The Amoeba Microkernel

Andrew S. Tanenbaum
M. Frans Kaashoek

Dept. of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081
1081 HV Amsterdam, The Netherlands
Internet: ast@cs.vu.nl, kaashoek@cs.vu.nl

ABSTRACT

In this paper we will give an up-to-date overview of the Amoeba distributed operating system microkernel. We will examine process management, memory management, and the communication primitives, emphasizing the latter since these contains the most new ideas.

1. INTRODUCTION

Amoeba is a distributed operating system designed to connect together a large number of machines in a transparent way. Its goal is to make the entire system look to the users like a single computer. The system consists of two parts: a microkernel and server processes. In this paper we will describe the microkernel. For information about other aspects of Amoeba, see Mullender et al., 1990 and Tanenbaum et al., 1990.

An Amoeba system consists of several components, including a pool of processors (compute service, where most of the work is done), terminals (e.g., computers running the X window servers) that handle the user interface, and specialized servers (e.g., directory servers). All these machines normally run the same (micro)kernel.

The microkernel has four primary functions:

1. Manage processes and threads.
2. Provide low-level memory management support.
3. Support communication.
4. 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

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

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.

A typical use for multiple threads might be in a file server, in which every incoming request is assigned to a separate thread to work on. That thread might begin processing the request, then block waiting for the disk, then continue work. By splitting the server up into multiple threads, each thread can be purely sequential, even if it has to block waiting for I/O. Nevertheless, all the threads can, for example, have access to a single shared software cache. Threads can synchronize using semaphores or mutexes to prevent two threads from accessing the shared cache simultaneously.

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 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. Normally, users do not think in terms of segments, but this facility could be used by libraries or language run-time systems.

The third job of the kernel is to handle interprocess communication. Two forms of communication are provided: point-to-point communication and group communication.

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. This request/reply exchange is the basis on which almost everything else is built. The request/reply is usually packaged in library routine so the remote call looks like a local procedure call. This mechanism is generally known as remote procedure call (RPC), and is discussed in Birrell and Nelson (1984).

The other form of communication is group communication. It allows a message to be sent from one source to multiple destinations. Software protocols provide reliable, fault-tolerant group communication to user processes even with lost messages and other errors.

Both the point-to-point message system and the group communication make use of a specialized protocol called FLIP. This protocol is a network layer protocol, and has been specifically designed to meet the needs of distributed computing. It deals with both unicasting and multicasting on complex internetworks.

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.

In the following sections we will discuss process management, memory management, and communication services and protocols provided by the Amoeba microkernel. However, since the concept of an object permeates the whole system, we will first briefly describe how objects fit into Amoeba.

2. OBJECTS

Amoeba is organized as a collection of objects (essentially abstract data types), each with some number of operations that processes can perform on it. Objects are generally large, like files, rather than small, like integers, due to the overhead required in accessing an object. Each object is managed by an object server process. Operations on an object are performed by sending a message to the object's server.

When an object is created, the server returns a *capability* to the process creating it. The capability is used to address and protect the object. A typical capability is shown in Fig. 1. The *Port* field identifies the server. The *Object* field tells which object is being referred to, since a server normally will manage thousands of objects. The *Rights* field specifies which

operations are allowed (e.g., a capability for a file may be read-only). Since capabilities are managed in user space the *Check* field is needed to protect them cryptographically, to prevent users from tampering with them.

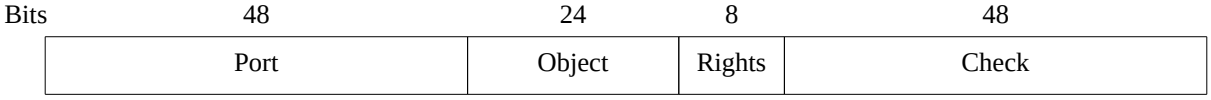

Fig. 1. A typical capability.

The basic algorithm used to protect objects is as follows (Tanenbaum et al., 1986). When an object is created, the server picks a random *Check* field and stores it both in the new capability and inside its own tables. All the rights bits in a new capability are initially on, and it is this **owner capability** that is returned to the client. When the capability is sent back to the server in a request to perform an operation, the *Check* field is verified.

To create a restricted capability, a client can pass a capability back to the server, along with a bit mask for the new rights. The server takes the original *Check* field from its tables, EXCLUSIVE ORs it with the new rights (which must be a subset of the rights in the capability), and then runs the result through a one-way function. Such a function, $y = f(x)$, has the property that given x it is easy to find y, but given only y, finding x requires an exhaustive search of all possible x values (Evans et al., 1974).

The server then creates a new capability, with the same value in the *Object* field, but the new rights bits in the *Rights* field and the output of the one-way function in the *Check* field. The new capability is then returned to the caller. In this way, processes can give other processes restricted access to their objects.

3. PROCESS MANAGEMENT IN AMOEBA

A process in Amoeba is basically an address space and a collection of threads that run in it. In this section we will explain how processes and threads work, and how they are implemented.

3.1. Processes

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, or destroyed.

Process creation in Amoeba is different from UNIX. The UNIX model of creating a child process by cloning the parent is inappropriate in a distributed system due to the potential overhead of first creating a copy somewhere (FORK) and almost immediately afterwards replacing the copy with a new program (EXEC). Instead, in Amoeba it is possible to create a new process on a specific processor with the intended memory image starting right at the beginning. The children, can, in turn, create their own children, leading to a tree of processes.

Process management is handled by calling 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 a higher level, a user-level server, the *run* server, can be invoked to choose a machine and start the process there. The *run* server keeps track of the load on the various processors and chooses the most favorable machine based on CPU load and memory usage.

Some of the process management calls use a data structure called a *process descriptor* to provide information about a process to be run. It is used both for new processes and those that

have run for a while and been suspended (e.g., by a debugger). One field in the process descriptor (see Fig. 2) tells which CPU architecture the process can run on. In heterogeneous systems, this field is essential to make sure 386 binaries are not run on SPARCs, and so on.

