THE AMOEBA DISTRIBUTED OPERATING SYSTEM

Andrew S. Tanenbaum

1. INTRODUCTION

In my first lecture, I gave a general introduction to distributed operating systems. In this one, I would like to go into some detail about a specific example of a distributed operating system of which I was one of the designers--Amoeba (Tanenbaum et al., 1990; Mullender et al., 1991). I will cover the goals, architecture, design and implementation, to give the student a better feel for what a distributed system is really like.

Twenty years ago computers were large and expensive, so many users had to share a single computer. This was the age of the mainframes and minicomputers. Ten years ago it became economically feasible to give each user his own computer, which led to the concept of personal computing. Prices have continued to drop, so it is now feasible to have many CPUs per user. The question is: how should such a system be organized? In particular, what should the software be like.

One possible structure is to give each user a personal multiprocessor, with 10, 20, or more CPUs. However, in such a system, most of the CPUs will be idle most of the time, leading to inefficient use of resources, especially when some users may need large bursts of computing power at random intervals. Instead, we envision a different model, one consisting of a large rack of single-board computers (the processor pool) located in the machine room, plus user terminals for accessing the system. All these machines are to be connected by a high-speed network.

Pool processors are inherently cheaper than workstations because they consist of just a single board with a network connection. There is no keyboard, monitor, or mouse, and the power supply can be shared by many boards. Since the pool processors are allocated only when needed, an idle user only ties up an inexpensive X terminal instead of an expensive workstation.

Amoeba was designed as the operating system for such a system. The Amoeba user can log into the system as a whole, and use it without having to think about which process is running where. To the user, Amoeba is a single integrated system, not as a loose collection of machines connected by a network. In particular, Amoeba has no concept of a ‘‘home machine’’ on which the user normally works, with occasional requests made to other machines. Instead, when a user logs in, a shell is started somewhere, on a machine started by the system. Subsequent commands are run on one or more machines chosen by the system. The user’s workstation simply functions as an X-terminal to allow access to the Amoeba resources. This model is fundamentally different from client-server computing used at many installations, in which the user’s machine is a client that makes requests of various servers from time to time.

In terms of software, an Amoeba system consists of processes and objects. Processes perform work, and communicate with each other by a structured form of message passing. Services and information in Amoeba are organized and protected as objects. Typical objects include processes, memory segments, files, and directories. Each object has a capability that controls it. A process holding the capability for an object can invoke its methods. Capabilities are protected cryptographically, to prevent processes from forging capabilities.

2. THE AMOEBA SYSTEM ARCHITECTURE

The hardware of an Amoeba system has three principal components all connected by a LAN:

- An X-terminal or workstation running X-Windows for each user
- The processor pool (a rack of single-board computers)
- A certain number of dedicated servers (file server, etc.)

Each user has an X-terminal (or workstation running X-Windows) for talking to the system. Although it is technically possible to run user jobs on this machine, that is not normally done, as it is the intention to have Amoeba programs use multiple processors in parallel to improve performance.

The processor pool idea is based on the assumption that the ratio of processors to users is large.

Currently at the VU we typically have 10 users and a pool with 80 processors. As time goes on, the ratio of machines to people will increase. The current personal computer model does not address this trend well, which is why we have devised an alternative model for the future.

In addition to the rack of identical pool processors, a small number of dedicated servers provide certain important services. A *service* is an abstract definition of what the server is prepared to do for its clients. This definition defines what the client can ask for and what the results will be, but it does not specify how many servers are working together to provide the service. In this way, the system has a mechanism for providing fault-tolerant services by having multiple servers doing the work.

An example is the directory server. There is nothing inherent about the directory server or the system design that would prevent a user from starting up a new directory server on a pool processor every time he wanted to look up a file name. However, doing so would be horrendously inefficient, so one or more directory servers are kept running all the time, generally on dedicated machines to enhance their performance. The decision to have some servers always running and others to be started explicitly when needed is up to the system administrator.

3. THE AMOEBA MICROKERNEL

Having looked at the Amoeba hardware model, let us now turn to the software model. Amoeba consists of two basic pieces: a microkernel, which runs on every processor, and a collection of servers that provide most of the traditional operating system functionality.

