GROUP COMMUNICATION IN AMOEBA AND ITS APPLICATIONS

M. Frans Kaashoek†
Andrew S. Tanenbaum
Kees Verstoep

Dept. of Math. and Comp. Sci.
Vrije Universiteit
Amsterdam, The Netherlands

Email: kaashoek@lcs.mit.edu, ast@cs.vu.nl, and versto@cs.vu.nl.

ABSTRACT

Unlike many other operating systems, Amoeba is a distributed operating system that provides group communication (i.e., one-to-many communication). We will discuss design issues for group communication, Amoeba's group system calls, and the protocols to implement group communication. To demonstrate that group communication is a useful abstraction, we will describe a design and implementation of a fault-tolerant directory service. We discuss two versions of the directory service: one with Non-Volatile RAM (NVRAM) and one without NVRAM. We will give performance figures for both implementations.

1. Introduction

Most current distributed operating systems provide only *Remote Procedure Call* (RPC) [6]. The idea is to hide the message passing, and make the communication look like an ordinary procedure call (see Figure 1). The sender, called the *client*, calls a *stub routine* on its own machine that builds a message containing the name of the procedure to be called and all the parameters. It then passes this message to the driver for transmission over the network. When it arrives, the remote driver gives it to a stub, which unpacks the message and makes an ordinary procedure call to the *server*. The reply from server to client follows the reverse path.

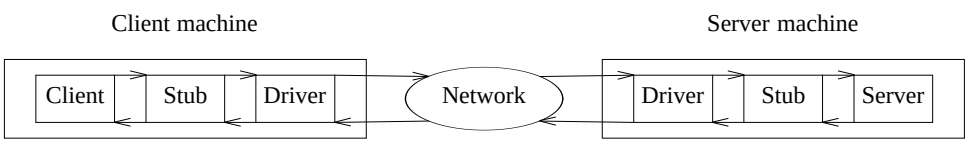

Figure 1: *Remote procedure call from a client to a server.*

† Current affiliation: Lab for Computer Science, MIT, Cambridge MA.

Although RPC is a very useful communication paradigm, many applications need something more. RPC is inherently point-to-point communication, but what often is needed is 1-to-*n* communication. Consider, for example, a parallel application. Typically in a parallel application a number of processes cooperate to compute a single result. If one of the processes finds a partial result (e.g., a better bound in a parallel branch-and-bound program) it is often necessary that this partial result is communicated immediately to the other processes, so that they do not waste cycles on computing something that is not interesting anymore, given the new partial result. What is needed here is a way to send a message from 1 process to *n* processes. This abstraction is called *group communication*.

Now consider a second application: a fault-tolerant storage service. A reliable storage service can be built by replicating data on multiple processors each with its own disk. If a piece of data needs to be changed, the service either has to send the new data to all processes or invalidate all other copies of the changed data. If only point-to-point communication were available, then the process would have to send $n - 1$ reliable point-to-point messages. In most systems this will cost at least $2(n - 1)$ messages (one packet for the actual message and one packet for the acknowledgement). If the message sent by the server has to be fragmented into multiple network packets, then the cost will be even higher. This method is slow, inefficient, and wasteful of network bandwidth.

In addition to being expensive, building distributed applications using only point-to-point communication is often difficult. If, for example, two servers in the reliable storage service receive a request to update the same data, they need a way to order the updates, otherwise the data may become inconsistent. The problem is illustrated in Figure 2. The copies of variable *x* become inconsistent because the messages from Server 1 and Server 2 are not ordered. The problem of ordering is not restricted to point-to-point communication, however.

Many network designers have realized that group communication is an important tool for building distributed applications; broadcast communication is provided by many networks, including LANs, geosynchronous satellites, and cellular radio systems [33]. Several commonly used LANs, such as Ethernet and some rings, even provide multicast communication. Using multicast communication, messages can be sent exactly to the group of processes that are interested in receiving them. Future networks, like Gigabit LANs, are also likely to implement broadcasting and/or multicasting to support high-performance applications such as multimedia [22].

The protocol presented in this paper for group communication uses the hardware multicast capability of a network, if one exists. Otherwise, it uses broadcast messages or point-to-point messages, depending on the size of the group and the availability of broadcast communication. Thus, Amoeba makes the hardware support for group communication available to application programs.

The outline of the rest of the paper is as follows. In Section 2, we will discuss design issues in group communication. In Section 3, we will discuss the Amoeba group system calls. Section 4 gives an overview of the protocols that are used to implement group communication in Amoeba. In Section 5, we will describe the design and implementation of a distributed application using group communication: a fault-tolerant directory service. In Section 6, we will give performance figures for two implementations of the directory service. In Section 7, some conclusions will be drawn.

