The Amoeba Distributed Operating System
(Part 2)

Sape J. Mullender
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The Amoeba Project is a distributed project on distributed operating systems. The project, which started as the author's PhD research project in 1978 [5], is now a joint project of CWI and Vrije Universiteit in Amsterdam. About a dozen people are working on the project, led by prof. dr. A. S. Tanenbaum (VU) and the author (CWI). Part 1 of this paper appeared in the previous Newsletter and discussed the interprocess communication facilities and the capability-based protection mechanisms of the Amoeba system. Here, in Part 2, some of the Amoeba services will be discussed.

1. Introduction

The basic entities of the Amoeba Distributed Operating System are *clients*, *services* and *objects*. A client process can request the service that manages an object to carry out an operation on it by sending a request to one of the service's server processes. For identification and protection, each object has a *capability*, a 128-bit string that contains enough sparseness that it cannot be forged. Inside the capability is a *port*, a 48-bit string that identifies the service managing the object. When a client makes a request, the system uses this port to find a server process to carry it out.

Traditional operating systems usually provide hundreds of *system calls*. These are 'procedure calls' into the operating system kernel. There are calls to open, read, write and close files, calls to create and destroy processes, to manipulate tape drives and terminals, and so forth. In Amoeba, there are only three* system calls, one for clients and two for servers: A client process makes a server request by calling *trans*; this sends a request to a service and waits for a reply. A server process receives a request by calling *getreq*, and sends a reply by a call to *putrep*. (See Part 1 for more details.)

* Actually, there are a few more for interrupting transactions and dealing with Gateways to other networks. Going into these would make this paper too technical, however.

There are no system calls for creating processes or reading files in Amoeba. When a new process has to be created, a request is sent to the process server, and when a file must be read, a request goes to the file server. Some of these services are *kernel services*, that is, they form part of the operating system kernel, others are implemented as user services, that is, the server runs as an ordinary user process. Amoeba makes no distinction, and clients perceive no difference.

The advantages of this method are obvious: Programmers have to deal with only one way of addressing services, and it allows the system designers to implement services outside the kernel for maintainability or inside it for speed. New versions of a service can always be tested outside the kernel, to be installed inside it only after thorough testing.

The next sections describe a few of the important services provided by the Amoeba system: Process Service, File Service, Directory Service, and Bank Service.

2. Process Service

Managing processes on Amoeba is a task that is carried out by a number of co-operating services. The central one is the service provided by the Amoeba operating system kernel. The kernel assigns processes to the processor so they can do their work. But there's more to process management: processes have to be assigned to the right processor; when a process crashes, or does something irregular, such as attempting to make a Unix system call on an Amoeba kernel, something has to be done about it; programs have to be fetched from a file before they can be run.

The design of the system is such that the Amoeba kernel implements a minimum of basic process management *mechanisms*, on top of which various *policies* can be implemented: If we ever decide to do process management in a different way, we want to run as little risk as possible that we have to change the kernel.

In the previous article, we explained that client processes block when they do transactions, and that server processes block when they wait for a client's request. It is very difficult to write programs using non-blocking transactions. It's much simpler using blocking ones. Additionally, blocking transactions can be implemented much more efficiently than non-blocking ones. To achieve parallelism, one uses parallel processes.

In traditional operating systems, each process runs in its own address space. In distributed systems, processes are created at such an enormous rate that the cost of making each one run in its own address space is prohibitive. Many distributed systems, therefore, provide *light-weight processes*, processes that share a single address space. Processes then have very little context, and process switching can be done very efficiently.

We call a light-weight process a *task*, and a group of tasks sharing an address space a *cluster*, although sometimes we'll use the term *process* for cluster as well. The address space is divided up into *segments*. A cluster can have a number of read-only segments (useful to hold the program's code), and a number of read-write segments, write-only segments (they can be useful, believe it or not), segments that can grow (for the stack), and so on.

Running a program on an Amoeba machine requires the following steps. First, the segments that the program will use must be created and filled with the proper contents. Then a cluster descriptor is given to the kernel, giving the mapping of the segments into the new cluster's address space, and the number and state of its tasks.

When a running cluster is stopped, the Amoeba Kernel hands over a cluster descriptor for it, describing the state of the cluster at the moment it was stopped.

The *process server* is a user-space server which runs on each Amoeba machine and acts as a sort of "agent on the spot" for remote execution. To run a process remotely, its cluster descriptor is sent to the process server on that machine. The process server there then fetches segments over the network, if necessary, and creates the cluster.

