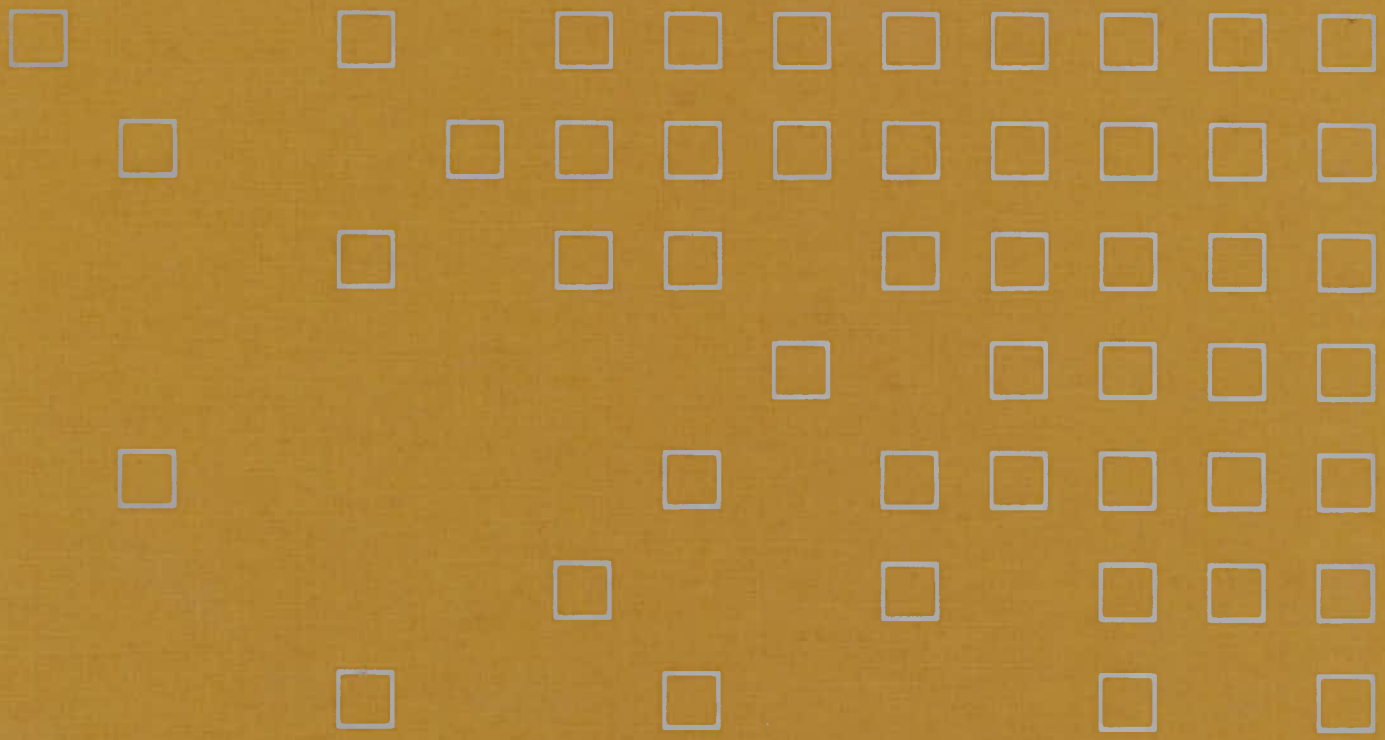

Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

S.J. Mullender, G. van Rossum, A.S. Tanenbaum,
R. van Renesse, J.M. van Staveren

Amoeba—High-performance distributed computing

Computer Science/Department of Algorithmics & Architecture Report CS-R8937 October

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

S.J. Mullender, G. van Rossum, A.S. Tanenbaum,
R. van Renesse, J.M. van Staveren

Amoeba—High-performance distributed computing

Computer Science/Department of Algorithmics & Architecture Report CS-R8937 October

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Amoeba — High-Performance Distributed Computing

Sape J. Mullender
Guido van Rossum
Centrum voor Wiskunde en Informatica

Andrew S. Tanenbaum
Robbert van Renesse
Hans van Staveren
Vrije Universiteit

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 was to combine the research themes of distributed systems, such as high availability, good parallelism and scalability with simplicity and high performance. Distributed systems are necessarily more complicated than centralized systems, so many have a tendency to be much slower. Amoeba was always designed to be used, so it was deemed unacceptable to build a system that would be inherently slower than its centralized counterparts.