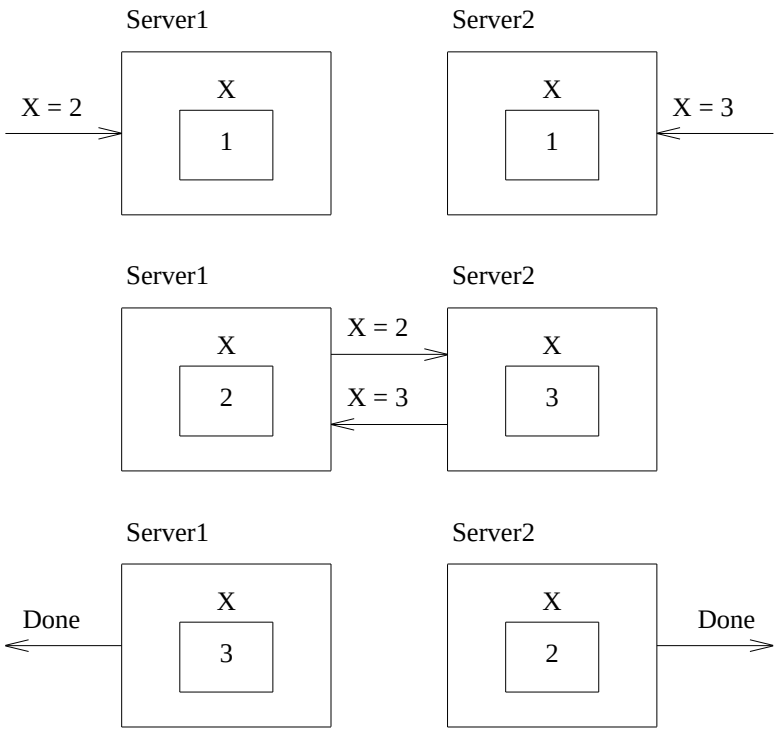

Figure 2: *Inconsistency due to lack of message ordering.*

2. Design Issues in Group Communication

Few existing operating systems provide application programs with support for group communication. To understand the differences between these existing systems, six design criteria are of interest: addressing, reliability, ordering, delivery semantics, response semantics, and group structure (see Table 1). We will discuss each one in turn.

Issue	**Description**
Addressing	Addressing method for a group (e.g., list of members)
Reliability	Reliable or unreliable communication
Ordering	Order among messages (e.g., total ordering)
Delivery semantics	How many processes must receive the message successfully
Response semantics	How to respond to a broadcast message
Group structure	Semantics of a group (e.g., dynamic versus static)

Table 1: *The main design issues for group communication.*

Four methods of *addressing* messages to a group exist. The simplest one is to require the sender to explicitly specify all the destinations to which the message should be delivered. A second method is to use a single address for the whole group. This method saves bandwidth and also allows a process to send a message without knowing which processes are members of the group. Two less common addressing methods are *source addressing* [13], and *functional*

addressing [16]. Using source addressing, a process accepts a message if the source is a member of the group. Using functional addressing, a process accepts a message if a user-defined function on the message evaluates to true. The disadvantage of the latter two methods is that they are hard to implement with current network interfaces.

The second design criterion, *reliability*, deals with recovery from communication failures, such as buffer overflows and garbled packets. Because reliability is more difficult to implement for group communication than for point-to-point communication, a number of existing operating systems provide *unreliable* group communication, whereas almost all operating systems provide *reliable* point-to-point communication, for example, in the form of RPC.

Another important design decision in group communication is the *ordering* of messages sent to a group. Roughly speaking, there are 4 possible orderings: no ordering, FIFO ordering, causal ordering, and total ordering [5]. No ordering is easy to understand and implement, but unfortunately makes programming often harder. FIFO ordering guarantees that all messages from a member are delivered in the order in which they were sent. Causal ordering guarantees that all messages that are related are ordered. More specifically: messages are in FIFO order and if a member after receiving message *A* sends a message *B*, it is guaranteed that all members will receive *A* before *B*. In the total ordering, each member receives all messages in the same order. The last ordering is stronger than any of the other orderings and makes programming easier, but it is harder to implement.

To illustrate the difference between FIFO and total ordering, consider a service that stores records for client processes. Assume that the service replicates the records on each server to increase availability and reliability and that it guarantees that all replicas are consistent. If a client may only update its own records, then it is sufficient that all messages from the same client will be ordered. Thus, in this case FIFO ordering can be used. If a client may update any of the records, then FIFO ordering is not sufficient. A total ordering on the updates, however, is sufficient to ensure consistency among the replicas. To see this, assume that two clients, C_1 and C_2, send an update for record *X* at the same time. As these two updates will be totally-ordered, all servers either (1) receive first the update from C_1 and then the update from C_2 or (2) receive first the update from C_2 and then the update from C_1. In either case, the replicas will stay consistent, because every server applies the updates in the same order. If in this case FIFO (or causal) ordering had been used, it might have happened that the servers applied the updates in different orders, resulting in inconsistent replicas.

The fourth item in the table, *delivery semantics* relates to when a message is considered delivered successfully to a group. There are 3 choices: *k*-delivery, quorum delivery, and atomic delivery. With *k*-delivery, a broadcast is successful when *k* processes have received the message for some constant *k*. With quorum delivery, a broadcast is defined as being successful when a majority of the current membership has received it. With atomic delivery either all processes receive it or none do. For many applications atomic delivery is the ideal semantics, but is harder to implement if processors can fail.

Item five, *response semantics* deals with what the sending process expects from the receiving processes [17]. There are four broad categories of what the sender can expect: no responses, a single response, many responses, and all responses. Operating systems that integrate group communication and RPC completely support all four choices [9].

The last design decision specific to group communication is *group structure*. Groups can be either closed or open [23]. In a *closed* group, only members can send messages to the group. In an *open* group, nonmembers may also send messages to the group. In addition, groups can be static or dynamic. In static groups processes cannot leave or join a group, but remain a member of the group for the lifetime of the process. Dynamic groups may have a varying number of

members over time. If processes can be members of multiple groups, the semantics for *overlapping groups* must be defined. Suppose that two processes are members of both groups G_1 and G_2 and that each group guarantees a total ordering. A design decision has to be made about the ordering between the messages of G_1 and G_2. All choices discussed in this section (none, FIFO, causal, and total ordering) are possible.

To make these design decisions more concrete, we briefly discuss two systems that support group communication. Both systems support open dynamic groups, but differ in their semantics for reliability and ordering. In the V system [9], groups are identified with a group identifier. If two processes concurrently broadcast two messages, *A* and *B*, respectively, some of the members may receive *A* first and others may receive *B* first. No guarantees about ordering are given. Group communication in the V system is unreliable. Users can, however, build their own group communication primitives with the basic primitives. They could, for example, implement a protocol with reliable communication and total ordering as a library package.