Co-operating process servers can use the kernel mechanisms to implement process migration for load balancing, for instance, or if a machine has to be brought down for maintenance.

Cluster descriptors also play an important role in exception handling. Each cluster descriptor contains a table that specifies for each kind of exception which server to call when it occurs. Special codes in the table can be used to ignore certain exceptions, or to kill the cluster.

When an exception occurs, the kernel sends the cluster descriptor to the specified server for handling. The handler can examine and manipulate the cluster using the information provided by the cluster descriptor, and access the cluster's memory through the segment capabilities in the cluster descriptor.

Operating system emulation can be viewed as a special case of debugging. A program, native to another operating system, can be run on Amoeba as if it ran on the operating system it was written for. Before the process is run, its environment is set up so that all system calls it does, all actions that cause exceptions are trapped to an operating system emulator server. The emulator can examine the state of the excepted process, determine what its original operating system would have done when this exception occurred and simulate that with the same mechanisms that the debugger uses.

The Amoeba process service mechanism thus provides a basic mechanism in the Amoeba kernel for process management (segments and cluster descriptors), which is used by user-space servers to augment this service with services for remote execution, load balancing through migration, local and remote debugging, checkpointing and operating system emulation.

3. File Service

The file system has been designed to be highly modular, both to enhance reliability and to provide a convenient testbed for doing research on distributed file systems. It consists of three completely independent pieces: the block service, the file service, and the directory service. In short, the block service provides commands to read and write raw disk blocks. As far as it is concerned, no two blocks are related in any way, that is, it has no concept of a file or other aggregation of blocks. The file service uses the block service to build up files with various properties. Finally, the directory service provides a mapping of symbolic names onto object capabilities.

The block service is responsible for managing raw disk storage. It provides an object-oriented interface to the outside world to relieve file servers from having to understand the details of how disks work. The principle operations it performs are:

- *allocate* a block, write data into it, and return a capability to the block
- given a capability for a block, *free* the block
- given a capability for a block, *read* and return the data contained in it
- given a capability for a block and some data, *write* the data into the block
- given a capability for a block and a key, *lock* or *unlock* the block

These primitives provide a convenient object-oriented interface for file servers to use. In fact, any client who is unsatisfied with the standard file system can use these operations to construct his own [8, 9].

The first four operations of *allocate*, *free*, *read*, and *write* hardly need much comment. The fifth one provides a way for clients to lock individual blocks. Although this mechanism is crude, it forms a sufficient basis for clients (e.g., file systems) to construct more elaborate locking schemes, should they so desire.

One other operation is worth noting. The data within a block is entirely under the control of the processes possessing capabilities for it, but we expect that most file servers will use a small portion of the data for redundancy purposes. For example, a file server might use the first 32 bits of data to contain a file number, and the next 32 bits to contain a relative block number within the file. The block server supports an operation *recovery*, in which the client provides the account number it uses in *allocate* operations and requests a list of all capabilities on the whole disk containing this account number. (The block server stores the account number for each block in a place not accessible to clients.) Although *recovery* is a very expensive operation, in effect requiring a search of the entire disk, armed with all the capabilities returned, a file server that lost all of its internal tables in a crash could use the first 64 bits of each block to rebuild its entire file list from scratch.

The purpose of splitting the block service and file service is to make it easy to provide a multiplicity of different file services for different applications. One such file service that we envision is one that supports flat files with no locking, in other words, the UNIX† model of a file as a linear sequence of bytes with no internal structure and essentially no concurrency control. This model is quite straightforward and will therefore not be discussed here further.

A more elaborate file service with explicit version and concurrency control for a multiuser environment will be described instead [4]. This file service is designed to support data base services, but it itself is just an ordinary, albeit slightly advanced, file service. The basic model behind this file service is that a file is a time-ordered sequence of versions, each version being a snapshot of the file made at a moment determined by a client [2, 7]. At any instant, exactly one version of the file is the *current version*. To use a file, a client sends a message to a file server process containing a file capability and a request to create a new, private version of the current version. The server returns a capability for this new version, which acts as if it is a block for block copy of the current version made at the instant of creation. In other words, no matter what other changes may happen to the file while the client is using his private version, none of them are visible to him. Only changes he makes himself are visible.

