of thread state allows process descriptors to be used not only for the representation of executable files, but also for processes being migrated, debugged, or 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 for communication: do_operation and get_request.

Processes can be in two states: running or stunned. A stunned process — for example, a process being debugged — exists but does not execute instructions. 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 is running again or until it is killed. Thus, communication continues with a process being interactively debugged.

A running process can be stunned by a stun request from the outside world (the stunner must have the process capability 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 that belongs to the process' state. After examining the process descriptor, and possibly modifying it or the stunned process' memory, the handler can reply with either a resume or a kill command.

Debugging and migration are done through stunning. The debugger takes the role of the handler. For migration, 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. They will then get a "process not here" reaction. After they find the process on its new host, communication will resume.

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

Our process management mechanisms are unusual, but they are intended for an unusual environment, one where remote execution is normal and local execution is the exception. The boundary conditions for our design were a few simple mechanisms that allowed us to implement process execution, migration, debugging, and checkpointing efficiently.

Unix emulation

Amoeba's system interface is quite different from those of today's popular operating systems. We did not want to write hundreds of utility programs for Amoeba from scratch, so we quickly decided to write a Unix emulation package to allow most Unix utilities to work on Amoeba, sometimes with small changes. We considered binary compatibility but rejected it for an initial emulation package because binary compatibility is more complicated and less useful. (First, we would have to choose a particular version of Unix; second, binaries usually work for only one machine architecture, while sources can be compiled for any machine architecture; and third, binary emulation is bound to be slow.)

Our emulation facility started as a library of Unix routines that have the standard Unix interface and semantics but do their work by calling the bullet service, the directory service, 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 weeks' work) and were enough to run a surprising number of Unix utilities.

Next a session server was developed to allocate Unix PIDs and PPIDs, and to assist in the handling of system calls involving them (for example, fork, exec, signal, kill). The session server is also used for dealing with Unix pipes and allows many other Unix utilities to run on Amoeba. Users each start one session server alongside their login shell.

About 150 utilities now run on Amoeba without any changes to the source code. We have not attempted to port some of the more esoteric Unix programs, but we are working to make our Unix interface compatible with some emerging standards (for example, IEEE Posix).

The X Window System has been ported to Amoeba and supports both TCP/IP and Amoeba RPCs, so an X client on Amoeba can converse with an X server on Amoeba and vice versa.

The Unix utilities have eased the transition to Amoeba. Gradually, however, many of them will be replaced by utilities better adapted to the Amoeba distributed environment. Our new parallel Make is an obvious example.

If we had designed a system that was binary compatible with Unix, it would not have been much of a step beyond the ideas of the early 1970s. We wanted a new system for the 1990s, designed from the ground up. If the Unix designers had constrained themselves to being binary compatible with the then-popular RT-11 operating system, Unix would not be where it is now.

Among the design decisions for Amoeba we have been most pleased with is our determination not to restrict ourselves to existing operating systems or operating system interfaces. Unix is an excellent operating system, but it was not designed for distributed systems. We could not have made such a balanced design with a Unix interface. Nevertheless, we found it remarkably easy to port to Amoeba all the Unix software we wanted to use. Programs that are hard to port are mostly for operations that Amoeba handles in other ways (network access and system maintenance and management, for example).

Amoeba's use of objects and capabilities means that when we design a service we need not worry about the protection of its objects. The capabilities mechanism automatically provides enough protection. The system also provides a very uniform and decentralized object-naming and object-access mechanism.

Building directly on the hardware instead of on an existing operating system has been absolutely essential to Amoeba's success. A primary goal 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 remote procedure calls 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 kernel is easy to port between hardware platforms. Amoeba now runs on VAXs and on Motorola MC68020 and MC68030 processors, and is currently being ported to the Intel 80386. ■

Acknowledgments

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

References

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

2. B. Walker et al., "The LOCUS Distributed Operating System," *Proc. Ninth Symp. Operating System Principles*, ACM, *Operating Systems Review*, Vol. 17, No. 5, 1983, pp. 49-70.

3. M. Accetta et al., "Mach: New Kernel Foundation for UNIX Development," *Proc. Summer Usenix Conference*, Usenix, Sunset Beach, Calif., 1986.

4. D.R. Cheriton, "The V Distributed System," *Comm. ACM*, Vol. 31, No. 3, Mar. 1988, pp. 314-333.

5. M. Rozier et al., "CHORUS Distributed Operating Systems," Report CS/Tech. Report-88-7.6, Chorus Systems, Paris, 1988.

6. S.J. Mullender and A.S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," *Computer J.*, Vol. 29, No. 4, Mar. 1986, pp. 289-300.

7. A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 39-59.

8. R. van Renesse, J.M. van Staveren, and A.S. Tanenbaum, "Performance of the Amoeba Distributed Operating System," *Software — Practice and Experience*, Vol. 19, No. 3, Mar. 1989, pp. 223-234.

Sape J. Mullender heads the distributed systems and computer networks research group at the Centre for Mathematics and Computer Science in Amsterdam. He has been a visiting scientist at DEC's Systems Research Center in California and a visiting research fellow at Cambridge University. Mullender's research interests include high-performance communication in distributed systems and the design of scalable fault-tolerant distributed file servers. He is also concerned with organization and protection in distributed systems that can span a continent.

Mullender is vice chairman and conference coordinator of the ACM Special Interest Group on Operating Systems. He received his PhD at the Free University in Amsterdam. He is a member of ACM and the IEEE.

Guido van Rossum is a research assistant at the Centre for Mathematics and Computer Science in Amsterdam. Since 1987 he has been with the Amoeba project, working on an RPC interface specification language, a Unix emulation facility, user interface issues, and system integration. Earlier he worked on the ABC Programming Language project.

Van Rossum studied mathematics and computer science at the University of Amsterdam and received a master's degree in 1982.

Andrew S. Tanenbaum is a professor of computer science at the Free University in Amsterdam. His research interests include distributed operating systems, programming languages, and compilers. He is the author of the Minix operating system, a principal designer of the Amsterdam Compiler Kit, and a chief architect of the Amoeba distributed operating system.

Tanenbaum received his BS from MIT and his PhD from the University of California, Berkeley. He is a member of ACM, the IEEE Computer Society, and Sigma Xi.

Robbert van Renesse is a researcher in the Computer Science Department at Cornell University on a grant from the Netherlands Organization for Scientific Research. He is working on the management of distributed systems to improve their robustness, performance, and scalability.

Van Renesse received his PhD in computer science in 1989 from the Free University of Amsterdam.

Hans van Staveren is one of the implementers of the Amoeba distributed operating system, working primarily on network protocols and kernel efficiency. Earlier he spent four years researching code generation in the framework of the Amsterdam Compiler Kit.

Van Staveren graduated from the Free University in Amsterdam in 1980.

The authors can be contacted at the Centre for Mathematics and Computer Science, PO Box 4079, 1009 AB Amsterdam, the Netherlands.