Amoeba is an object-based distributed system using capabilities for protection and naming. Objects are managed by a service and objects are named using capabilities chosen randomly from a sparse name space.

Processes consist of a segmented address space and one or more threads of control. Processes can be created, managed and debugged remotely and processes may migrate at any point during their execution. Remote operations are used for interprocess communication.

The principal file system offers an immutable-file interface with caching in the file server's main memory. The directory server maps path names to capabilities and allows atomic update of sets of mappings, obviating a separate transaction management system.

Most applications written for Unix can be run on Amoeba without change with the help of an emulation server. Programs that use the more obscure features of Unix must often be changed or rewritten.

Amoeba is nearly ten years old now and many of the original design decisions have not survived the decade; new ones have taken their place. This paper gives an overview and an explanation of the design of the Amoeba distributed system as it is today.

CR Categories: D.4, C.2.4, D.2.5.
1980 Mathematics Subject Classification: 68A05, 68B20.
Key Words & Phrases: distributed operating system, RPC, protection, security, communication, naming, performance, process management, memory management

1. Introduction

Only ten years ago, computer systems were primarily time-sharing systems supporting many users simultaneously, who would access the system using terminals. Today, most computer systems are personal workstations, on or under people's desks; access to shared resources such as printers, tape drives and shared file systems is usually provided through a local network.

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, and Digital Equipment Corporation.

Report CS-R8937
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The functions of the central time-sharing mainframes have been distributed over a number of processors connected by a network. The processing takes place in the workstations, the file system is located in the file server, and the printer spooling system may be on yet another machine. We refer to such systems as *network operating systems*.

To a large extent, a network operating system is just a time-sharing system whose various functions have been implemented separately and execute on different machines. This arrangement, however, does give the user much more functionality than a time-sharing system would give. A personal workstation provides a guaranteed processing capacity which makes it possible, for instance, to give the user very reasonable graphical performance.

The problem with the network operating system model is that it is not very tolerant of failures. Both the workstation and all the file servers needed have to be up and running to get work done. In addition, the model is wasteful of resources. Even a single user of the system at a particular moment, can only use the capacity of one workstation while all the others are idle.

This is a great pity, because the hardware configuration makes it possible to achieve exactly the opposite. A system with many processors and several file servers has built-in redundancy which can be exploited by making the system carry on properly in spite of many kinds of failures. Also, the system can be made to take advantage of the available processors by splitting up work into parallel tasks.

1.1. Distributed Systems

Operating systems that exploit replicated hardware and independent failure of hardware components for achieving fault tolerance and parallel processing transparently are called *distributed operating systems*. Their study has been popular now for more than two decades. Many of the techniques discovered in distributed systems research have made it into network operating systems and made them more reliable and their distribution of functionality more transparent.

But even though many of the issues in distributed computing are understood and techniques for solving many of the problems are known, it is still poorly understood how to integrate solutions into a coherent and efficient design. The complexity of the parts forming a distributed system may be controllable, yet the complexity resulting from putting the parts together tends to grow well out of bounds.

The problem is illustrated by the fact that, though there are many distributed systems projects, there are very few distributed systems and even fewer are in daily use. 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], the 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.