The Amoeba microkernel runs on all machines in the system. The same kernel can be used on the pool processors, the terminals (assuming that they are computers, rather than physical X terminals), and the specialized servers. The microkernel has four primary functions:

- Manage processes and threads.
- Provide low-level memory management support.
- Support communication.
- Handle low-level I/O.

Let us consider each of these in turn.

Like most operating systems, Amoeba supports the concept of a process. In addition, Amoeba also supports multiple threads of control within a single address space. A process with one thread is essentially the same as a process in UNIX. Such a process has a single address space, a set of registers, a program counter, and a stack.

In contrast, although a process with multiple threads still has a single address space shared by all threads, each thread logically has its own registers, its own program counter, and its own stack. In effect, a collection of threads in a process is similar to a collection of independent processes in UNIX, with the one exception that they all share a single common address space.

The second task of the kernel is to provide low-level memory management. Threads can allocate and deallocate blocks of memory, called *segments*. These segments can be read and written, and can be mapped into and out of the address space of the process to which the calling thread belongs. A process must have at least one segment, but it may also have many more of them. Segments can be used for text, data, stack, or any other purpose the process desires. The operating system does not enforce any particular pattern on segment usage.

The third job of the kernel is to handle interprocess communication. Two forms of communication are provided: point-to-point communication and group communication. These are closely integrated to make them similar. Point-to-point communication is based on the model of a client sending a message to a server, then blocking until the server has sent a reply back. Group communication allows a message to be sent from one source to multiple destinations. Software protocols provide reliable, fault-tolerant group communication to user processes in the presence of lost messages and other errors.

The fourth function of the kernel is to manage low-level I/O. For each I/O device attached to a machine, there is a device driver in the kernel. The driver manages all I/O for the device. Drivers are linked with the kernel and cannot be loaded dynamically.

4. COMMUNICATION IN AMOEBA

Amoeba supports two forms of communication: remote procedure call (RPC) using point-to-point message passing, and group communication. At the lowest level, an RPC consists of a request message followed by a reply message. Group communication uses hardware broadcasting or multicasting if it is available; otherwise, the kernel transparently simulates it with individual messages. In this section we will describe both Amoeba RPC and Amoeba group communication.

Normal point-to-point communication in Amoeba consists of a client sending a message to a server followed by the server sending a reply back to the client. The RPC primitive that sends the request automatically blocks the caller until the reply comes back, thus forcing a certain amount of structure on programs. Separate *send* and *receive* primitives can be thought of as the distributed system's answer to the *goto* statement: parallel spaghetti programming. They should be avoided by user programs and used only by language runtime systems that have unusual communication requirements.

Each standard server defines a procedural interface that clients can call. These library routines are stubs that pack the parameters into messages and invoke the kernel primitives to send the message. During message transmission, the stub, and hence the calling thread, are blocked. When the reply comes back, the stub returns the status and results to the client.

In order for a client thread to do an RPC with a server thread, the client must know the server's address. Addressing is done by allowing any thread to choose a random 48-bit number, called a *port*, to be used as the address for messages sent to it. Different threads in a process may use different ports if they so desire. All messages are addressed from a sender to a destination port. A port is nothing more than a kind of logical thread address. There is no data structure and no storage associated with a port. It is similar to an IP address or an Ethernet address in that respect, except that it is not tied to any particular physical location.

RPC is not the only form of communication supported by Amoeba. It also supports group communication. A group in Amoeba consists of one or more processes that are cooperating to carry out some task or provide some service. Amoeba works best on LANs that support either multicasting or broadcasting (or like Ethernet, both). For simplicity, we will just refer to broadcasting, although in fact the implementation uses multicasting when it can to avoid disturbing machines that are not interested in the message being sent. It is assumed that the hardware broadcast is good, but not perfect. In practice, lost packets are rare, but receiver overruns do happen occasionally. Since these errors can occur they cannot simply be ignored, so the protocol has been designed to deal with them.

The key idea that forms the basis of the implementation of group communication is *reliable broadcasting*. By this we mean that when a user process broadcasts a message the user-supplied message is delivered correctly to all members of the group, even though the hardware may lose packets. Central to the algorithm is a process called the *sequencer*, which numbers broadcasts in order. When an application process executes a broadcasting primitive, a trap to its kernel occurs. The kernel sends the message to the sequencer using a normal point-to-point message.