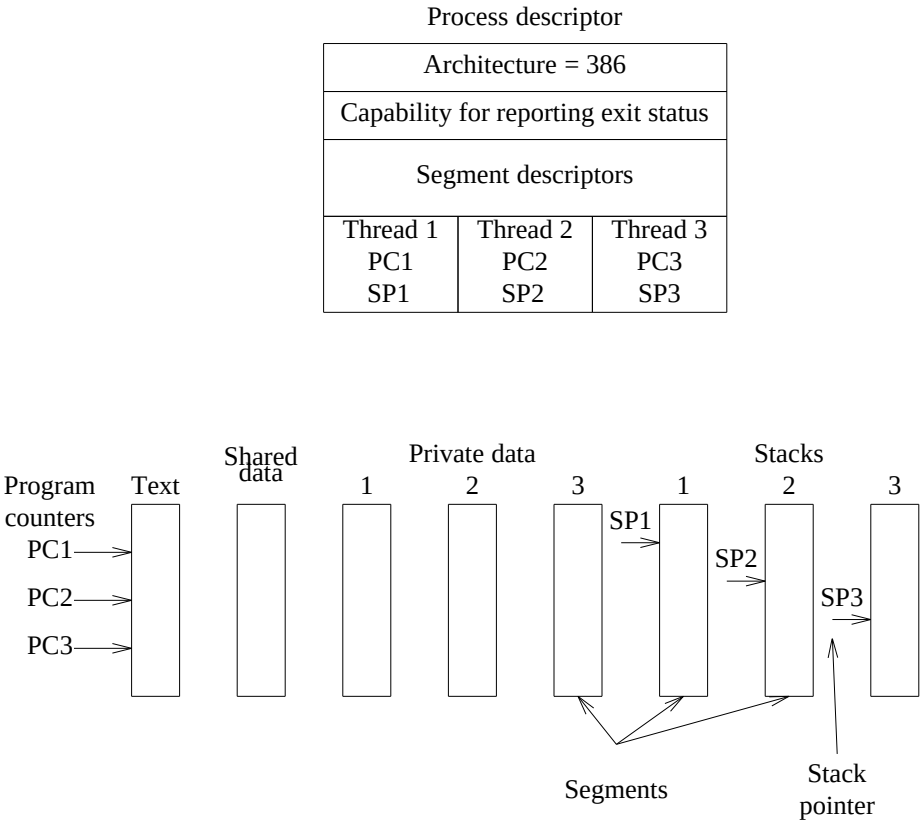

Fig. 2. A process descriptor and the corresponding process. In this example, the process has one text segment, one data segment shared by all threads, three segments that are each private to one thread, and three stack segments.

Another field contains a capability for communicating the exit status to the owner. 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.

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. It may also contain additional information necessary to run the thread, including other registers, the thread's state, and various flags.

The low-level process interface to the process management system consists of several procedures. Only three of these will concern us here. The first one, *exec*, is the most important. It has two input parameters, the capability for a process server and a process descriptor. Its function is to do an RPC with the specified process server asking it to run the process. If the call is successful, a capability for the new process is returned to the caller.

A second important procedure is *getload*. It returns information about the CPU speed, current load, and amount of memory free at the moment. It is used by the run server to determine the best place to execute a new process.

A third major procedure is *stun*. A process' parent can suspend it by *stunning* it. More

commonly, the parent can give the process' capability to a debugger, which can stun it and later restart it for interactive debugging purposes. Two kinds of stuns are supported: normal and emergency. They differ with respect to what happens if the process is blocked on one or more RPCs at the time it is stunned. With a normal stun, the process sends a message to the server it is currently waiting for saying, in effect: ''I have been stunned. Finish your work instantly and send me a reply.'' If the server is also blocked, waiting for another server, the message is propagated further, all the way down the line to the end, where it generates an interrupt. If the server at the end of the line catches the interrupt, it replies with a special error message. In this way, all the pending RPCs are terminated quickly in a clean way, with all of the servers finishing properly. The nesting structure is not violated, and no ''long jumps'' are needed. Processes that do not want to be interrupted can have their wish by simply not enabling handlers (the default is to ignore stuns). Then, the client process stays alive until it receives the reply from the server process.

An emergency stun stops the process instantly. It sends messages to servers that are currently working for the stunned process, but does not wait for the replies. The computations being done by the servers become orphans. When the servers finally finish and send replies, these replies are discarded.

3.2. Threads

Amoeba supports a simple threads model. When a process starts up, it has at least one thread and possibly more. The number of threads is dynamic. During execution, the process can create additional threads, and existing threads can terminate. 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. In addition, if a thread wants to create and use variables that are global to all its procedures but invisible to other threads, library procedures are provided for that purpose. These variables are managed by the thread itself; the kernel does not intervene.

Three methods are provided for thread synchronization: signals, mutexes, and semaphores. Signals are asynchronous interrupts sent from one thread to another thread in the same process. They are conceptually similar to UNIX signals, except that they are between threads rather than between processes. Signals can be raised, caught, or ignored. Asynchronous interrupts between processes use the stun mechanism.

The second form of interthread communication is the mutex. A *mutex* is like a binary semaphore. It can be in one of two states, locked or unlocked. Trying to lock an unlocked mutex causes it to become locked. The calling thread continues. Trying to lock a mutex that is already locked causes the calling thread to block until another thread unlocks the mutex. If more than one thread is waiting on a mutex, when it is unlocked, exactly one thread is released. In addition to the calls to lock and unlock mutexes, there is also a call that tries to lock a mutex, but if it is unable to do so within a specified interval, it times out and returns an error code.

The third way threads can synchronize is by counting semaphores. These are slower than mutexes, but there are times when they are needed. They work in the usual way, except that here too an additional call is provided to allow a DOWN operation to time out if it is unable to succeed within a specified interval.

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 having higher priority than user threads. Within a user process, threads do not have priorities, and run nonpreemptively.

4. MEMORY MANAGEMENT IN AMOEBA

Amoeba also has a simple memory model. A process can have any number of segments and they can be located wherever it wants in the process' virtual address space. Segments are not swapped or paged, so a process must be entirely memory resident to run. Since the hardware MMU is used, a segment can be located anywhere within the virtual address space. Each segment is stored contiguously in physical memory.

Although this design is perhaps somewhat unusual these days, it was done for three reasons: performance, simplicity, and economics. Having a process entirely in memory all the time makes RPC go faster. When a large block of data must be sent, the system knows that all of the data is contiguous not only in virtual memory, but also in physical memory. This knowledge saves having to check if all the pages containing the buffer happen to be around at the moment, and eliminates having to wait for them if they are not. Similarly, on input, the buffer is always in memory, so the incoming data can be placed there simply and without page faults. This design was one of the factors that allowed Amoeba to achieve high transfer rates for large RPCs (Tanenbaum et al., 1990).

The second reason for the design is simplicity. Not having paging or swapping makes the system considerably simpler and makes the kernel smaller and more manageable. However, it is the third reason that makes the first two feasible. Memory is becoming so cheap that within a few years, all Amoeba machines will probably have tens of megabytes of it. Such large memories will reduce the need for paging and swapping, namely, to fit large programs into small machines. Programs that do not fit in physical memory cannot be run on Amoeba.

Processes have several calls available to them for managing segments. Most important among these is the ability to create, destroy, read, and write segments. When a segment is created, the caller gets back a capability for it. This capability is used for all the other calls involving the segment.

Because segments can be read and written, it is possible to use them to construct a main memory file server. To start, the server creates a segment as large as it can, determining the maximum size by asking the kernel. This segment will be used as a simulated disk. The server then formats the segment as a file system, putting in whatever data structures it needs to keep track of files. After that, it is open for business, accepting and processing requests from clients.

Virtual address spaces in Amoeba are constructed by mapping segments into them. When a process is started, it must have at least one segment. Once it is running, a process can create additional segments and map them into its address space at any unused virtual address. Figure 3 shows a process with three memory segments currently mapped in.

A process can also unmap segments. Furthermore, a process can specify a range of virtual addresses and request that the range be unmapped, after which those addresses are no longer legal. When a segment or a range of addresses is unmapped, a capability is returned, so the segment may still be accessed, or even mapped back in again later, possibly at a different virtual address (on the same processor).

A segment may be mapped into the address space of two or more processes at the same time. This allows processes to operate on shared memory. For example, two processes can map the screen buffer or other hardware devices into their respective address spaces. Also, cooperating processes can share a buffer. Segments cannot be shared over a network.