There are two large problems facing the distributed-system designer. The first has been mentioned already. It is how to integrate techniques into a coherent, transparent and easy-to-use distributed system. The second, which is as least as hard, is how to do this with a system that has at least the same performance as conventional time-sharing or network operating systems. This is hard to achieve, because distributed systems do more than centralized systems: they have to replicate data and do operations carefully in order to be fault tolerant.

1.2. Amoeba

The Amoeba Distributed Systems Project [Mullender, 1985; Mullender and Tanenbaum, 1986] is a project that has solving these two problems as its primary goal. It is a joint project 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 is now about to produce a distributed operating system that can be released to the world as a system that can be used in an engineering workstation environment.

Amoeba is an *object-based system*. Client processes use remote procedure calls to send requests for carrying out operations to objects. Each object is both identified and protected by a *capability*. 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 is the key to protection in Amoeba.

Objects are implemented in terms of server processes that manage them. Capabilities have the identity of the object's server encoded into them so that, given its capability, the system can easily find a server process for an object. The RPC system guarantees that requests and replies are delivered at most once and only to authorized processes. Protection and communication are discussed in Section 2.

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 server*. It maintains a mapping of ASCII path names onto capabilities. For replicated objects it can map the name onto a set of capabilities, one for each replica. The directory server has mechanisms for doing atomic operations on arbitrary collections of name-to-capability mappings. Thus, as long as the objects themselves are used as immutable objects, the directory server can be used as a simple transaction-management system. The directory server is described in Section 3.

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. The bullet server keeps the file metadata in primary memory so it can read and create-write (files are immutable) files in exactly one disk operation. It runs on machines with large main memories (e.g., 32 Mbyte) which can hold the working sets for several users. Clients are encouraged to read and forced to write files in their entirety. Performance figures are given in Section 6.

The Amoeba kernel manages memory segments, multithreaded processes and interprocess communication. All other services, such as file service, that traditional operating system kernels offer are provided in Amoeba by user-space services. The process-management facilities allow remote process creation, debugging, checkpointing, and migration, all using a few simple mechanisms explained in Section 4.

In principle, 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 that much with the user and, as much as possible, they are run elsewhere. Amoeba has a *processor pool* for running most applications. The processor pool typically consists of a large number of single-board computers with a minimum of peripherals (just a network connection, usually). 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 available for other work.

In the Amoeba design, concessions to existing operating systems and software were carefully avoided. Programs developed for, say, Unix, therefore, will not run on Amoeba without adaptation. Since it is rather useful to be able to run existing software on Amoeba, a Unix emulation service has been developed. The Ajax library is linked to Unix object code to produce a binary that will run on Amoeba. For a few system calls, the additional help of the Ajax session server has to be enlisted as well. Ajax is discussed in Section 5.

2. Communication and Protection

Amoeba's communication model is that of a client thread making remote procedure calls [Birrell and Nelson, 1984] on objects to manipulate them. The model is implemented in terms of the client sending a *request* message to the *service* that manages the object. A server thread will carry out the request and return a *reply* message back to the client.

Conceptually, clients communicate with active objects. An active object is implemented as a set of (multithreaded) server processes that manage the (passive) representation of the object — as well as the representations of many other objects, usually. A set of server processes that jointly manages a collection of objects of the same type is referred to as a *service*.

2.1. *Remote Procedure Calls*

The interface for manipulating a type of object is called the object type's *class*. Classes can be composed hierarchically; that is, a class may contain the operations from one or several more primitive 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.).

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 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]);
};
```

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 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, `bio_read` and `bio_write`, 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 `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`.

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 — and will be — extended with more data types.

2.2. *RPC Transport*

AIL generates the code for marshalling and unmarshalling the parameters of remote procedure calls into and out of message buffers and then calls on 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 (among which, 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 be arbitrarily long (the maximum size imposed by the transport mechanism is one gigabyte) and is mostly used to hold the variable-length arguments of RPCs. 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.

The transport mechanism itself consists of the server calls `get_request` and `put_reply`, usually arranged in a loop of a server thread, optionally generated by AIL, as follows:

