Using Group Communication to Implement a Fault-Tolerant Directory Service

M. Frans Kaashoek† *Andrew S. Tanenbaum* *Kees Verstoep*

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

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

Abstract

Group communication is an important paradigm for building distributed applications. This paper discusses a fault-tolerant distributed directory service based on group communication, and compares it with the previous design and implementation based on remote procedure call. The group directory service uses an active replication scheme and, when triplicated, can handle 627 lookup operations per second and 88 update operations per second (using nonvolatile RAM). This performance is better than the performance for the RPC implementation and it is even better than the performance for directory operations under SunOS, which does not provide any fault tolerance at all. The paper concludes that the implementation using group communication is simpler and has better performance than the one based on remote procedure call, supporting the claim that a distributed operating system should provide both remote procedure call and group communication.

Keywords: distributed (operating) systems, group communication, multicast, distributed applications, fault-tolerance, directory service, Amoeba.

1. Introduction

Many distributed operating systems support only point-to-point communication [1, 2, 3]. Amoeba [4, 5], on the other hand, is a distributed operating system that provides both point-to-point communication and 1-to-*n* communication (group communication). This paper discusses the design and implementation of a fault-tolerant distributed directory service using reliable and totally-ordered group communication and compares it to the previous implementation based on remote procedure call (RPC) [6]. Although the group directory service is better (it also tolerates network partitions), we conclude that its design and implementation are simpler and that it has better performance. The directory service is an example application that tolerates faults by using active replication. Our results extend, in principle, to other services that are based on active replication. Based on our experience with the directory service and run-time systems for distributed parallel programming [7], we claim that a distributed operating system should provide support for both remote procedure call and totally-ordered group communication.

The Amoeba directory service has a long history. The version in use for the last two years is based on RPC. It consists of two servers, each running on a separate machine [8]. Directory operations that do not modify a directory can be executed by either of the two servers, without any communication with the other server. Directory operations that modify a directory result in communication between the two servers. When an update request comes in at one server, it performs an RPC with the other server informing it of the intended update. If the other server is not busy performing a conflicting operation, it stores the intentions on disk and sends an OK message back. On receiving the reply, the original server performs the update and sends a reply back to the client. To avoid the costs of immediately creating two copies on separate disks of a directory that is updated, the directory service uses *lazy replication*. The server that processes the client request creates a copy on its local disk, but the second copy at the other server is created later in the background. As the directory service is only duplicated, it can not guarantee consistency in the presence of network partitions.

This paper discusses an alternative design and implementation of the directory service using Amoeba's primitives for group communication (see Fig. 1). These primitives guarantee that all processes constituting a group see each event in the same total order. Random mixtures, where some members see first a message from *A* and then a message from *B*, while other members see them in the reverse order are guaranteed not to happen. A programmer can request by specifying a *resilience degree*, *r*, that even in the face of *r* processor failures each surviving member will see all messages in the same order. For example, if a programmer specifies $r = 2$ for a group with 4 members, the system will guarantee that even in the face of 2 processor failures the remaining members will receive each message in a total order. By setting *r*, the programmer can trade performance against fault tolerance. The protocols that Amoeba uses to implement the group communication primitives have been described in an earlier publication [9]. The design of the directory service depends on the total ordering of events, but this

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

does not restrict our work to Amoeba alone. Other systems, such as Isis [10], Psync [11], extensions to the V [12] system [13], and Delta-4 [14] provide primitives with similar semantics.

Primitive	Description
CreateGroup	Create a new group
JoinGroup	Make a process member of a group
LeaveGroup	Leave a group
SendToGroup	Send message to all members of a group
ReceiveFromGroup	Receive the next message in sequence
ResetGroup	Rebuild group after a failure
GetInfoGroup	Get group information from kernel

Fig. 1 Amoeba's system calls for group communication.

The outline of the rest of the paper is as follows. In Section 2 we will describe the functionality required from the directory service and its role in the Amoeba system. In Section 3 we will discuss in detail the design and implementation using group communication. In Section 4 we will compare the directory service using group communication with the directory service using RPC and give performance measurements for both. In the same section, we will also compare the performance of directory operations under Amoeba with similar operations under SunOS and compare the performance of the group directory service with the same service using nonvolatile RAM (NVRAM). In Section 5 we will discuss the results of the comparisons and describe related work. In Section 6 we will draw our conclusions.

2. A Fault-Tolerant Directory Service

The directory service is a vital service in the Amoeba distributed operating system [8]. It provides, among other things, a mapping from ASCII names to capabilities. A *capability* in Amoeba identifies and protects an object (e.g., a file). A capability is 128-bit string consisting of 4 parts: 1) a *port* that identifies the service; 2) an *object number* that identifies an object at the service specified by the port; 3) a *rights field* that specifies which operations the holder of the capability may perform; 4) a *check field* that determines if the capability is valid or not. The set of capabilities a user possesses determines which objects he can access. The directory service allows the users to store these capabilities under ASCII names to make life easier for them.