5. COMMUNICATION IN AMOEBA

Amoeba supports two forms of communication: RPC, which is based on point-to-point message passing, and group communication. At the lowest level, an RPC consists of a request message sent by a client to a server followed by a reply message from the server back to the client. Group communication uses hardware broadcasting or multicasting if it is available;

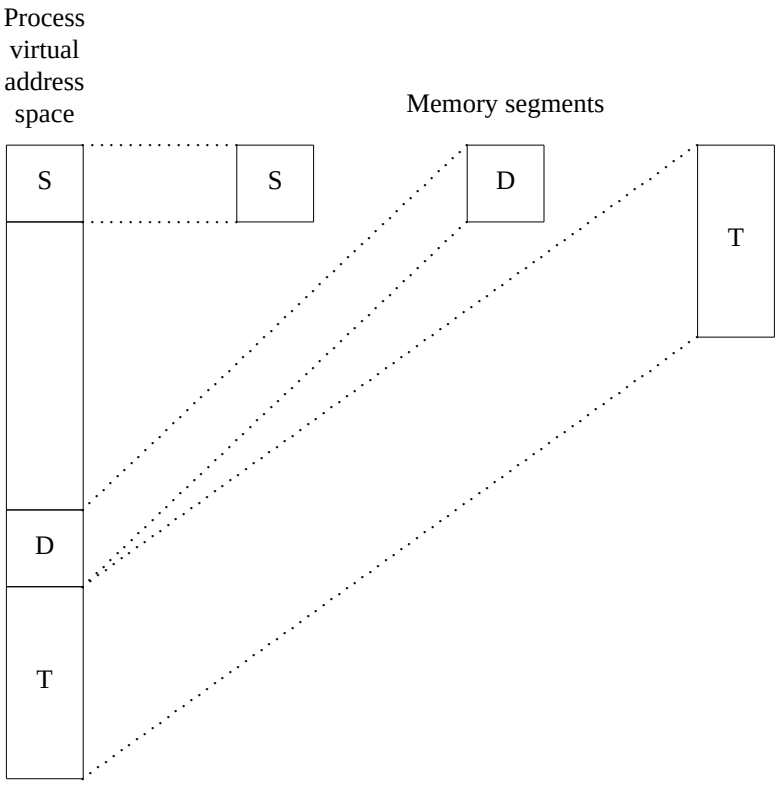

Fig. 3. A process with three segments mapped into its virtual address space.

otherwise it transparently simulates it with individual messages. In this section we will describe both RPC and group communication, and then discuss the underlying FLIP protocol that is used to support them.

5.1. Remote Procedure Call

All 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. It is not possible for a client to just send a message and then go do something else. The 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.

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 actually send the message. During message transmission, the stub, and hence the calling thread, is blocked. When the reply comes back, the stub returns the status and results to the client. Although the kernel-level primitives are closely related to the message passing, the use of stubs makes this mechanism look like RPC to the programmer, so we will refer to the basic communication primitives as RPC, rather than the slightly more precise ''request/reply message exchange.'' Stubs can either be hand written or generated by a stub compiler.

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. The first field in each capability gives the port of the server that manages the object.

RPC Primitives

The RPC mechanism makes use of three principal kernel primitives, as listed below. Programs (or more often, library procedures) can make these calls to send and receive messages.

1. get_request - indicates a server's willingness to listen on a port
2. put_reply - done by a server when it has a reply to send
3. trans - send a message from client to server and wait for the reply

The first two are used by servers. The third is used by clients to *transmit* a message and wait for a reply. All three are true system calls, that is, they do not work by sending a message to a communication server thread. Users access the calls through library procedures, as usual, however.

When a server wants to go to sleep waiting for an incoming request, it calls *get_request*. This procedure has three parameters, as follows:

```
get_request(&header, buffer, bytes)
```

The first parameter points to a message header, the second points to a data buffer, and the third tells how big the data buffer is. This call is analogous to

```
read(fd, buffer, bytes)
```

in UNIX in that the first parameter identifies what is being read, the second provides a buffer to put the data, and the third tells how big the buffer is. The analogy is not strict because the header contains multiple fields, some of which are filled in when the call returns.

When a request message is transmitted over the network, it contains a header and (optionally) a data buffer. The header is a fixed 32-byte structure and is shown in Fig. 4. The first parameter of the *get_request* calls tells the kernel where to put the incoming header. In addition, prior to making the *get_request* call, the server must initialize the header's *Port* field to contain the port it is listening to. This is how the kernel knows which server is listening to which port. The incoming header overwrites the one initialized by the server.

When a server is blocked on a *get_request* waiting for a message and one arrives, the server is unblocked. It normally first inspects the header to find out what the client wants. The *Signature* field is currently not in use, but is reserved for authentication purposes.

The remaining fields are not specified by the RPC protocol, so a server and client can agree to use them any way they want. The normal conventions are as follows. Most requests to servers contain a capability, to specify the object being operated on. Many replies also have a capability as a return value. The *Private* part is normally used to hold the rightmost three fields of the capability.

Most servers support multiple operations on their objects, such as reading, writing, and destroying. The *Command* field is conventionally used on requests to indicate which operation is needed. On replies it tells whether the operation was successful or not, and if not, it gives the reason for failure.

The last three fields hold parameters, if any. For example, when reading a segment or file, they can be used to indicate the offset within the object to begin reading at, and the number of bytes to read.

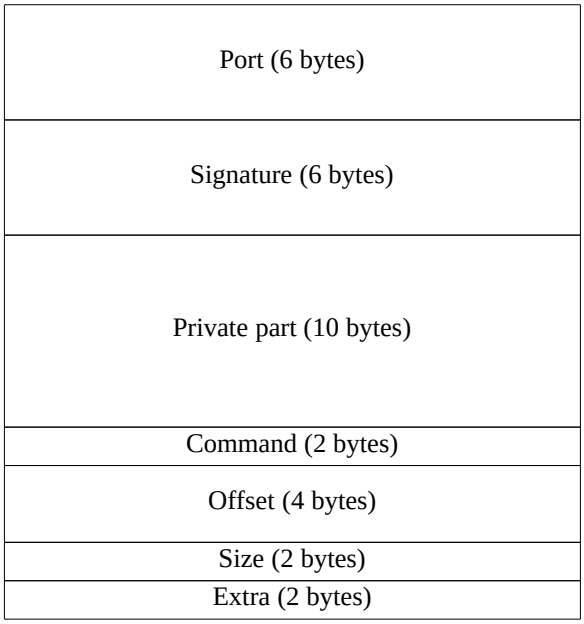

Fig. 4. The header used on all Amoeba request and reply messages. The numbers in parentheses give the field sizes in bytes.

Note that for many operations, no buffer is needed or used. In the case of reading again, the object capability, the offset, and the size all fit in the header. When writing, the buffer contains the data to be written. On the other hand, the reply to a READ contains a buffer, whereas the reply to a WRITE does not.

After the server has completed its work, it makes a call

```
put_reply(&header, buffer, bytes)
```

to send back the reply. The first parameter provides the header and the second provides the buffer. The third parameter tells how big the buffer is. If a server thread does a *put_reply* without having previously done an unmatched *get_request*, the *put_reply* fails with an error. Similarly, two consecutive *get_request* calls fail. The two calls must be paired in the correct way.