```
/* 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 */
        put_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);
```

These three calls are implemented as system calls of the Amoeba kernel. The protocol for the transport of messages is network dependent. Over wide-area networks, standard protocols, such as IP or X.25 are used. Over local networks, specialized protocols, designed for fast response and high throughput are used.

2.3. 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. Capabilities consist of 3 parts, a *port*, which identifies the service that manages the object that the capability refers to, a *location hint* which can be used to provide a clue for the location of the object, and an *object part* that identifies the object further within the service. The structure of a capability is shown in FIGURE 1.

When a server thread makes a `get_request` call, it provides its service port to the system. When a client thread calls `do_transaction`, it is the system's job 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*. If objects and processes never moved, locating servers would be easy. The object's (and thus the server's) network address could be put in the location hint field of the capability of the object and that would be that. Unfortunately, making objects stay put is inconvenient for other reasons, so they are allowed to move from one location to another. There can be several reasons for moving them around: machine crashes, server crashes, reducing distance between an object and its clients, load balancing.

Objects can be replicated over several (but usually not all) of the server processes implementing the service that manages the object. Ideally, a locate operation for a client would find the nearest server process that holds a copy of the object. The next best thing would be to find the location of any server that has a copy of the object. In all cases must the location operation find the location of a server process for the object. If it does not hold a copy of the object, it has to be able to get hold of one.

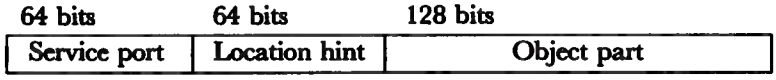

FIGURE 1. Structure of a capability. The service port identifies the service that manages the object. The location hint helps the system locate a suitable server process. The object part identifies the object within the service.

The purpose of the location hint is primarily to find the location of a 'good' server process for an object. If the location hint fails, broadcast is used to find a nearby alternative. The location hint must, in effect, be a sort of 'internet address': It must identify the *local internet*, the local network and the machine address where the object's primary server resides.

The notion of local internet is an important one. A local internet consists of one or several local networks, connected together in such a way that broadcast packets can be conveniently sent over them. If a location hint fails, an attempt is made to find a server by broadcasting a 'where-are-you?' packet over the client's local internet and — if it is different — over the local internet indicated by the capability's location hint.

When a server has been found and it has carried out a remote operation, it may return an 'improved' capability for the object, a capability with a more up-to-date hint field. When objects migrate, this is the mechanism that prevents a client from getting to the object via the old location more than once.

This mechanism has been designed, but has not yet been put in place. Currently, servers are always located by broadcasting for them over the client's local internet. Clients cache the results of these locate operations so that the number of broadcasts remains small. Objects in other local internets can only be located if the gateway server knows about the remote service. The new design was made because the current one does not scale properly [Mullender and Vitányi, 1988].

2.4. Protection

Processes are assumed to run in environments that are secure enough for the purpose for which they execute. A file server, which stores sensitive information for many users must run in a secure machine. An editor, used to edit a public source to fix a bug, can probably be run on almost any machine. The system must be and can be set up in such a way that applications run in a sufficiently trustworthy environment; that is, on a physically secure machine executing an authenticated copy of a secure operating system kernel.