In the Isis system [4], messages are sent to a group identifier or to a list of addresses. When sending a message, a user specifies how many replies are expected. Messages can be totally-ordered, even for groups that overlap. Reliability in Isis means that either *all* or *no* members of a group will receive a message, even in the face of processor failures. Because these semantics are hard to implement efficiently, Isis also provides primitives that give weaker semantics, but better performance. It is up to the programmer to decide which primitive is required.

Recently the protocols for Isis have been redesigned [5]. The system is now completely based on a broadcast primitive that provides causal ordering. The implementation of this primitive uses reliable point-to-point communication. The protocol for totally-ordered broadcast is based on causal broadcast. The new version of Isis no longer supports a total ordering for overlapping groups.

3. Group Communication in Amoeba

Amoeba is a distributed operating system based on the client/server model [34, 27]. Services in Amoeba are addressed by *port*s, which are large random numbers. When a service is started, it generates a new port and registers the port with the directory service. A client can look up the port using the directory service and ask its own kernel to send a message to the given port. The kernel will map the port on a network address. If multiple servers listen to the same port, only one (arbitrary) server will get the message.

Ports are also used to identify groups. RPC and group primitives that are called with the same port do not interfere with each other. When creating a group, the user specifies a port. Other processes can use this port, for example, to join the group or to send a message to the group. Thus, in Amoeba all entities, processes and groups, are addressed in a uniform way.

Groups in Amoeba are closed. A process that is not a member and that wishes to communicate with a group can use RPC (or it can join the group). The reason for doing so is that a client need not be aware whether a service consists of multiple servers which perhaps broadcast messages to communicate with one another, or a single server. Also, a service should not have to know whether the client consists of a single process or a group of processes. This design decision is in the spirit of the client-server paradigm: a client knows what operations are allowed, but should not know how these operations are implemented by the service.

The primitives to manage groups and to communicate within a group are given in Table 2. We will discuss the most important one: *SendToGroup*. This primitive guarantees that *hdr* and *buf* will be delivered to all members, even in the face of unreliable communication and finite buffers. Furthermore, when the *resilience degree* of the group is *r* (as specified in

Function(parameters) → result	**Description**
CreateGroup(port, resilience, max_group, nr_buf, max_msg) → gd	Create a group. A process specifies how many member failures must be tolerated without loss of any message.
JoinGroup(hdr) → gd	Join a specified group.
LeaveGroup(gd, hdr)	Leave a group. The last member leaving causes the group to vanish.
SendToGroup(gd, hdr, buf, bufsize)	Atomically send a message to all the members of the group. All messages are totally-ordered.
ReceiveFromGroup(gd, &hdr, &buf, bufsize, &more) → size	Block until a message arrives. *More* tells if the system has buffered any other messages.
ResetGroup(gd, hdr, nr_members) → group_size	Recover from processor failure. If the newly reset group has at least *nr_member* members, it succeeds.
GetInfoGroup(gd, &state)	Return state information about the group, such as the number of group members and the caller's member id.
ForwardRequest(gd, member_id)	Forward a request for the group to another group member.

Table 2: *Primitives to manage a group and to communicate within a group.*

CreateGroup), the protocol guarantees that even in the event of a simultaneous crash of up to *r* members, it will either deliver the message to all remaining members or to none. Choosing a large value for *r* provides a high degree of fault tolerance, but extracts a penalty in performance. The tradeoff chosen is up to the user.

In addition to reliability, the protocol guarantees that messages are delivered in the same order to all members. Thus, if two members (on two different machines), simultaneously broadcast two messages, *A* and *B*, the protocol guarantees that either

(1) All members receive *A* first and then *B*, or

(2) All members receive *B* first and then *A*.

Random mixtures, where some members get *A* first and others get *B* first, are guaranteed not to occur. Application programs can count on it.

Table 3 lists the design issues and the choices for Amoeba. To summarize, the group primitives provide an abstraction that enables programmers to design applications consisting of multiple processes running (typically) on different machines. It is a simple, but powerful, abstraction. All members of a group see all events in the same order. Even the events of a new member joining the group, a member leaving the group, and recovery from a crashed member are totally-ordered. If, for example, one process calls *JoinGroup* and a member calls *SendToGroup*, either all members first receive the join and then the broadcast or all members first receive the broadcast and then the join. In the first case the process that called *JoinGroup* will also receive the broadcast message. In the second case, it will not receive the broadcast message. A mixture of

these two orderings is guaranteed not to happen. This property makes reasoning about a distributed application much easier. Furthermore, the group interface gives support for building fault tolerant applications by choosing an appropriate resilience degree.

Issue	**Choice**
Addressing	Group identifier (port)
Reliability	Reliable communication; fault tolerance if specified
Ordering	Total ordering per group
Delivery semantics	All or none
Response semantics	None (RPC is available)
Group structure	Closed and dynamic

Table 3: *Important design issues of Table 1 and the choices made for Amoeba.*

4. Implementation of Group Communication

In this section we will describe the Amoeba group communication protocol. Many other protocols exist which implement similar semantics [5, 8, 12, 28, 1]. A detailed comparison between these, our and other protocols with respect to ordering semantics, fault-tolerance and performance can be found in [18].

The protocol to be described runs inside the kernel and is accessible through the primitives described in the previous section. It assumes that *unreliable* message passing between processes is possible; fragmentation, reassembly, and routing of messages are done at lower layers in the kernel [21]. The protocol performs best on a network that supports hardware multicast. Lower layers, however, treat multicast as an optimization of sending point-to-point messages; if multicast is not available, then point-to-point communication will be used. Even if only point-to-point communication is available, the protocol is in most cases still more efficient than performing n RPCs. (In a mesh interconnection network, for example, the routing protocol will ensure that the delay of sending n messages is only in the order of $\log_2 n$.)

Each kernel running a group member maintains information about the group (or groups) to which the member belongs. It stores, for example, the size of the group and information about the other members in the group. Any group member can, at any instant, decide to broadcast a message to its group. It is the job of the kernel and the protocol to achieve reliable broadcasting, even in the face of unreliable communication, lost packets, finite buffers, and node failures. We assume, however, that Byzantine failures (in which a kernel sends malicious or contradictory messages) do not occur.