Now let us turn from the server to the client. To do an RPC, the client calls a stub which makes the following call:

```
trans(&header_in, buffer_in, bytes_in, &header_out, buffer_out, bytes_out)
```

The first three parameters provide information about the header and buffer of the outgoing request. The last three provide the same information for the incoming reply. The *trans* call sends the request and blocks the client until the reply has come in. This design forces processes to stick closely to the client-server RPC communication paradigm, analogous to the way structured programming techniques prevent programmers from doing things that generally lead to poorly structured programs (such as using unconstrained GOTO statements).

If Amoeba actually worked as described above, it would be possible for an intruder to impersonate a server just by doing a *get_request* on the server's port. These ports are public after all, since clients must know them to contact the servers. Amoeba solves this problem cryptographically. Each port is actually a pair of ports: the *get-port*, which is private, only

known to the server, and the *put-port*, which is known to the whole world. The two are related through a one-way function, *F*, according to the relation:

put-port = *F*(get-port)

When a server does a *get_request*, the corresponding put-port is computed by the kernel and stored in a table of ports being listened to. All *trans* requests use put-ports, so when a packet arrives at a machine, the kernel compares the put-port in the header to the put-ports in its table to see if any match. Since get-ports never appear on the network and cannot be derived from the publicly known put-ports, the scheme is secure. It is illustrated in Fig. 5 and described in more detail in (Tanenbaum et al., 1986).

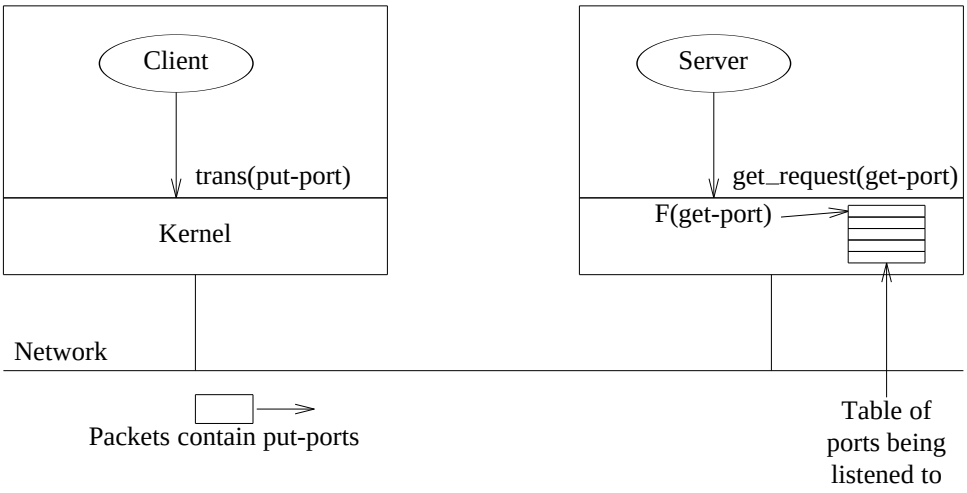

Fig. 5. Relationship between get-ports and put-ports.

Amoeba RPC supports at-most-once semantics. In other words, when an RPC is done, the system guarantees that an RPC will never be carried out more than one time, even in the face of server crashes and rapid reboots.

5.2. Group Communication in Amoeba

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. Processes can be members of several groups at the same time. Groups are closed. The usual way for a client to access a service provided by a group is to do an RPC with one of its members. That member then uses group communication within the group, if necessary, to determine who will do what.

Group Communication Primitives

The operations available for group communication in Amoeba are listed in Fig. 6. *CreateGroup* creates a new group and returns a group identifier used in the other calls to identify which group is meant. The parameters specify various sizes and how much fault tolerance is required (how many failed members the group must be able to withstand and continue to function correctly).

JoinGroup and *LeaveGroup* allow processes to enter and exit from existing groups. One of the parameters of *JoinGroup* is a small message that is sent to all group members to announce the presence of a newcomer. Similarly, one of the parameters of *LeaveGroup* is

Call	Description
CreateGroup	Create a new group and set its parameters
JoinGroup	Make the caller a member of a group
LeaveGroup	Remove the caller from a group
SendToGroup	Reliably send a message to all members of a group
ReceiveFromGroup	Block until a message arrives from a group member
ResetGroup	Initiate recovery after a process crash

Fig. 6. Amoeba group communication primitives.

another small message sent to all members to say goodbye and wish them good luck in their future activities. The point of the little messages is to make it possible for all members to know who their comrades are, in case they are interested. When the last member of a group calls *LeaveGroup*, the group is destroyed.

SendToGroup atomically broadcasts a message to all members of a specified group, in spite of lost messages, finite buffers, and processor crashes. If two processes call *SendToGroup* nearly simultaneously, the system ensures that all group members will receive the messages in the same order. This is guaranteed; programmers can count on it.

ReceiveFromGroup tries to get a message from a specified group. If no message is available (buffered by the kernel) the caller blocks until one is available. If a message has already arrived, the caller gets the message with no delay. The protocol insures that under no conditions are messages irretrievably lost.

The final call, *ResetGroup* is used to recover from crashes. It specifies how many members the new group must have as a minimum. If the kernel is able to establish contact with the requisite number of processes and rebuild the group, it returns the size of the new group. Otherwise, it fails.

The Amoeba Reliable Broadcast Protocol

Let us now look at how Amoeba implements group communication. 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, 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 (e.g., with *SendToGroup*) the user-supplied message is correctly delivered to all members of the group, even though the hardware may lose packets. For simplicity, we will assume that each message fits into a single packet. For the moment, we will assume that processors do not crash. We will consider the case of unreliable processors afterwards. The description given below is just an outline. For more details, see (Kaashoek and Tanenbaum, 1991; and Kaashoek et al., 1989). Other reliable broadcast protocols are discussed in (Birman and Joseph, 1987a; Chang and Maxemchuk, 1984; Garcia-Molina and Tseung, 1989).

The hardware/software configuration required for reliable broadcasting in Amoeba is shown in Fig. 7. The hardware of all the machines is normally identical, and they all run

exactly the same kernel. However, when the application starts up, one of the machines is elected as sequencer (like a committee electing a chairman). If the sequencer machine subsequently crashes, the remaining members elect a new one. Many election algorithms are known, such as choosing the process with the highest network address.

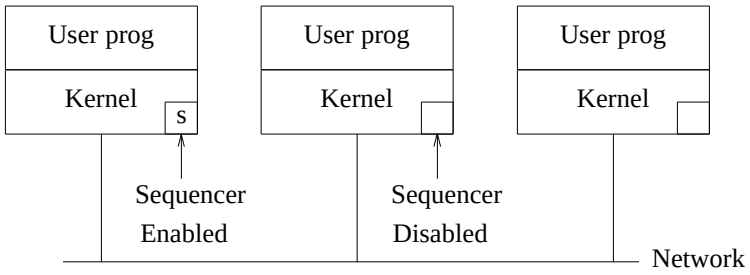

Fig. 7. System structure for group communication in Amoeba.

One sequence of events that can be used to achieve reliable broadcasting can be summarized as follows.

1. The thread traps to the kernel.
2. The thread, now in kernel mode, adds a protocol header and sends the message to the sequencer using a point-to-point message.
3. When the sequencer gets the message, it allocates the next available sequence number, puts the sequence number in the protocol header, and broadcasts the message (and sequence number).
4. When the sending kernel sees the broadcast message, it unblocks the calling process to let it continue execution.