This does not mean that there can be no intruders in the system. A workstation outside a secure computer room can be tampered with by bringing up an unauthorized modified copy of the operating system. A workstation may be present on the network that runs an insecure (or just different) operating system and it may be used to send and receive arbitrary packets through the network.

When a client process sends requests to the file server, one wants to make quite certain that they go to the file server and not to an intruder process. When the reply comes back, one also wants to be absolutely sure that the reply was sent by the file server and no other process. Authentication mechanisms are needed that prevent one process from impersonating another and prevent unauthorized processes from looking at other processes' messages.

Amoeba provides a single protection abstraction with two different implementations, one for reasonably friendly environments and one for hostile environments. Both will be described here briefly, but first, the protection abstraction will be described.

The way it was described earlier, the interface for message exchange is insecure. A malicious client in possession of the capability of a file knows the port of the file service and it can therefore do `get_requests` on the file server's port in order to intercept read and write requests from unsuspecting clients of the file server. Fortunately, the interface does not work exactly as described before.

In order to receive requests, a server has to know the *get port* of the service it implements. In order to send requests to a service, the client has to know (as part of a capability) the *put port* of that service. To make it feasible for the system to deliver requests addressed with a put port to a process asking for requests on a get port, it has to know the relationship between get ports and put ports.

This relationship is provided by a one-way function F. One-way functions are functions that take an argument from a very large domain and compute a value in a range of approximately (or exactly) the same size. The special property of these functions is that, although it is reasonably simple to compute the function, finding an inverse for a arbitrary value of the function is computationally infeasible [Evans, Kantrowitz, and Weiss, 1974]. F is a publicly known one-way function that defines the relationship between a get port G and a put port P as follows:

$$P = F(G)$$

Thus, when one knows the get port, the associated put port can be straightforwardly computed, but knowledge of a put port alone cannot be used to find the associated get port.

A server does `get_requests` with the get port as one of the arguments and the client does `do_operations` with the service's put port in the capability argument. In order to get replies back equally securely, clients also use a pair of ports. Internal to the implementation of `do_operation` the client asks for replies addressed to the client's get port and the server addresses the reply with the client's put port. The client's get port may be viewed as the client process' UID — it is kept in the process' environment and normally inherited on process creation.

As long as one assumes that Amoeba processes use the `get_request`, `put_reply` and `do_operation` interfaces exclusively, this mechanism is secure. Unfortunately, it is very difficult to enforce the use of this interface and prevent use of another. If just one machine on an Ethernet cheats, the security of the whole system is compromised.

The friendly-environment implementation of the Amoeba protection mechanism assumes that the exclusive use of the Amoeba communication interface can indeed be enforced. The implementation is in the Amoeba kernel and also in the Amoeba-communication package of our Unix kernels (in a Unix device driver). Under the assumption that the Amoeba kernel is tamper proof, that the super-users on Unix can be trusted, and that there are no other untrustworthy machines on the network, the friendly-environment implementation is secure. In the Amoeba systems in use at CWI and VU, friendly-environment protection is deemed sufficient — we keep few secrets anyway.