Now consider what happens at the sequencer when a *Request for Broadcast* packet arrives there. If the message is new (normal case), the next sequence number is assigned to it, and the sequencer counter incremented by 1 for next time. The message and its identifier are then stored in a *history buffer*, and the message is then broadcast. If the packet is a retransmission, it is ignored.

Finally, let us consider what happens when a kernel receives a broadcast. First, the sequence number is compared to the sequence number of the broadcast received most recently. If the new one is 1 higher (normal case), no broadcasts have been missed, so the message is passed up to the application program, assuming that it is waiting. If it is not waiting, it is buffered until the program calls *ReceiveFromGroup*.

Suppose that the newly received broadcast has sequence number 25, while the previous one had number 23. The kernel is immediately alerted to the fact that it has missed number 24, so it sends a point-to-point message to the sequencer asking for a private retransmission of the missing message. The sequencer fetches the missing message from its history buffer and sends it. When it arrives, the receiving kernel processes 24 and 25, passing them to the application program in numerical order. Thus the only effect of a lost message is a (normally) minor time delay. All application programs see all broadcasts in the same order, even if some messages are lost.

Several variations of this algorithm also exist. In one, the process wanting to send a broadcast just does so, and the sequencer just broadcasts an OK message giving its sequence number. In another variant, processes first ask the sequencer for a number, then use this number in their own broadcasts. These variants make different tradeoffs between bandwidth and interrrupts.

5. PROCESS MANAGEMENT IN AMOEBA

A process in Amoeba is basically an address space and a collection of threads that run in it. A process with one thread is roughly analogous to a UNIX process in terms of how it behaves and what it can do. A process is an object in Amoeba. When a process is created, the parent process is given a capability for the child process, just as with any other newly created object. Using this capability, the child can be suspended, restarted, signaled, or destroyed.

Process management is handled at three different levels in Amoeba. At the lowest level are the process servers, which are kernel threads running on every machine. To create a process on a given machine, another process does an RPC with that machine's process server, providing it with the necessary information. At the next level up we have a set of library procedures that provide a more convenient interface for user programs. Several flavors are provided. They do their job by calling the low-level interface procedures. Finally, the simplest way to create a process is to use the run server, which does most of the work of determining where to run the new process.

Some of the process management calls use a data structure called a *process descriptor* to provide information about the process to be run. One field in the process descriptor tells which CPU architecture the process can run on. In heterogeneous systems, this field is essential to make sure that Pentium binaries are not run on SPARCs, and so on. Another field contains the process' owner's capability. When the process terminates or is stunned (see below), RPCs will be done using this capability to report the event. It also contains descriptors for all the process' segments, which collectively define its address space, as well as descriptors for all its threads. Finally, the process descriptor also contains a descriptor for each thread in the process. The content of a thread descriptor is architecture dependent, but as a bare minimum, it contains the thread's program counter and stack pointer.

Amoeba supports a simple threads model. When a process starts up, it has one thread. During execution, the process can create additional threads, and existing threads can terminate. The number of threads is therefore completely dynamic. When a new thread is created, the parameters to the call specify the procedure to run and the size of the initial stack. Although all threads in a process share the same program text and global data, each thread has its own stack, its own stack pointer, and its own copy of the machine registers.

Three methods are provided for threads to synchronize: signals, mutexes, and semaphores. Signals are asynchronous interrupts sent from one thread to another thread in the same process. A mutex is like a binary semaphore. General semaphores are also provided.

All threads are managed by the kernel. The advantage of this design is that when a thread does an RPC, the kernel can block that thread and schedule another one in the same process if one is ready. Thread scheduling is done using priorities, with kernel threads getting higher priority than user threads. Thread scheduling can be set up to be either pre-emptive or run-to-completion (i.e., threads continue to run until they block), as the process wishes.

6. OBJECTS AND CAPABILITIES IN AMOEBA

The basic unifying concept underlying all the Amoeba servers and the services they provide is the *object*. An object is an encapsulated piece of data upon which certain well-defined operations may be performed. It is, in essence, an abstract data type. Objects are passive. They do not contain processes or methods or other active entities that do things. Instead, each object is managed by a server process.