Without loss of generality, we assume for the remainder of this section that the system contains one group, with each member running on a separate processor. All machines run exactly the same kernel and application software. However, when the application starts up, the machine on which the group is created is made the *sequencer*. If the sequencer machine subsequently crashes, the remaining members elect a new one. The sequencer machine is in no way special—it has the same hardware and runs the same kernel as all the other machines. The only difference is that it is currently performing the sequencer function.

Basic protocol

A brief description of the protocol is as follows (a complete description is given in [18]). When a group member calls *SendToGroup* to send a message, *M,* it hands the message to its kernel and is blocked. The kernel encapsulates *M* in an ordinary point-to-point message and sends it to the sequencer. When the sequencer receives *M*, it allocates the next sequence number, *s*, and broadcasts a message containing *M* and *s*. Thus all broadcasts are issued from the same node, the sequencer. Assuming that no messages are lost, it is easy to see that if two members concurrently want to broadcast, one of them will reach the sequencer first and its message will be broadcast first. Only when that broadcast has been completed will the other broadcast be started. Thus, the sequencer provides a total time ordering. In this way, we can easily guarantee the indivisibility of broadcasting per group.

When the kernel that sent *M*, receives the message from the network, it knows that its broadcast has been successful. It unblocks the member that called *SendToGroup*.

Although most modern networks are highly reliable, they are not perfect, so the protocol must deal with errors. Suppose some node misses a broadcast packet, either due to a communication failure or lack of buffer space when the packet arrived. When the following broadcast message eventually arrives, the kernel will immediately notice a gap in the sequence numbers. If it was expecting *s* next, and it receives $s + 1$ instead, it knows it has missed one.

The kernel then sends a special point-to-point message to the sequencer asking it for a copy of the missing message (or messages, if several have been missed). To be able to reply to such requests, the sequencer stores broadcast messages in the *history buffer*. The sequencer sends the missing messages to the process requesting them as point-to-point messages. The other kernels also keep a history buffer, to be able to recover from sequencer failures and to buffer messages when there is no outstanding *ReceiveFromGroup* call.

As a practical matter, a kernel has only a finite amount of space in its history buffer, so it cannot store broadcast messages indefinitely. However, if it could somehow discover that all members have received broadcasts up to and including *m*, it could then purge the broadcast messages up to *m* from the history buffer.

The protocol has several ways of letting a kernel discover this information. For one thing, each point-to-point message to the sequencer (e.g., a broadcast request), contains, in a header field, the sequence number of the last broadcast received by the sender of the message (i.e., a piggybacked acknowledgement). This information is also included in the message from the sequencer to the other kernels. In this way, a kernel can maintain a table, indexed by member number, showing that member *i* has received all broadcast messages up to T_i (and perhaps more). At any instant, a kernel can compute the lowest value in this table, and safely discard all broadcast messages up to and including that value. For example, if the values of this table are 8, 7, 9, 8, 6, and 8, the kernel knows that everyone has received broadcasts 0 through 6, so they can safely be deleted from the history buffer. If a node does not do any broadcasting for a while, the sequencer will not have an up-to-date idea of which broadcasts it has received. To provide this information, nodes that have been quiet for a certain interval send the sequencer a special packet acknowledging all received broadcasts. The sequencer can also request this information when it runs out of space in its history buffer.

PB method and BB method

There is a subtle design point in the protocol; there are actually two ways to do a broadcast. In the method we have just described, the sender sends a point-to-point message to the sequencer, which then broadcasts it. We call this the *PB method* (Point-to-point followed by a Broadcast).

In the *BB method*, the sender broadcasts the message. When the sequencer sees the broadcast, it broadcasts a special *accept* message containing the newly assigned sequence number. A broadcast message is only ''official'' when the *accept* message has been sent.

These methods are logically equivalent, but they have different performance characteristics. In the PB method, each message appears on the network twice: once to the sequencer and once from the sequencer. Thus a message of length *n* bytes consumes 2*n* bytes of network bandwidth. However, only the second message is broadcast, so each user machine is interrupted only once (for the second message).

In the BB method, the full message appears only once on the network, plus a very short *accept* message from the sequencer. Thus, only about *n* bytes of bandwidth are consumed. On the other hand, every machine is interrupted twice, once for the message and once for the *accept*. Thus the PB method wastes bandwidth to reduce the number of interrupts and the BB method minimizes bandwidth usage at the cost of more interrupts. The protocol switches dynamically between the PB method and BB method depending on the message size.

Processor failures

The protocol described so far recovers from communication failures, but does not guarantee that all surviving members receive all messages that have been sent before a member crashed. For example, suppose a process sends a message to the sequencer, which broadcasts it. The sender receives the broadcast and delivers it to the application, which interacts with the external world. Now assume all other processes miss the broadcast, and the sender and sequencer both crash. Now, the effects of the message are visible but none of the other members will receive it. This is a dangerous situation that can lead to all kinds of disasters, because the ''all-or-none'' semantics have been violated.

To avoid this situation, *CreateGroup* has a parameter *r*, the *resilience degree* that specifies the resiliency. This means that the *SendToGroup* primitive does not return control to the application until the kernel knows that at least *r* other kernels have received the message. To achieve this, a kernel sends the message to the sequencer point-to-point (PB method) or broadcasts the message to the group (BB method). The sequencer allocates the next sequence number, but does not officially accept the message yet. Instead, it buffers the message and broadcasts the message and sequence number as a request for broadcasting to the group. On receiving such a request with a sequence number, kernels buffer the message in their history and the *r* lowest-numbered send acknowledgement messages to the sequencer. After receiving these acknowledgements, the sequencer broadcasts the *accept* message. Only after receiving the *accept* message can members other than the sequencer deliver the message to the application. That way, no matter which *r* machines crash, there will be at least one left containing the full history, so everyone else can be brought up-to-date during the recovery. Thus, an increase in fault tolerance is paid for by a decrease in performance. The tradeoff chosen is up to the user.