The hostile-environment implementation has the same interface for clients and servers, but uses cryptography to enforce its exclusive use. It depends on a physically secure and trusted *authentication server* that keeps a database of server names, associated ports and encryption keys. When a client establishes communication with a server, the authentication is used to provide proof of the identities of the processes involved and an encryption key with which the messages exchanged between client and server can be encrypted. Clients and servers share an encryption key with the authentication server that is sued to bootstrap the process. These keys are derived from the get port using a one-way function F', which is distinct from F. Thus, the encryption key K_x that a client or server X shares with the authentication server AS and the put port P_x of X derive from the get port G_x of X by

$$K_x = F'(G_x), \text{ and } P_x = F(G_x)$$

The authentication protocol is integrated into the protocols for locating a server. Packet 1 of FIGURE 2 is the broadcast packet with which the client tries to find a server process in the network. Packet 2 is the reply that tells the client where the server process is located. Packets 3, 4 and 5 authenticate client and server to each other and establish a communication key K. The messages are given in FIGURE 2. We use the notation '$[M_1, \{M_2\}_K]$' for a message containing M_1 in plain text and M_2 encrypted with key K.

Client and server each generate a *nonce*, a random number, chosen 'for the nonce', which is used to check whether the key provided by the authentication server is indeed new. The nonce is used to prevent an intruder from replaying an old key which may, by now, be compromised. The client's nonce is N_c, the server's is N_s.

The messages of FIGURE 2 contain more information than shown. In particular, they must contain source and destination network addresses. Packet 3 must also contain the server's network address. The unshown information, however, is not essential to the authentication algorithm, merely to its implementation.

The protocol just described has been vetted by the 'logic for authentication' by BURROWS, ABADI, and NEEDHAM [1988] and was given a clean bill of health; the protocol is correct, nor does it have redundant packets or encryption.

The hostile-environment protection mechanism is currently being implemented, therefore we cannot report any performance experience yet. We do anticipate, however, that session keys can be cached and used for many operations. Only the first operation of a client-server pair will suffer from the

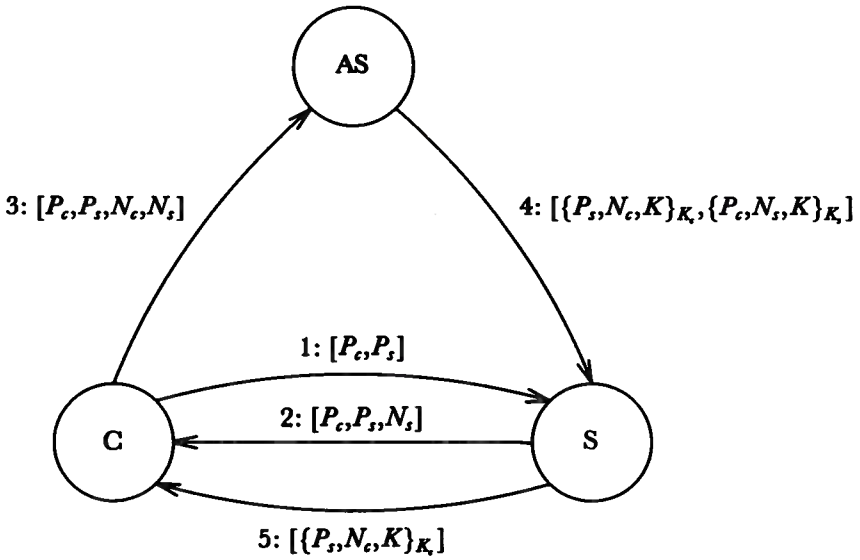

FIGURE 2. Message sequence for the Amoeba authentication protocol.

authentication protocol's overhead, although, of course, every message will be slowed down by having to be encrypted.

3. NAMING

Capabilities form the low-level naming mechanism of Amoeba, but they are very impractical for use by human beings. An extra level of mapping is therefore provided from human-sensible hierarchical path names to capabilities. On Amoeba, a typical user has access to literally thousands of capabilities — 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 needs to be put in only one place.

3.1. Sharing

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 lots of other capabilities.

It does not make sense in this naming hierarchy of capabilities to have a common root: Through the root, every user would be able to access exactly the same set of capabilities and this is obviously not desirable. Instead, every *principal* — that is, every entity that can maintain a set of private objects, a human user, a service, or something else — has a *home directory* which serves as the root of that principal's naming universe. When an individual logs in, a *login server*, which is assumed to be secure and trusted, starts a command interpreter and provides it with the capability of his home directory. From the home directory, all the capabilities that a user needs must be reachable.

Although the directory hierarchy facilitates sharing somewhat, the procedure for making a read-shared object available is still quite involved: the owner capability must be stored in one place, where only the

owner can retrieve it, while the read-only capability must be stored in another, more accessible, place. Things become even more complicated when an object is shared by different groups in different ways.