To perform an operation on an object, a client does an RPC with the server, specifying the object, the operation to be performed, and optionally, any parameters needed. The server does the work and returns the answer. Operations are performed synchronously, that is, after initiating an RPC with a server to get some work done, the client thread is blocked until the server replies. Other threads in the same process are still runnable, however.

Objects are named and protected in a uniform way, by special tickets called *capabilities*. To

create an object, a client does an RPC with the appropriate server specifying what it wants. The server then creates the object and returns a capability to the client. On subsequent operations, the client must present the capability to identify the object. A capability is just a long binary number.

Capabilities are used throughout Amoeba for both naming of all objects and for protecting them. This single mechanism leads to a uniform naming and protection scheme. It also is fully location transparent. To perform an operation on an object, it is not necessary to know where the object resides. In fact, even if this knowledge were available, there would be no way to use it.

The standard file system consists of three servers, the *bullet server*, which handles file storage, the *directory server*, which takes care of file naming and directory management, and the *replication server*, which handles file replication. The file system has been split into these separate components to achieve increased flexibility and make each of the servers straightforward to implement.

Very briefly, a client process can create a file using the *create* call. The bullet server responds by sending back a capability that can be used in subsequent calls to *read* to retrieve all or part of the file. In most cases, the user will then give the file an ASCII name, and the (ASCII name, capability) pair will be given to the directory server for storage in a directory, but this operation has nothing to do with the bullet server.

The bullet server was designed to be very fast (hence the name). It was also designed to run on machines having large primary memories and huge disks, rather than on low-end machines, where memory is always scarce. The organization is quite different from that of most conventional file servers. In particular, files are *immutable*. Once a file has been created, it cannot subsequently be changed. It can be deleted and a new file created in its place, but the new file has a different capability from the old one. This fact simplifies automatic replication, as will be seen. It is also well suited for use on large-capacity, write-once optical disks.

Because files cannot be modified after their creation, the size of a file is always known at creation time. This property allows files to be stored contiguously on the disk and also in the main memory cache. By storing files contiguously, they can be read into memory in a single disk operation, and they can be sent to users in a single RPC reply message. These simplifications lead to the high performance.

The conceptual model behind the file system is thus that a client creates an entire file in its own memory, and then transmits it in a single RPC to the bullet server, which stores it and returns a capability for accessing it later. To modify this file (e.g., to edit a program or document), the client sends back the capability and asks for the file, which is then (ideally) sent in one RPC to the client's memory. The client can then modify the file locally any way it wants to. When it is done, it sends the file to the server (ideally) in one RPC, thus causing a new file to be created and a new capability to be returned. At this point the client can ask the server to destroy the original file, or it can keep the old file as a backup.

As a concession to reality, the bullet server also supports clients that have too little memory to receive or send entire files in a single RPC. When reading, it is possible to ask for a section of a file, specified by an offset and a byte count. This feature allows clients to read files in whatever size unit they find convenient.

The bullet server, as we have seen, just handles file storage. The naming of files and other objects is handled by the *directory server*. Its primary function is to provide a mapping from human-readable (ASCII) names to capabilities. Processes can create one or more directories, each of which can contain multiple rows. Each row describes one object and contains both the object's name and its capability. Operations are provided to create and delete directories, add and delete rows, and look up names in directories. Unlike bullet files, directories are *not* immutable. Entries can be added to existing directories and entries can be deleted from existing directories.

Directories themselves are objects and are protected by capabilities, just as other objects. The operations on a directory, such as looking up names and adding new entries, are protected by bits in the *Rights* field, in the usual way. Directory capabilities may be stored in other directories, permitting hierarchical directory trees and more general structures. In addition to these servers, a variety of other servers also exist for many typical operating system functions.

REFERENCES

Tanenbaum, A.S., Renesse, R. van, Staveren, H. van., Sharp, G.J., Mullender, S.J., Jansen, J., and Rossum, G. van: "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM* vol. 33, pp. 46-63, Dec. 1990.

Mullender, S.J., Rossum, G. van, Tanenbaum, A.S., Renesse, R. van, and Staveren, H. van: "Amoeba—A Distributed Operating System for the 1990s," *IEEE Computer*, vol. 23, pp. 44-53, May 1990.