Let us now consider these steps in more detail. When an application process executes a broadcast primitive, such as *SendToGroup*, a trap to its kernel occurs. The calling thread switches to kernel mode and builds a message containing a kernel-supplied header and the application-supplied data. The header contains the message type (*Request for Broadcast* in this case), a unique message identifier (used to detect duplicates), the number of the next broadcast expected by the kernel and some other information.

The kernel sends the message to the sequencer using a normal point-to-point message, and simultaneously starts a timer. If the broadcast comes back before the timer runs out (normal case), the sending kernel stops the timer and returns control to the caller. In practice, this case happens well over 99% of the time, because LANs are highly reliable.

On the other hand, if the broadcast has not come back before the timer expires, the kernel assumes that either the message or the broadcast has been lost. Either way, it retransmits the message. If the original message was lost, no harm has been done, and the second (or subsequent) attempt will trigger the broadcast in the usual way. If the message got to the sequencer and was broadcast, but the sender missed the broadcast, the sequencer will detect the retransmission as a duplicate (from the message identifier) and just tell the sender that everything is all right. The message is not broadcast a second time.

A third possibility is that a broadcast comes back before the timer runs out, but it is the wrong broadcast. This situation arises when two processes attempt to broadcast simultaneously. One of them, *A*, gets to the sequencer first, and its message is broadcast. *A* sees the

broadcast and unblocks its application program. However its competitor, *B*, sees *A*'s broadcast and realizes that it has failed to go first. Nevertheless, *B* knows that its message probably got to the sequencer (since lost messages are rare) where it will be queued, and broadcast next. Thus *B* accepts *A*'s broadcast and continues to wait for its own broadcast to come back or its timer to expire.

Now consider what happens at the sequencer when a *Request for Broadcast* arrives there. First a check is made to see if the message is a retransmission, and if so, the sender is informed that the broadcast has already been done, as mentioned above. If the message is new (normal case), the next sequence number is assigned to it, and the sequencer counter is incremented by one. The message and its identifier are then stored in a *history buffer*, and the message is then broadcast. The message is also passed to the application running on the sequencer's machine (because the broadcast does not interrupt itself).

Finally, let us consider what happens when a kernel receives a broadcast. First, the sequence number is compared to the sequence number of the most recently received broadcast. 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 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 minor time delay. All application programs see all broadcasts in the same order, even if some messages are lost.

The reliable broadcast protocol is illustrated in Fig. 8. Here the application program running on machine *A* passes a message, *M*, to its kernel for broadcasting. The kernel sends the message to the sequencer, where it is assigned sequence number 25. The message (containing the sequence number 25) is now broadcast to all machines and is also passed to the application running on the sequencer itself. This broadcast message is denoted by *M25* in the figure.

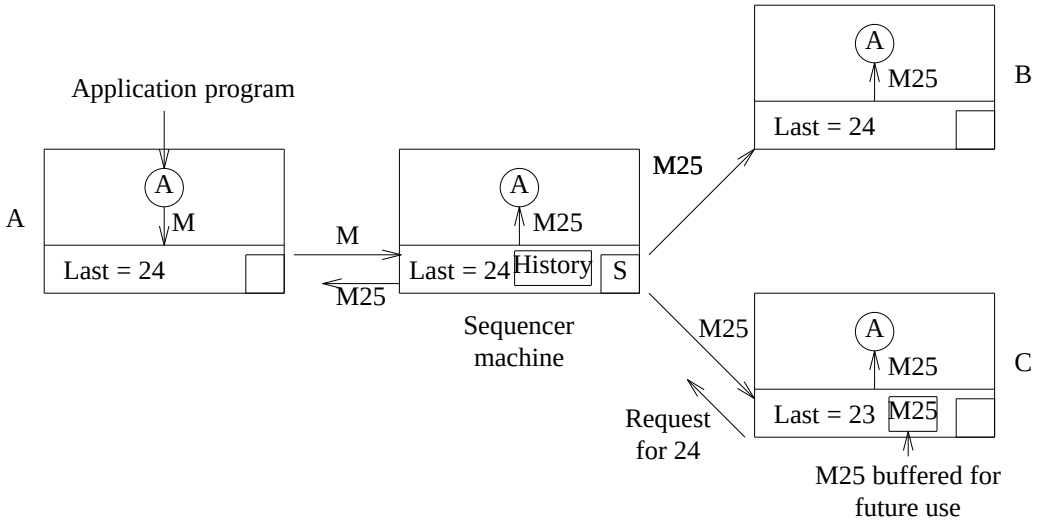

Fig. 8. The application of machine *A* sends a message to the sequencer, which then adds a sequence number (25) and broadcasts it. At *B* it is accepted, but at *C* it is buffered until 24, which was missed, can be retrieved from the sequencer.

The *M25* message arrives at machines *B* and *C*. At machine B the kernel sees that it has already processed all broadcasts up to and including 24, so it immediately passes *M25* up to the application program. At *C*, however, the last message to arrive was 23 (24 must have been lost), so *M25* is buffered in the kernel, and a point-to-point message requesting 24 is sent to the sequencer. Only after the reply has come back and been given to the application program will *M25* be passed upwards as well.

Now let us look at the management of the history buffer. Unless something is done to prevent it, the history buffer will quickly fill up. However, if the sequencer knows that all machines have correctly received broadcasts, say, 0 through 23, it can delete these from its history buffer.

Several mechanisms are provided to allow the sequencer to discover this information. The basic one is that each *Request for Broadcast* message sent to the sequencer carries a piggybacked acknowledgement, *k*, meaning that all broadcasts up to and including *k–1* have been correctly received and that it expects *k* next. This way, the sequencer can maintain a piggyback table, indexed by machine number, telling for each machine which broadcast was the last one received. Whenever the history buffer begins to fill up, the sequencer can make a pass through this table to find the smallest value. It can then safely discard all messages up to and including this value.

If a machine happens to be silent for a long period of time, the sequencer will not know what its status is. To inform the sequencer, it is required to send a short acknowledgement message when it has sent no broadcast messages for a certain period of time. Furthermore, the sequencer can broadcast a *Request for Status* message, which asks all other machines to send it a message giving the number of the highest broadcast received in sequence. In this way, the sequencer can update its piggyback table and then truncate its history buffer.

Although in practice *Request for Status* messages are rare, they do occur, and thus raise the mean number of messages required for a reliable broadcast slightly above 2, even when there are no lost messages. The effect increases slightly as the number of machines grows.

There is a subtle design point concerning this protocol that should be clarified. There are two ways to do the broadcast. In method 1 (described above), the user sends a point-to-point message to the sequencer, which then broadcasts it. In method 2, the user broadcasts the message, including a unique identifier. When the sequencer sees this, it broadcasts a special *Accept* message containing the unique identifier and its newly assigned sequence number. A broadcast is only ‘‘official’’ when the *Accept* message has been sent. The two methods are compared in Fig. 9.