Robbert VAN RENESSE [1989] nicely solved this problem in the design of the Amoeba directory service, which, for reasons too complicated to explain here, is called Soap. A Soap directory can be viewed as an $n+1$-column table with names in column 0 and capabilities in columns 1 through n. Capabilities can be retrieved from a particular column of a directory with a lookup capability that has the number of the column encoded into it.

Typically, the owner of a directory will have lookup and modify rights for each of the columns of a directory. The owner will store the owner capabilities of his objects in column one and capabilities with fewer rights in the other columns. Other users are then given lookup capabilities only for one or more of the other columns.

3.2. 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 with which (independently implemented) services can collaborate in atomic updates of multiple objects managed by several services [SPECTOR, PAUSCH, and BRUELL, 1988].

In Amoeba, a different approach to atomic updates has been chosen, described in detail in VAN RENESSE [1989]. In the Amoeba approach, the name-to-capability mapping service, the Soap 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, have to be immutable, either because the services that manage them refuse to change them or because the users refrain from changing them.

The atomic transactions as provided by the Soap 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 (such as the password file) resulting in one lost update.

3.3. 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. The directory server obviously must never stop.

The Soap directory service is a replicated service that replicates all the data it stores. It has been designed such that no single-site failure will bring the service down [VAN RENESSE, 1989]. 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-test software, for instance 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 could 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.

This method requires an intruder to break both the directory server and intercept the capabilities from clients talking to it. Hopefully, this is a very tall order [MULLENDER, 1985].

4. PROCESS MANAGEMENT

Amoeba processes can have multiple threads of control. A process, essentially, 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 again, 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* in an *execute process* request to a kernel. 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. The other is the capability of a *handler*, a service that deals with process exit, exceptions, signals and other anomalies of the process.

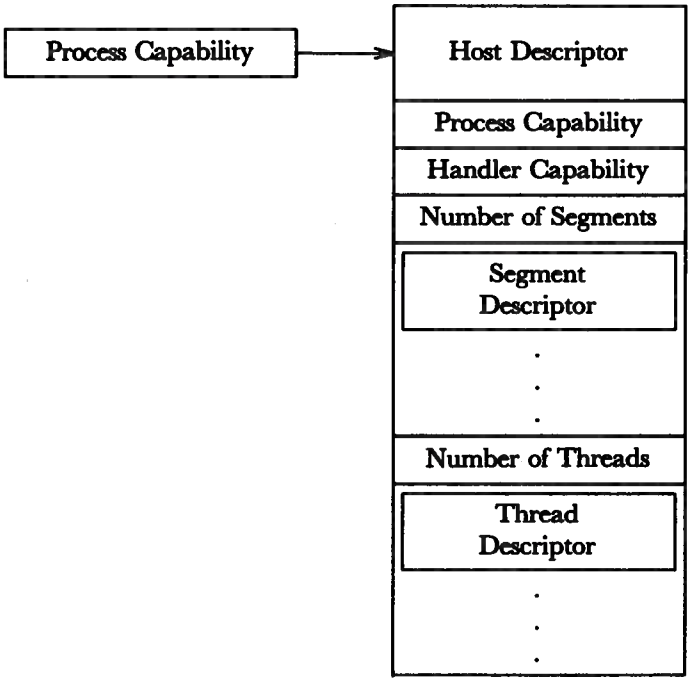

FIGURE 3. Layout of a process descriptor

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, 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, being debugged or being checkpointed.

In most operating systems, system call state is a very 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: `do_operation` and `get_request`.

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 which is taken as evidence it is the owner), 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 either reply 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.

5. Ajax

Amoeba is a new operating system with a system interface that is quite different to that of the popular operating systems of today. Amoeba was developed by a group of people who all used Unix as their operating system vehicle, so it will come as no surprise that the absence of Unix tools on Amoeba would be a major obstacle to using Amoeba for daily work.

As it turns out, writing the software to support Unix code (and, probably, support code for most other operating systems) is fairly straightforward, provided one is prepared to make small changes to a few Unix applications.

The Amoeba support for Unix is called *Ajax*, not after the ancient Greek hero of that name, but because the name starts with an 'A' and ends with an 'X' and because the training grounds of the Ajax soccer club are visible from the CWI windows. It was designed and implemented by Guido van Rossum who used the approach that 90% of the effort goes into getting the last 10% of the Unix utilities running on Amoeba and nobody needs those Unix utilities anyway.