5. An Application of Group Communication: a Fault-tolerant Directory Service

The group communication primitives have been used in parallel applications [3, 35], and in a fault-tolerant implementation of the Orca programming language [19]. In this section, we discuss a fault-tolerant design and implementation of Amoeba's directory service. The directory service exemplifies distributed services that provide high reliability and availability by replicating data. For example, Amoeba's Bullet file service [29], is currently also being made fault-tolerant using active replication and group communication.

The directory service is a vital service in the Amoeba distributed operating system [30]. It provides among other things a mapping from ASCII names to capabilities. In its simplest form a

directory is basically a table with 2 columns: one storing the ASCII string and one storing the corresponding capability. Capabilities in Amoeba identify an object (e.g., a file). The set of capabilities a user possesses determines which objects it can access and which not. The directory service allows the users to store these capabilities under ASCII names to make life easier for them.

The previous design and implementation of the directory service is based on RPC [30]. The RPC directory service is duplicated and recovers therefore only from one processor failure. Furthermore, it cannot tolerate network partitions. We will now discuss the design and implementation of a directory service based on group communication. A comparison of the two directory services can be found in [20].

The group directory service is triplicated (though four or more replicas are also possible, without changing the protocol) and uses active replication. Also, it allows network partitions. To keep the copies consistent, it uses a modified version of read-one write-all policy, called *accessible copies* [11]. Recovery is based on the protocol described by Skeen [32]. The main purpose of this section is to describe a fault-tolerant service based on group communication. Other projects have implemented similar services [25, 31, 15, 26, 7, 24, 2].

The organization of the group directory service is depicted in Figure 3. The directory service is currently built out of three directory servers, each using its own Bullet file server and disk server. A Bullet server and a disk server share one disk. Each directory server stores a copy of all directories.

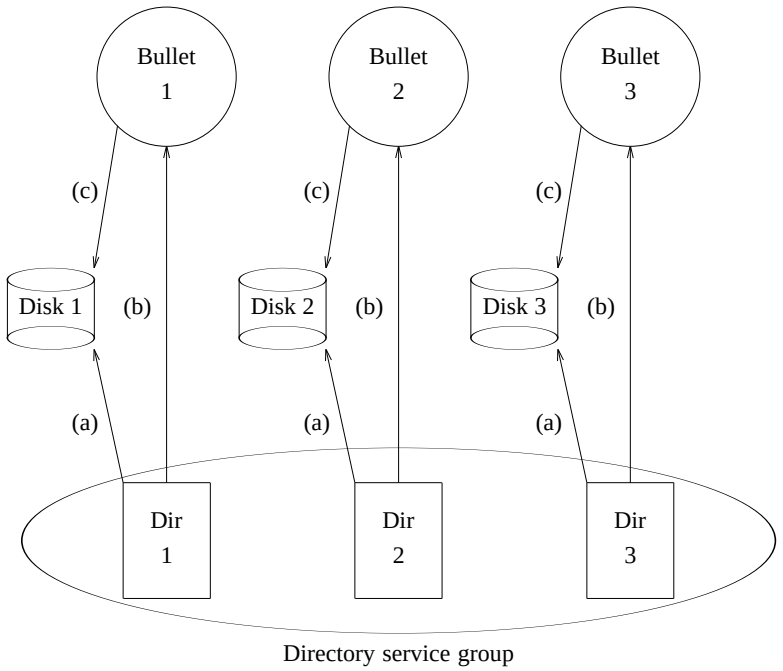

Figure 3: *Organization of the service (a) Administrative data; (b) Directories; (c) Files.*

The directory servers form a group with a resilience degree, *r*, of 2. This means that if *SendToGroup* returns successfully, it is guaranteed, even if two processors fail, that the message still will be delivered to the third one. Furthermore, it is guaranteed even in the presence of communication and processor failures that each server will receive all messages in the same order.

The strong semantics of *SendToGroup* make the implementation of the group directory service simple.

The service stores the administrative data on a raw disk partition of n fixed-length blocks. Block 0 contains information needed during recovery (see below). Blocks 1 to $n - 1$ contain a table of capabilities, indexed by object number. The capability in the object table points to a Bullet file that stores the directory, random number for access protection, and the sequence number of the last change. We assume that a block of the object table can be updated atomically. Bullet files are never modified (they are immutable).

Default operation

Each server in the directory service consists of several threads: multiple server threads and one group thread. The server threads are waiting for requests from a clients. The group thread is waiting for an internal message sent to the group. There can be multiple server threads, but there is only one group thread. A server thread that receives a request and initiates a directory operation is called the *initiator*.

The initiator first checks if the current group has a majority (i.e., at least two of the three servers must be up). If not, the request is refused; otherwise the request is processed. The reason why even a read request requires a majority is because the network might become partitioned. Consider the following situation. Two servers and a client are on one side of the network partition and the client deletes the directory *foo*. This update will be performed, because the two servers have a majority. Now assume that the two servers crash and that the network partition is repaired. If the client asks the remaining server to list the directory *foo*, it would get the contents of a directory that it had successfully deleted earlier. Therefore, read requests are refused if the group of servers does not have a majority. (There is an escape for system administrators in case two servers lose their data forever due to, for example, a head crash.)

Read operations can be handled by any server without the need for communication between the servers. When a read request is received, the initiator checks if the kernel has any messages buffered using *GetInfoGroup*. If so, it blocks to give the group thread a chance to process the buffered messages; before performing a read operation, the initiator has to be sure that it has performed all preceding write operations. If a client, for example, deletes a directory and then tries to read it back, it has to receive an error, even if the client requests were processed at different directory servers. As messages are sent using $r = 2$, it is sufficient to see if there are any messages buffered on arrival of the read request. Once these buffered messages are processed, the initiator can perform the read request.