Of course, for implementation efficiency, the file is not really copied block for block. What actually happens is that when a version is created, a table of pointers (capabilities) to all the file's blocks is created. The capability granted to the client for the new version actually refers to this version table rather than the file itself. Whenever the client reads a block from the file, a bit is set in the version table to indicate that the corresponding block has been read. When a block is modified in the version, a new block is allocated using the block server, the new block replaces the original one, and its capability is inserted into the version table. A bit indicating that the block is a new one rather than an original is also set. This mechanism is sometimes called "copy on write."

Versions that have been created and modified by a client are called *uncommitted versions*. At a particular moment, the current version may have several (different) uncommitted versions derived from it in use by different clients. When a client is finished modifying his private version, he can ask the file server to *commit* his version, that is, make it the current version instead of the then current version. If the version from which the to-be-committed version was derived is still current at the time of the commit, the commit succeeds and becomes the new current version.

† UNIX is a Trademark of AT&T Bell Laboratories.

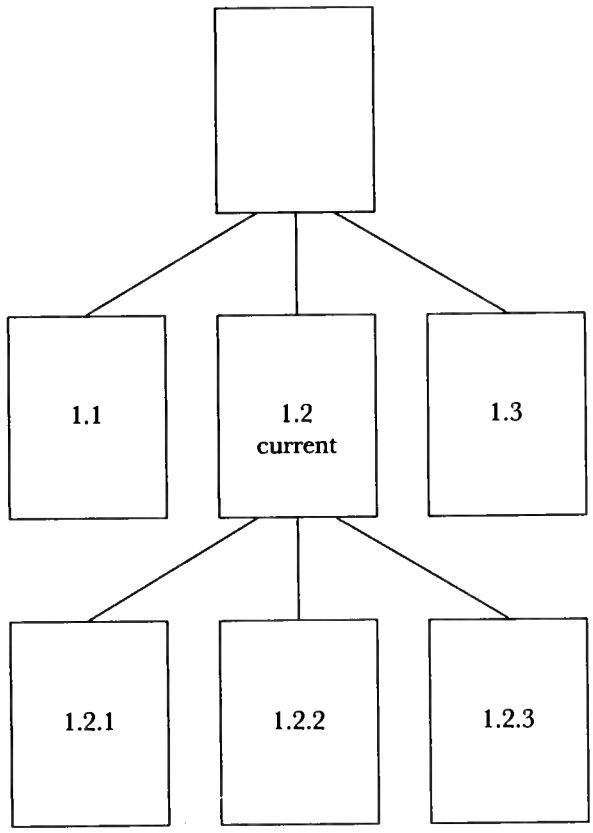

FIGURE 1.

As an example, suppose version 1 is initially the current version, with various clients creating private versions 1.1, 1.2, and 1.3 based on it. If version 1.2 is the first to commit, it wins and 1.2 becomes the new current version, as illustrated in FIGURE 1. Subsequent requests by other clients to create a version will result in versions 1.2.1, 1.2.2, and 1.2.3, all initially copies of 1.2.

The fun begins when the owner of version 1.3 now tries to commit. Version 1, on which it is based, is no longer the current version, so a problem arises. To see how this should be handled, we must introduce a concept from the data base world, *serializability* [1,6]. Two updates to a file are said to be serializable if the net result is the same if they are run sequentially, in one or the other order. As a simple example, consider a two character file initially containing "ab." Client 1 wants to write a "c" into the first character, wait a while, and then write a "d" into the second character. Client 2 wants to write an "e" into the first character, wait a while, and then write an "f" into the second character. If 1 runs first we get "cd"; if 2 runs first we get "ef." Both of these are legal results, since the file server cannot dictate when the users

run. However, its job is to prevent final configurations of "cf" or "de," both of which result from interleaving the requests. If a client locks the file before starting, does all its work, and then unlocks the file, the result will always be either "cd" or "ef," but never "cf" or "de." What we are trying to do is accomplish the same goal without using locking.

The idea behind not locking is that most updates, even on the same file, do not affect the same parts of the file, and hence do not conflict. For example, changes to an airline reservation data base for flights from San Francisco to Los Angeles do not conflict with changes for flights from Amsterdam to London. The strategy behind our commit mechanism is to let everyone make and modify versions at will, with a check for serializability when a commit is attempted. This mechanism has been proposed for data base systems [3], but as far as we know, not for file systems.

The serializability check is straightforward. If a version to be committed, A, is based on the version that is still current, B, it is serializable and the commit succeeds. If it is not, a check must be made to see if all of the blocks belonging to A that the client has read are the same in the current version as they were in the version from which A was derived. If so, the previous commit or commits only changed blocks that the client trying to commit A was not using, so there is no problem and the commit can succeed.