A library was developed that implemented the most common Unix system calls using data structures in the library and calls on the Bullet file server, the Soap 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 haven't made any attempts to port every Unix utility (there are many of which we don't even know what they are for) and the system-maintenance tools are useless on Amoeba anyway. We also have not emulated Unix interprocess communication. The communication provided by Amoeba is a lot faster and certainly an order of magnitude easier to program. Work is in progress to make our Unix interface compatible with the emerging standards (e.g., Posix).

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

We have found that the availability of the Unix utilities have 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.

6. 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 by 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. I 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).

Andy Tanenbaum's plan to build a capability- and object-based system 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 never be done on top of another system. As far as we can tell, only systems with custom-built hardware can outperform Amoeba's RPC and file system performance, taking the capabilities of the underlying hardware into consideration.

On 16.7 MHz Motorola 68020 machines, running the Amoeba kernel, connected by Ethernet (using Lance-chip interfaces), network RPC takes 1.4 ms for a null call. RPCs repeated in a loop with a 30,000-byte `in` (or `out`) character-array argument can ship 5.4 megabits per second between a client and a server process, more than half the Ethernet bandwidth. Five client-server pairs together can achieve a total throughput of 8.4 megabits per second, not counting Ethernet and Amoeba packet headers [van Renesse, 1989].

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 (remote to the client) takes 7 ms; a 1 megabyte file takes 1.6 seconds. More detailed performance numbers and comparisons with other systems can be found in van Renesse, van Staveren, and Tanenbaum [1988].

The Amoeba kernel is small and simple. It implements only a few operations for process management, 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, Motorola 68020 and 68030, National Semiconductor 32000 and Intel 80386 processors and is being ported to MIPS R-2000.

7. References

A. D. Birrell and B. J. Nelson (1984). 'Implementing Remote Procedure Calls'. *ACM Transactions on Computer Systems* **2** (1): 39—59, February 1984.

M. Burrows, M. Abadi, and R. M. Needham (1988). 'Authentication: A practical study in Belief and Action'. *Second Conference on Theoretical Aspects of Reasoning about Knowledge*, 1988.

D. R. Cheriton (1988). 'The V Distributed System'. *Communications of the ACM* **31**: 314—333, March 1988.

A. Evans, W. Kantrowitz, and E. Weiss (1974). 'A User Authentication Scheme Not Requiring Secrecy in the Computer'. *Communications of the ACM* **17** (8): 437—442, August 1974.

S. J. Mullender (1985). *Principles of Distributed Operating System Design.* Ph. D. Thesis, Vrije Universiteit, Amsterdam, October 1985.

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

S. J. Mullender and P. M. B. Vitányi (1988). 'Distributed Match-Making'. *Algorithmica* **3**: 367—391, 1988.

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

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser (1988). *CHORUS Distributed Operating Systems.* Report CS/Technical Report-88-7.6, Chorus Systèmes, Paris, Nov. 1988.

A. Spector, R. Pausch, and R. Bruell (1988). 'Camelot A Flexible, Distributed Transaction Processing System'. *Proceedings Compcon 88*: 432—437, San Francisco, CA, February 1988.

R. van Renesse, H. van Staveren, and A. S. Tanenbaum (1988). 'Performance of the World's Fastest Distributed Operating System'. *Operating System Review* **22** (4): 25—34, October 1988.

R. van Renesse (1989). *The Functional-Processing Model.* Ph. D. Thesis, Vrije Universiteit, Amsterdam, October 1989.

G. van Rossum (1989). 'AIL — A Class-Oriented Stub Generator for Amoeba'. In W. Zimmer (Ed.), *Proceedings of the Workshop on Experience with Distributed Systems.* Springer Verlag, 1989. (To appear.)

B. Walker, G. Popek, R. English, C. Kline, and G. Thiel (1983). 'The LOCUS Distributed Operating System'. *Proceedings Ninth Symposium on Operating System Principles*: 49—70, 1983.