Write operations require communication among the servers. First, the initiator generates a new capability, because all the servers must use the same capability when creating a new directory. Otherwise, some servers may consider a directory capability valid, while others do not. The initiator broadcasts the request to the group using the primitive *SendToGroup* and blocks until the group thread received and executed the request. Once it is unblocked, it sends the result of the request back to the client.

The group thread is continuously waiting for a message sent to the group (i.e., it is blocked in *ReceiveFromGroup*). If *ReceiveFromGroup* returns, the group thread first checks if the call to *ReceiveFromGroup* returned successfully. If not, one of the servers must have crashed. In this case, it rebuilds the group by calling *ResetGroup*, updates its commit block, and calls *ReceiveFromGroup* again. If it does not succeed in building a group with a majority of the members of the original group, the server enters recovery mode.

If *ReceiveFromGroup* returns successfully, the server creates the new directory on its Bullet

server, updates its cache, updates its object table, and writes the changed entry in the object table to its disk. As soon as one server writes the new entry to disk, the operation is committed. If no server fails, each server will receive all requests and service all requests in the same order and therefore all the copies of the directories stay consistent. There might be a small delay, but eventually each server will receive all messages.

When the client's RPC returns successfully, the user knows that one new copy of the directory is stored on disk and that at least two other servers have received the request and stored the new directory on disk, too, or will do so shortly. If one server fails, the client can still access its directories.

Let us analyze the cost of a directory operation in terms of communication cost and disk operations. Read operations do not involve communication or disk operations (if the requested directory is in the cache). Write operations require one group message sent with $r = 2$, a Bullet operation to store the new directory, and one disk operation to store the changed entry in the object table.

Recovery protocol

Block 0, the commit block, contains information that is needed during recovery and is shown in Figure 4. It contains the *configuration vector*. The configuration vector is a bit vector, indexed by server number. If server 2, for example, is down, bit 2 in the vector is set to 0.

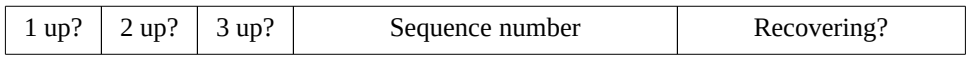

Figure 4: *Layout of the commit block.*

During recovery, the sequence number is computed by taking the maximum of all the sequence numbers stored with the directory files and the sequence number stored in the commit block. At first sight it may seem strange that a sequence number is also stored in the commit block, but this is needed for the following case. When a directory is deleted, the Bullet file containing the sequence number is deleted, but the server must store somewhere that it performed an update. The sequence number in the commit block is used for this case. It is only updated when a directory is deleted.

The *recovering* field is needed to keep track if a server crashed during recovery. If this field is set, the server knows that it crashed during recovery. In this case, it sets the sequence number to zero, because its state is inconsistent. It may have recent versions of some directories and old versions of other directories. The sequence number is set to zero to ensure that other servers will not try to update their directories from a server whose state is inconsistent.

A server starts executing the recovery protocol when it is a member of a group that forms a minority or when it comes up after having been down. Two conditions have to be met to recover:

1. The new group must have a majority to avoid inconsistencies during network partitions;
2. The new group must contain the set of servers that possibly performed the latest update.

It is the latter requirement that makes recovery of the group service complicated. During

recovery the servers need an algorithm to determine which servers failed last.

Such an algorithm exists; it is due to Skeen [32] and it works as follows. Each server keeps a *mourned set* of servers that crashed before it. When a server starts recovering, it sets the new group to only itself. Then, it exchanges with all other alive servers its mourned set. Each time it receives a new mourned set, it adds the servers in the received *mourned set* to its own *mourned set*. Furthermore, it puts the server with whom it exchanged the mourned set in the new group. The algorithm terminates when all servers minus the *mourned* set are a subset of the new group.

The complete recovery protocol is as follows. When a server enters recovery mode, it first tries to join the group. If this fails, it assumes that the group is not created yet and it creates the group. If after a certain waiting period, an insufficient number of members joined the group, it leaves the group and starts all over again. It may have happened that two servers recreated the group (e.g., two servers on each side of the network partition) and that they both cannot acquire a majority of the members. To avoid simultaneous group rebuilds happening forever, each server has a different waiting periode before it will retry. Furthermore, in case one of the members cannot start up at all (e.g., because of a disk head crash) an escape mechanism is available, allowing the remaining members to form a new group.

Once a server has created or joined a group that contains a majority of all directory servers, it executes Skeen's algorithm to determine the set of servers that crashed last, the *last set*. If this set is not a subset of the new group, the server has to wait for servers from the *last set* to join the group. If the *last set* is a subset of the new group, the new group has the most recent version of the directories. The server determines who in the group has them and gets them. Once it is up-to-date, it writes the new configuration to disk and enters normal operation.

This recovery protocol can still be improved. Skeen's algorithm assumes that network partitions do not occur. To make his algorithm work for our assumption, we forced the servers that have a minority to fail. Now the recovery protocol will fail in certain cases in which it is actually possible to recover. Consider the following sequence of events. Server 1, 2, and 3 are up; server 3 crashes; server 1 and 2 form a new group; server 2 crashes. Now as we want to tolerate network partitions correctly, we forced server 1 to fail. However, this is too strict. If server 1 stays alive and server 3 is restarted, server 1 and 3 can form a new group, because server 1 must have performed all the updates that server 2 could have performed. The rule in general is that two servers can recover, if the server that did not fail has a higher *sequence number*, as in this case it is certain that the new member has not formed a group with the (now) unavailable member in the meantime. We will incorporate this improvement in our directory service in the near future.