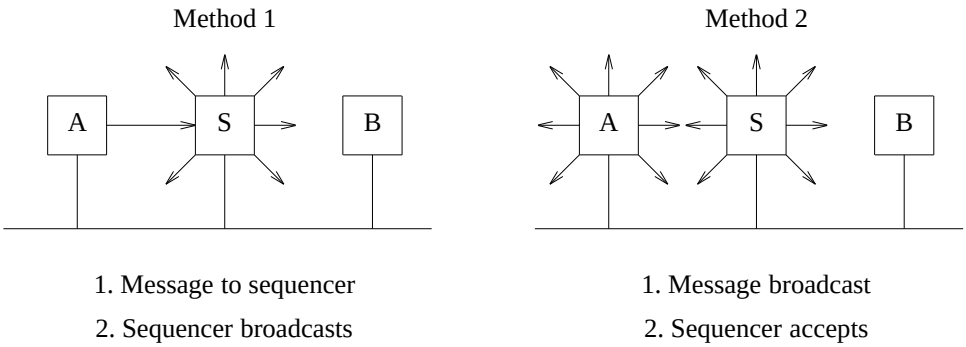

Fig. 9. Two methods for doing reliable broadcasting.

These protocols are logically equivalent, but they have different performance characteristics. In method 1, each message appears in full on the network twice: once to the sequencer

and once from the sequencer. Thus a message of length *m* bytes consumes 2*m* bytes worth of network bandwidth. However, only the second of these is broadcast, so each user machine is only interrupted once (for the second message).

In method 2, the full message only appears once on the network, plus a very short *Accept* message from the sequencer, so only half the bandwidth is consumed. On the other hand, every machine is interrupted twice, once for the message and once for the *Accept*. Thus method 1 wastes bandwidth to reduce interrupts compared to method 2. Depending on the average message size, one may be preferable to the other.

In summary, this protocol allows reliable broadcasting to be done on an unreliable network in just over two messages per reliable broadcast. Each broadcast is indivisible, and all applications receive all messages in the same order, no matter how many are lost. The worst that can happen is that a short delay is introduced when a message is lost, which rarely happens. If two processes attempt to broadcast at the same time, one of them will get to the sequencer first and win. The other will see a broadcast from its competitor coming back from the sequencer, and will realize that its request has been queued and will appear shortly, so it simply waits.

5.3. The Fast Local Internet Protocol (FLIP)

Amoeba uses a custom protocol called *FLIP* (*Fast Local Internet Protocol*) for actual message transmission (Kaashoek et al., 1991). This protocol supports both RPC and group communication and is below them in the protocol hierarchy. In OSI terms, FLIP is a network layer protocol, whereas RPC is more of a connectionless transport or session protocol (the exact location is arguable, since OSI was designed for connection-oriented networks). Conceptually, FLIP can be replaced by another network layer protocol, such as IP, although doing so would cause some of Amoeba's transparency to be lost. Although FLIP was designed in the context of Amoeba, it is intended to be useful in other operating systems as well. In this section we will describe its design and implementation.

Protocol Requirements for Distributed Systems

Before getting into the details of FLIP, it is useful to understand something about why it was designed. After all, there are plenty of existing protocols, so the invention of a new one clearly has to be justified. In Fig. 10 we list the principal requirements that a protocol for a distributed system should meet. First, the protocol must support both RPC and group communication efficiently. If the underlying network has hardware multicast or broadcast, as Ethernet does, for example, the protocol should use it for group communication. On the other hand, if the network does not have either of these features, group communication must still work exactly the same way, even though the implementation will have to be different.

A characteristic that is increasingly important is support for process migration. A process should be able to move from one machine to another, even to one in a different network, with nobody noticing. Protocols such as OSI, X.25, and TCP/IP that use machine addresses to identify processes make migration difficult, because a process cannot take its address with it when it moves.

Security is also an issue. Although the get-ports and put-ports provide security for Amoeba, a security mechanism should also be present in the packet protocol so it can be used with operating systems that do not have Amoeba-type cryptographically secure addresses.

Another point on which most existing protocols score badly is network management. It should not be necessary to have elaborate configuration tables telling which network is connected to which other network. Furthermore, if the configuration changes, due to routers (gateways) going down or coming back up, the protocol should adapt to the new configuration automatically.

Item	Description
RPC	The protocol should support RPC
Group communication	The protocol should support group communication
Process migration	Processes should be able to take their addresses with them
Security	Processes should not be able to impersonate other processes
Network management	Support should be provided for automatic reconfiguration
Wide-area networks	The protocol should work on wide-area networks

Fig. 10. Desirable characteristics for a distributed system protocol.

Finally, the protocol should work on both local and wide-area networks. In particular, the same protocol should be usable on both.

The FLIP Interface

The FLIP protocol and its associated architecture was designed to meet all these requirements, although when used on wide-area networks, it is best suited to a modest number of sites. A typical FLIP configuration is shown in Fig. 11. Here we see five machines, two on an Ethernet and four on a token ring. Each machine has one user process, *A* through *E*. One of the machines is connected to both networks, and as such automatically functions as a router. Routers may also run clients and servers, just like other nodes.

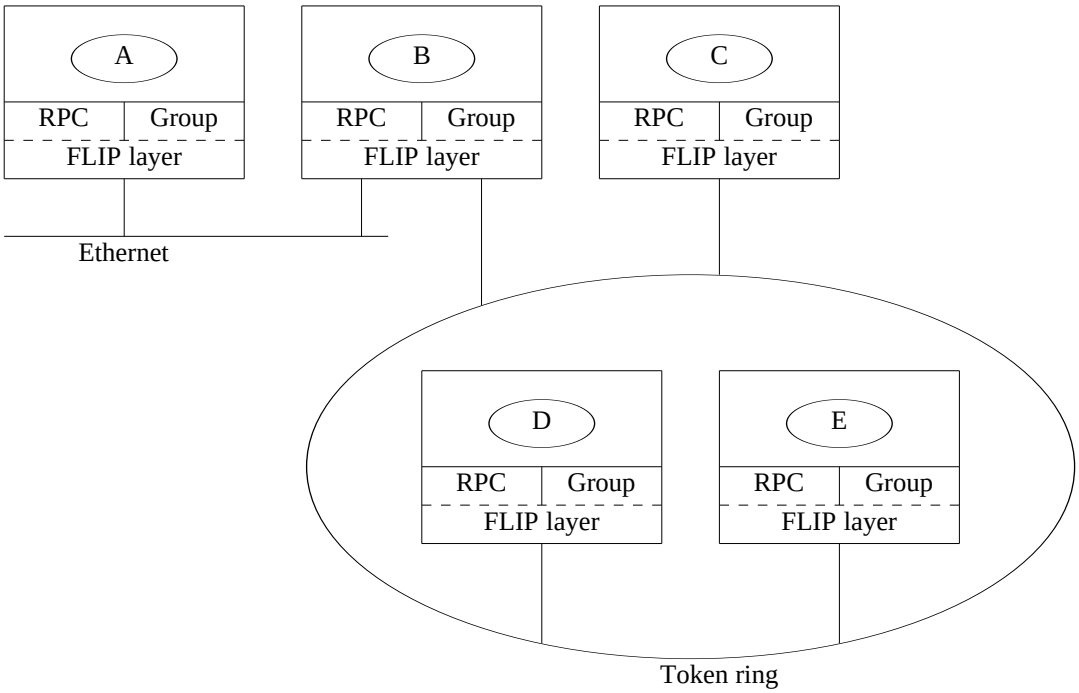

Fig. 11. A FLIP system with five machines and two networks.

The software is structured as shown in Fig. 11. The kernel contains two layers. The top layer handles calls from user processes for RPC or group communication services. The

bottom layer handles the FLIP protocol. For example, when a client calls *trans*, it traps to the kernel. The RPC layer examines the header and buffer, builds a message from them, and passes the message down to the FLIP layer for transmission.