If, however, some blocks have been changed, modifications that A's owner has made may be based on data that are now obsolete, so the commit must be refused, but a list is returned to A's owner of blocks that caused conflicts, that is, blocks marked "read" in A and marked "written" in the current version (or any of its ancestors up to the version on which A is based). At this point, A's owner can make a new version and start all over again. Our assumption is that this event is very unlikely, and that its occasional occurrence is a price worth paying for not having locking, deadlocks, and the delays associated with waiting for locks.

Because it is frequently inconvenient to deal with long binary bit strings such as capabilities, a directory service is needed to provide symbolic naming. The directory service's task is to manage directories, each of which contains a collection of (ASCII name, capability) pairs. The principal operation on a directory object is for a client to present a capability for a directory and an ASCII name, and request the directory service to look up and return the capability associated with the ASCII name. The inverse operation is to store an (ASCII name, capability) pair in a directory whose capability is presented.

4. Bank Service

The bank service is the heart of the resource management mechanism. It implements an object called a "bank account" with operations to transfer virtual money between accounts and to inspect the status of accounts. Bank accounts come in two varieties: individual and business. Most users of the system will just have one individual account containing all their virtual money.

This money is used to pay for CPU time, disk blocks, typesetter pages, and all other resources for which the service owning the resource decides to levy a charge.

Business accounts are used by services to keep track of who has paid them and how much. Each business account has a subaccount for each registered client. When a client transfers money from his individual account to the service's business account, the money transferred is kept in the subaccount for that client, so the service can later ascertain each client's balance. As an example of how this mechanism works, a file service could charge for each disk block written, deducting some amount from the client's balance. When the balance reached zero, no more blocks could be written. Large advance payments and simple caching strategies can reduce the number of messages sent to a small number.

Another aspect of the bank service is its maintenance of multiple currencies. It can keep track of say, virtual dollars, virtual yen, virtual guilders and other virtual currencies, with or without the possibility of conversion among them. This feature makes it easy for subsystem designers to create new currencies and control how they are allocated among the subsystems users.

The bank service described above allows different subsystems to have different accounting policies. For example, a file or block service could decide to use either a buy-sell or a rental model for accounting. In the former, whenever a block was allocated to a client, the client's account with the service would be debited by the cost of one block. When the block was freed, the account would be credited. This scheme provides a way to implement absolute limits (quotas) on resource use. In the latter model, the client is charged for rental of blocks at a rate of X units per kiloblock-second or block-month or something else. In this model, virtual money is constantly flowing from the clients to the servers, in which case clients need some form of income to keep them going. The policy about how income is generated and dispensed is determined by the owner of the currency in question, and is outside the scope of the bank server.

REFERENCES

1. K. P. ESWARAN, J. N. GRAY, R. A. LORIE, AND I. L. TRAIGER, (November 1976). The Notions of Consistency and Predicate Locks in a Database Operating System, *Comm. ACM*, 19.11, 624-633.
2. M. FRIDRICH AND W. OLDER, (December 1981). The Felix File Server, *Proc. Eighth Symposium on Operating Systems Principles*, 15.5, 37-44.
3. H. T. KUNG AND J. T. ROBINSON, (June 1981). On Optimistic Methods for Concurrency Control, *ACM Transactions on Database Systems*, 6.2, 213-226.
4. S.J. MULLENDER AND A.S. TANENBAUM (November 1982). A Distributed File Server Based on Optimistic Concurrency Control, *IR-80*, Vrije Universiteit, Amsterdam.

5. S. J. MULLENDER, (October 1985). *Principles of Distributed Operating System Design*: SMC, Amsterdam.
6. C. H. PAPADIMITRIOU, (October 1979). Serializability of Concurrent Updates, *J. ACM*, 26.4, 631-653.
7. D. REED AND L. SVOBODOVA, (1981). SWALLOW: A Distributed Data Storage System for a Local Network, *Proc. IFIP*, 355-373.
8. M. STONEBRAKER, (July 1981). Operating System Support for Database Management, *Comm. ACM*, 24.7, 412-418.
9. A. S. TANENBAUM AND S. J. MULLENDER, (1982). Operating System Requirements for Distributed Data Base Systems, pp. 105-114 in *Distributed Data Bases, ed. H. J. Schneider, North-Holland Publishing Co.*.