6. Performance of the Directory Service

The directory service has been used in an experimental environment for several months. It runs on machines comparable to a Sun3/60 connected by 10 Mbit/s Ethernet. The Bullet servers run on Sun3/60s and are equipped with Wren IV SCSI disks. In the experiments we were using a group of three replicated members (with $r = 2$).

We have measured the performance of three kinds of operations. The results are shown in Table 4. The first experiment measures the time to append a new (name, capability) pair to a directory and delete it subsequently (e.g., appending and deleting a name for a temporary file). The second experiment measures the time to create a 4-byte file, register its capability with the directory service, look up the name, read the file back from the file service, and delete the name from the directory service. This corresponds with the use of a temporary file that is the output of the first phase of a compiler and then is used as an input file for the second phase. Thus, the first experiment measures only the directory service, while the second experiment measures both the directory and file service. The third experiment measures the performance of the directory server for

read operations.

Operation	Group (*r = 2*)	Sun NFS	Group +NVRAM
Append-delete	184	87	27
Tmp file	215	111	52
Directory lookup	5	6	5

Table 4: *Performance of 3 kinds of directory operations (times in msec).*

For comparison reasons, we ran the same experiments using Sun NFS; the results are listed in the second column. The measurements were run on SunOS4.1.1 and the file used was located in */usr/tmp/*. NFS does not provide any fault tolerance or consistency (e.g., if another client has cached the directory, this copy will not be updated consistently when the original is changed). Compared to NFS, providing high reliability and availability costs a factor of 2.1 in performance for the ‘‘append-delete’’ test and 1.9 in performance for the ‘‘tmp file’’ test.

The dominant cost in providing a fault-tolerant directory service is the cost for doing the disk operations. Therefore, we have implemented a third version of the directory service, which does not perform any disk operations in the critical path. Instead of directly storing modified directories on disk, this implementation stores the modifications to a directory in a 24Kbyte Non Volatile RAM (NVRAM). Note that this relatively small amount of NVRAM available makes it more suited for use in a directory service than in a file service. An update to a directory generally requires less than 100 bytes of data to be modified. When the server is idle, it applies the modifications logged in NVRAM to the directories stored on disk. Because NVRAM is a reliable medium, this implementation provides the same degree of fault tolerance as the other implementations, while the performance is much better. A similar optimization has been used in [10, 24, 14].

Using NVRAM, some sequences of directory operations do not require any disk operations at all. Consider the use of */tmp*. A file written in */tmp* is often deleted shortly after it is used. If the append operation is still logged in NVRAM when the delete is performed, then both the append and the delete modifications to */tmp* can be removed from NVRAM without executing any disk operations at all.

We have implemented and measured a version of the directory service that uses NVRAM. Using group communication and NVRAM, the performance improvements for the experiments are enormous (see third column in Table 4). This implementation is 6.8 and 4.1 times more efficient than the pure group implementation. The implementation based on NVRAM is even faster than Sun NFS, which provides less fault tolerance and has a lower availability. We would need to compare it to Sun NFS with NVRAM support to obtain a better comparison, however.

7. Conclusion

Six design issues are important in group communication: addressing, reliability, ordering, delivery semantics, response semantics, and group structure. We have described the choices that have been made for Amoeba. Amoeba groups are addressed by a port and provide reliable totally-ordered communication. Furthermore, users can trade performance for fault tolerance.

To implement group communication, Amoeba uses a centralized negative acknowledgement protocol. The total ordering is enforced by a centralized machine, called the sequencer. Instead

of acknowledging every messages, members of the group piggyback the sequence number for the latest received messages on messages sent to the sequencer. The result is two simple and efficient protocols: the PB and BB protocols. If no failures occur, both protocols need on average only slightly more than two messages per reliable totally-ordered group message.

To illustrate the usage of group communication, we discussed the design and implementation of Amoeba's directory service. To achieve high availability and high reliability, the directory service replicates directories on three machines, each with its own disk. The replicas of a directory are kept consistent using group communication. We described two implementations of the directory service: one using NVRAM and one without NVRAM. NVRAM is used to avoid disk operations in the critical path.

Acknowledgements

Henri Bal, Leendert van Doorn, Greg Sharp, and Mark Wood provided comments on drafts of this paper, which improved its content and presentation substantially. We also want to thank the referees for their useful suggestions.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki, ''Transis: A Communication Sub-System for High Availability,'' *Proc. 22nd International Symposium on Fault-Tolerant Computing*, Boston, MA, pp. 76-84 (June 1992).
2. M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, ''Non-Volatile Memory for Fast, Reliable File Systems,'' *Proc. Fifth Int. Conf. on Architectural Support for Programming Language and Operating Systems*, Boston, MA, pp. 10-22 (Oct. 1992).
3. H.E. Bal, *Programming Distributed Systems*, Silicon Press, Summit, NJ (1990).
4. K.P. Birman and T.A. Joseph, ''Reliable Communication in the Presence of Failures,'' *ACM Trans. Comp. Syst.* **5**(1), pp. 47-76 (Feb. 1987).
5. K.P. Birman, A. Schiper, and P. Stephenson, ''Lightweight Causal and Atomic Group Multicast,'' *ACM Trans. Comp. Syst.* **9**(3), pp. 272-314 (Aug. 1991).
6. A.D. Birrell and B.J. Nelson, ''Implementing Remote Procedure Calls,'' *ACM Trans. Comp. Syst.* **2**(1), pp. 39-59 (Feb. 1984).
7. J.J. Bloch, D.S. Daniels, and A.Z. Spector, ''A Weighted Voting Algorithm for Replicated Directories,'' *Journal of the ACM* **34**(4), pp. 859-909 (Oct. 1987).
8. J. Chang and N.F. Maxemchuk, ''Reliable Broadcast Protocols,'' *ACM Trans. Comp. Syst.* **2**(3), pp. 251-273 (August 1984).
9. D.R. Cheriton and W. Zwaenepoel, ''Distributed Process Groups in the V kernel,'' *ACM Trans. Comp. Syst.* **3**(2), pp. 77-107 (May 1985).
10. D.S. Daniels, A.Z. Spector, and D.S. Thompson, ''Distributed Logging for Transaction Processing,'' *Proc. ACM SIGMOD 1987 Annual Conference*, San Francisco, CA, pp. 82-96 (May 1987).
11. A. El Abbadi, D. Skeen, and F. Cristian, ''An Efficient, Fault-Tolerant Algorithm for Replicated Data Management,'' *Proc. Fifth Symposium on Principles of Database Systems*, Portland, OR, pp. 215-229 (March 1985).
12. E.N. Elnozahy and W. Zwaenepoel, ''Replicated Distributed Processes in Manetho,'' *Proc. 22nd International Symposium on Fault-Tolerant Computing*, Boston, MA, pp. 18-27 (July 1992).