All low-level communication in Amoeba is based on *FLIP addresses*. Each process has one or more FLIP addresses: 64-bit random numbers chosen by the system when the process is created. If the process ever migrates, it takes its FLIP address with it. If the network is ever reconfigured, so that all machines are assigned new (hardware) network numbers or network addresses, the FLIP addresses still remain unchanged. It is the fact that a FLIP address uniquely identifies a process (or a group of processes), not a machine, that makes communication in Amoeba insensitive to changes in network topology and network addressing.

A FLIP address is really two addresses, a public-address and a private-address, related by

Public-address = *DES*(private-address)

where DES is the Data Encryption Standard. To compute the public-address from the private one, the private-address is used as a DES key to encrypt a 64-bit block of 0s. Given a public-address, finding the corresponding private address is computationally infeasible. Servers listen to private-addresses, but clients send to public-addresses, analogous to the way put-ports and get-ports work, but at a lower level.

FLIP has been designed to work not only with Amoeba, but also with other operating systems. A version for UNIX also exists, although for technical reasons it differs slightly from the Amoeba version. The security provided by the private-address, public-address scheme also works for UNIX to UNIX communication using FLIP, independent of Amoeba.

Furthermore, FLIP has been designed so that it can be built in hardware, for example, as part of the network interface chip. For this reason, a precise interface with the layer above it has been specified. The interface between the FLIP layer and the layer above it (which we will call the RPC layer) has nine primitives, seven for outgoing traffic and two for incoming traffic. Each one has a library procedure that invokes it. The nine calls are listed in Fig. 12.

Call	**Description**	**Direction**
Init	Allocate a table slot	Down
End	Return a table slot	Down
Register	Listen to a FLIP address	Down
Unregister	Stop listening to a FLIP address	Down
Unicast	Send a point-to-point message	Down
Multicast	Send a multicast message	Down
Broadcast	Send a broadcast message	Down
Receive	Packet received	Up
Notdeliver	Undeliverable packet received	Up

Fig. 12. The calls supported by the FLIP layer.

The first one, *init*, allows the RPC layer to allocate a table slot and initialize it with pointers to two procedures (or in a hardware implementation, two interrupt vectors). These procedures are the ones called when normal and undeliverable packets arrive, respectively. *End* deallocates the slot when the process is being shut down.

Register is invoked to register a process' FLIP address (or a group address) with the FLIP layer. It is called when the process starts up (or at least, on the first attempt at getting or

sending a message). The FLIP layer immediately runs the private-address offered to it through the DES function, and stores the public-address in its tables. If an incoming packet is addressed to the public FLIP address, it will be passed to the RPC layer for delivery. The *unregister* call removes an entry from the FLIP layer's tables. The difference between *end* and *unregister* is that a process may use multiple FLIP addresses. *Unregister* removes one of these from the table, but leaves the others. When no more communication is needed, *end* is called to free the interface slot.

The next three calls are for sending point-to-point messages, multicast messages, and broadcast messages, respectively. None of these guarantee delivery. To make RPC reliable, acknowledgements are used. To make group communication reliable, even in the fact of lost packets, the sequencer protocol discussed above is used.

The last two calls are for incoming traffic. The first is for messages originating elsewhere and directed to this machine. The second is for messages sent by this machine but sent back as undeliverable.

Operation of the FLIP Layer

Packets passed by the RPC layer or group communication layer (see Fig. 11) to the FLIP layer are addressed by FLIP addresses, so the FLIP layer must be able to convert these addresses to network addresses for actual transmission. In order to perform this function, the FLIP layer maintains the routing table shown in Fig. 13. Currently this table is maintained in software, but future chip designers could implement it in hardware.

FLIP address	Network address	Hop count	Trusted bit	Age

Fig. 13. The FLIP routing table.

Whenever an incoming packet arrives at any machine, it is first handled by the FLIP layer, which extracts from it the FLIP address and network address of the sender. The number of hops the packet has made is also recorded. Since the hop count is only incremented when a packet is forwarded by a router, the hop count tells how many routers the packet has passed through. The hop count is therefore a crude measure of how far away the source is. (Actually, things are slightly better than this, as slow networks count for multiple hops, with the weight a function of the network speed.) If the FLIP address is not presently in the routing table, it is entered. This entry can later be used to send packets *to* that FLIP address, since its network number and address are now known.

An additional bit present in each packet tells whether the path the packet has followed so far is entirely over trusted networks. It is managed by the routers. If the packet has gone through one or more untrusted networks, packets to the source address should be encrypted if absolute security is desired. With trusted networks, encryption is not needed.

The last field of each routing table entry gives the age of the routing table entry. It is reset to 0 whenever a packet is received from the corresponding FLIP address. Periodically, all the ages are incremented. This field allows the FLIP layer to find a suitable table entry to purge if the table fills up (large numbers indicate that there has been no traffic for a long time).

Locating Put-Ports

To see how FLIP works in the context of Amoeba, let us consider a simple example using the configuration of Fig. 11. *A* is a client and *B* is a server. With FLIP, any machine having connections to two or more networks is automatically a router, so the fact that *B* happens to be running on a router machine is irrelevant.

When *B* is created, the RPC layer picks a new random FLIP address for it and registers it with the FLIP layer. After starting, *B* initializes itself and then does a *get_request* on its get-port, which causes a trap to the kernel. The RPC layer computes the put-port from the get-port and makes a note that a process is listening to that port. It then blocks until a request comes in.

Later, *A* does a *trans* on the put-port. Its RPC layer looks in its tables to see if it knows the FLIP address of the server process that listens to the put-port. Since it does not, the RPC layer sends a special broadcast packet to find it. This packet has a maximum hop count of 1 to make sure that the broadcast is confined to its own network. (When a router sees a packet whose current hop count is already equal to its maximum hop count, the packet is discarded instead of being forwarded.) If the broadcast fails, the sending RPC layer times out and tries again with a maximum hop count of 2, and so on, until it locates the server.

When the broadcast packet arrives at *B*'s machine, the RPC layer there sends back a reply announcing its get-port. This packet, like all incoming packets, causes *A*'s FLIP layer to make an entry for that FLIP address before passing the reply packet up to the RPC layer. The RPC layer now makes an entry in its own tables mapping the put-port onto the FLIP address. Then it sends the request to the server. Since the FLIP layer now has an entry for the server's FLIP address, it can build a packet containing the proper network address and send it without further ado. Subsequent requests to the server's put-port use the RPC layer's cache to find the FLIP address and the FLIP layer's routing table to find the network address. Thus broadcasting is only used the very first time a server is contacted. After that, the kernel tables provide the necessary information.

To summarize, locating a put-port requires two mappings:

1. From the put-port to the FLIP address (done by the RPC layer).
2. From the FLIP address to the network address (done by the FLIP layer).

The reason for this two-stage process is twofold. First, FLIP has been designed as a general-purpose protocol for use in distributed systems, including non-Amoeba systems. Since these systems generally do not use Amoeba-style ports, the mapping of put-ports to FLIP addresses has not been built into the FLIP layer. Other users of FLIP may just use FLIP addresses directly.

Second, a put-port really identifies a *service* rather than a *server*. A service may be provided by multiple servers to enhance performance and reliability. Although all the servers listen to the same put-port, each one has its own private FLIP address. When a client's RPC layer issues a broadcast to find the FLIP address corresponding to a put-port, any or all of the servers may respond. Since each server has a different FLIP address, each response creates a different entry in the put-port table of Fig. 5.

The advantage of this scheme over having just a single (port, network address) cache is that it permits servers to migrate to new machines or have their machines be wheeled over to new networks and plugged in without requiring any manual reconfiguration, as say, TCP/IP does. There is a strong analogy here with a person moving and being assigned the same telephone number at the new residence as he had at the old one. (For the record, Amoeba does not currently support process migration, but it could be added later.)

The advantage over having clients and servers use FLIP addresses directly is that FLIP addresses are temporary, whereas ports may be valid for a long time. If a server crashes, it

will pick a new FLIP address when it reboots. Attempts to use the old FLIP address will time out, allowing the RPC layer to indicate failure to the client. This mechanism is how at-most-once semantics are guaranteed. The client, however, can just try again with the same put-port if it wishes, since that is not necessarily invalidated by server crashes.

FLIP over Wide-Area Networks

FLIP also works transparently over wide-area networks. In Fig. 14 we have three local-area networks connected by a wide-area network. Suppose the client *A* wants to do an RPC with the server *E*. *A*'s RPC layer first tries to locate the put-port using a maximum hop count of 1. When that fails, it tries again with a maximum hop count of 2. This time, *C* forwards the broadcast packet to all the routers that are connected to the wide-area network, namely, *D* and *G*. Effectively, *C* simulates broadcast over the wide-area network by sending individual messages to all the other routers. When this broadcast fails to turn up the server, a third broadcast is sent, this time with a maximum hop count of 3. This one succeeds. The reply contains *E*'s network address and FLIP address, which are then entered into *A*'s routing table. From this point on, communication between *A* and *E* happens using normal point-to-point communication. No more broadcasts are needed.

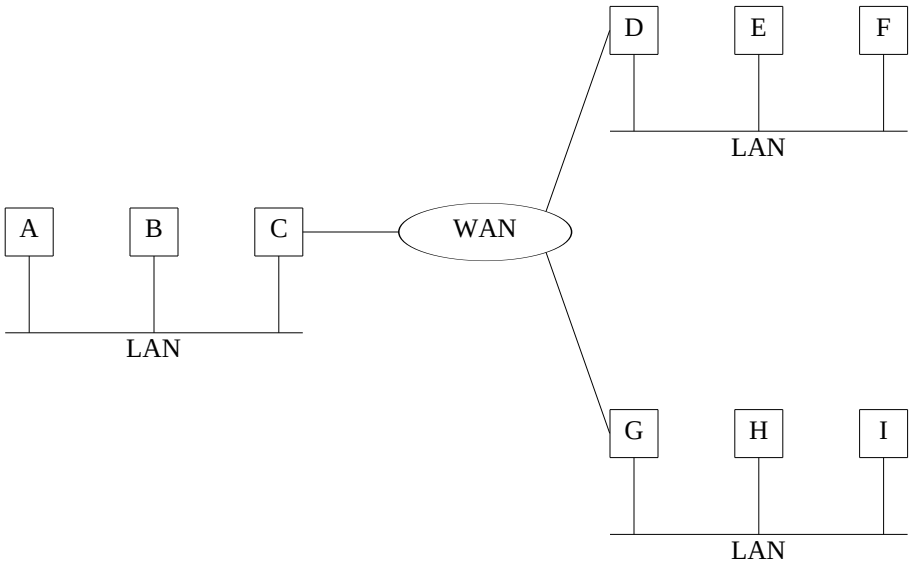

Fig. 14. Three LANs connected by a WAN.

Communication over the wide-area network is encapsulated in whatever protocol the wide-area network requires. For example, on a TCP/IP network, *C* might have open connections to *D* and *G* all the time. Alternatively, the implementation might decide to close any connection not used for a certain length of time.

Although this method does not scale to large networks, we expect that for modest numbers it may be usable, based on our initial experiments with an internetwork of five networks on two continents. In practice, few servers move between sites, so that once a server has been located by broadcasting, subsequent requests will use the cached entries. Using this method, a modest number of machines all over the world can work together in a totally transparent way. An RPC to a thread in the caller's address space and an RPC to a thread halfway around the world are done in exactly the same way.

Group communication also uses FLIP. When a message is sent to multiple destinations,

FLIP uses the hardware multicast or broadcast on those networks where it is available. On those that do not have it, broadcast is simulated by sending individual messages, just as we saw on the wide-area network. The choice of mechanism is done by the FLIP layer, with the same user semantics in all cases.

6. SUMMARY

The Amoeba microkernel manages process and memory, and handles communication. At the lowest level, processes are started by generating a process descriptor and doing an RPC with the kernel thread responsible for starting new processes on the target machine. Higher level services help with locating a suitable machine to run on.

Memory is based on the concept of segments, which can be mapped into and out of processes address spaces at arbitrary addresses.

Communication comes in two forms: RPC and group communication. The former is for sending point-to-point messages, and the latter is for sending many-to-one messages reliable, in the face of various errors. Both use the FLIP protocol for actual data transport.

Amoeba has been operational for several years. It is now available for use. Interested parties should contact the first author by electronic mail.

7. ACKNOWLEDGEMENTS

We would like to thank Greg Sharp and Mark Wood for their comments on the manuscript.

8. REFERENCES

1. Birman, K.P. and Joseph, T.: ‘‘Reliable Communication in the Presence of Failures,’’ *ACM Trans. on Computer Systems*, vol. 5, pp. 47-76, Feb. 1987.

2. Birrell, A.D., and Nelson, B.J. Implementing Remote Procedure Calls, *ACM Trans. Comput. Systems* vol. 2, pp. 39-59, Feb. 1984.

3. Chang, J. and Maxemchuk. N.F.: ‘‘Reliable Broadcast Protocols,’’ *ACM Trans. on Computer Systems*, vol. 2, pp. 251-273, Aug. 1984.

4. Evans, A., Kantrowitz, W., and Weiss, E. A User Authentication Scheme Not Requiring Secrecy in the Computer. *Commun. ACM* vol. 17, pp. 437-442, Aug. 1974.

5. Garcia-Molina, H.: ‘‘Elections in a Distributed Computing System,’’ *IEEE Trans. on Computers*, vol. 31, pp. 48-59, Jan. 1982.

6. Kaashoek, M.F., Renesse, R. van, Staveren, H. van, and Tanenbaum, A.S.: "FLIP: an Internetwork Protocol for Supporting Distributed Systems," Report IR-251, Dept. of Math. and Computer Science, Vrije Univ., 1991.

7. 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 Magazine*, vol. 23, pp. 44-53, May 1990.

8. Tanenbaum, A.S., Mullender, S.J., and van Renesse, R.: "Using Sparse Capabilities in a Distributed Operating System" *Proc. Sixth International Conf. on Distr. Computer Systems*, IEEE, pp. 558-663, 1986.

9. 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.

10. Tanenbaum, A.S., Mullender, S.J., and van Renesse, R.: "Using Sparse Capabilities in a Distributed Operating System" *Proc. Sixth International Conf. on Distr. Computer Systems*, IEEE, 1986.

11. Tseung, L.N.: ‘‘Guaranteed, Reliable, Secure Broadcast Networks,’’ *IEEE Network Magazine*, vol. 3, pp. 33-37. Nov. 1989.