13. R. Gueth, J. Kriz, and S. Zueger, ‘‘Broadcasting Source-Addressed Messages,’’ *Proc. Fifth International Conference on Distributed Computing Systems*, Denver, CO, pp. 108-115 (1985).

14. S. Hariri, A. Choudhary, and B. Sarikaya, ‘‘Architectural Support for Designing Fault-Tolerant Open Distributed Systems,’’ *IEEE Computer* **25**(6), pp. 50-61 (June 1992).

15. A. Hisgen, A.D. Birrell, C. Jerian, T. Mann, M. Schroeder, and C. Swart, ‘‘Granularity and Semantic Level of Replication in the Echo Distributed File System,’’ *IEEE TCOS Newsletter* **4**(3), pp. 30-32 (1990).

16. L. Hughes, ‘‘A Multicast Interface for UNIX 4.3,’’ *Software Practice and Experience* **18**(1), pp. 15-27 (Jan. 1988).

17. L. Hughes, ‘‘Multicast Response Handling Taxonomy,’’ *Computer Communications* **12**(1), pp. 39-46 (Feb. 1989).

18. M.F. Kaashoek, ‘‘Group Communication in Distributed Computer Systems,’’ Ph.D. thesis, Vrije Universiteit, Amsterdam (Dec. 1992).

19. M.F. Kaashoek, R. Michiels, H.E. Bal, and A.S. Tanenbaum, ‘‘Transparent Fault-tolerance in Parallel Orca Programs,’’ *Proc. Symposium on Experiences with Distributed and Multiprocessor Systems III*, Newport Beach, CA, pp. 297-312 (March 1992).

20. M.F. Kaashoek, A.S. Tanenbaum, and K. Verstoep, ‘‘An Experimental Comparison of Remote Procedure Call and Group Communication,’’ *Proc. Fifth ACM SIGOPS European Workshop* , Le Mont Saint-Michel, France (Sept. 1992).

21. M.F. Kaashoek, R. van Renesse, H. van Staveren, and A.S. Tanenbaum, ‘‘FLIP: an Internetwork Protocol for Supporting Distributed Systems,’’ *ACM Trans. Comp. Syst.* (Feb. 1993).

22. H.T. Kung, ‘‘Gigabit Local Area Networks: a Systems Perspective,’’ *IEEE Communications Magazine* **30**(4), pp. 79-89 (April 1992).

23. L. Liang, S.T. Chanson, and G.W. Neufeld, ‘‘Process Groups and Group Communication: Classification and Requirements,’’ *IEEE Computer* **23**(2) (Feb. 1990).

24. B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, ‘‘Replication in the Harp File System,’’ *Proc. Thirteenth Symposium on Operating System Principles*, Pacific Grove, CA, pp. 226-238 (Oct. 1991).

25. K. Marzullo and F. Schmuck, ‘‘Supplying High Availability with a Standard Network File System,’’ *Proc. Eighth International Conference on Distributed Computing Systems*, San Jose, CA, pp. 447-453 (June 1988).

26. S. Mishra, L.L. Peterson, and R.D. Schlichting, ‘‘Implementing Fault-Tolerant Replicated Objects Using Psync,’’ *Proc. Eighth Symposium on Reliable Distributed Systems*, Seattle, WA (Oct. 1989).

27. S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren, ‘‘Amoeba: A Distributed Operating System for the 1990s,’’ *IEEE Computer* **23**(5), pp. 44-53 (May 1990).

28. L.L. Peterson, N.C. Buchholtz, and R.D. Schlichting, ‘‘Preserving and Using Context Information in IPC,’’ *ACM Trans. Comp. Syst.* **7**(3), pp. 217-246 (Aug. 1989).

29. R. van Renesse, A. S. Tanenbaum, and A. Wilschut, ‘‘The Design of a High-Performance File Server,’’ *Proc. Ninth International Conference on Distributed Computing Systems*, Newport Beach, CA, pp. 22-27 (June 1989).

30. R. van Renesse, ‘‘The Functional Processing Model,’’ Ph.D. Thesis, Vrije Universiteit,

Amsterdam (1989).

31. M. Satyanarayanan, ‘‘Scalable, Secure, and Highly Available Distributed File Access,’’ *IEEE Computer* **23**(5), pp. 9-22 (May 1990).
32. D. Skeen, ‘‘Determining the Last Process to Fail,’’ *ACM Trans. Comp. Syst.* **3**(1), pp. 15-30 (Feb. 1985).
33. A.S. Tanenbaum, *Computer Networks 2nd ed.*, Prentice-Hall, Englewood Cliffs, NJ (1989).
34. A.S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S.J Mullender, A. Jansen, and G. van Rossum, ‘‘Experiences with the Amoeba Distributed Operating System,’’ *Commun. ACM* **33**(12), pp. 46-63 (Dec. 1990).
35. A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal, ‘‘Parallel Programming Using Shared Objects and Broadcasting,’’ *IEEE Computer* **25** (Aug. 1992).