A directory in Amoeba is a table with several columns, one for each protection domain. For example, the first column might store capabilities for the owner (with all the rights bits on), the second might store capabilities for members of the owner's group (with some of the rights bits turned off), and the third might store capabilities for everyone else (with only the read bit turned on). When the owner of a directory gives away a capability for it, the capability is really a capability for a single column, not for the directory as a whole. When giving a directory capability to an unrelated person, the owner could give a capability for the third column. The recipient of this capability would have no access to the more powerful capabilities in the first two columns.

The directory service supports the operations shown in Figure 2. There are operations to manipulate directories, to manipulate a single row (i.e., a tuple consisting of a string and a capability) of a directory, and to manipulate a set of rows. One of the most important things to know about the directory service is the frequency of the read operations (e.g., list directory) and write operations (e.g., delete directory), because these numbers influence the design. Measurements over three weeks showed that 98% of all directory operations are reads. Therefore, both the RPC directory service and the group directory service optimize read operations.

Operation	Description
Create dir	Create a new directory
Delete dir	Delete a directory
List dir	List a directory
Append row	Add a new row to a directory
Chmod row	Change protection
Delete row	Delete row of a directory
Lookup set	Lookup capabilities in a set of rows
Replace set	Replace capabilities in a set of rows

Fig. 2 Operations supported by the directory service.

In this paper, we focus on the design and implementation of the directory service and not on the reasons why this interface was chosen. This has been discussed by Van Renesse [8]. For other papers discussing the design of a naming service we refer the reader to [15, 16, 17, 18].

The directory service must be highly reliable and highly available. Users rely on the directory service to store capabilities without losing them and users must always be able to access their capabilities. To fulfill these demands the directory service replicates (name, capability) pairs on multiple machines, each with its own disk. If one of the machines is unavailable, one of the other machines will be able to reply to a user's request. If one of the disks becomes unreadable, one of the other disks can be used to reply to a user's request. The key problem is to keep the replicas of a name-capability pair consistent in an efficient way. An update to a directory must be performed quickly, because otherwise many applications will run less efficiently.

We require that the directory service maintains *one-copy serializability* [19]. The execution of operations on the directory service must be equivalent to a serial execution of the operations on a nonreplicated directory service. To achieve this goal, each operation of the directory service is executed indivisibly. The directory service does not support indivisible execution of a set of operations, as this requires atomic transactions [20, 21]. It also does not support failure-free operations for clients, as this

requires updating a log file on each operation. We feel that these semantics are too expensive to support, and, moreover, are seldom necessary.

Both implementations of the directory service assume clean failures. A processor works or does not work (i.e., fail-stop failures), and it does not send malicious or contradictory messages (i.e., it does not exhibit Byzantine failures). The RPC implementation also assumes that network partitions will not happen; the group implementation, however, guarantees consistency even in the case of clean network partitions (e.g., any two processors in the same partition can communicate while any two processors in different partitions cannot communicate) [22]. Stronger failure semantics could have been implemented using techniques as described in [23, 14, 24, 25]. Again, we feel that these stronger semantics are too expensive to support, and, moreover, are overkill for an application like the directory service.

In the rest of this paper, we assume that the following basic requirements for a fault-tolerant directory service are met. Each directory server should be located on a separate electrical group (with its own fuse) and all the directory servers should be connected by multiple, redundant, networks. Because the Amoeba communication primitives are implemented on top of a network protocol (FLIP) [26], the latter requirement can be fulfilled. Although it could run on multiple networks without a single software modification, the current implementation runs on a single network.

3. Using Group Communication

Unlike the RPC directory service, the group implementation 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 the read-one write-all policy, called *accessible copies* [27]. Recovery is based on the protocol described by Skeen [28]. In this section, we will describe in detail the algorithms used in the implementation of the group directory service.

The organization of the group directory service is depicted in Figure 3. The directory service is currently built out of three directory servers, three Bullet file servers [29] and three disk servers. Each directory server only uses one Bullet server and one disk server, which share the same disk. Each directory server stores one copy of each directory in a separate Bullet file.

The directory servers initially form a group with a resilience degree, *r*, of 2. This means that if *SendToGroup* returns successfully, it is guaranteed that all three have received the message and thus that, even if two processors fail, the message will still be processed by the third one. Furthermore, it is guaranteed that even in the presence of communication and processor failures, each server will receive all messages in the same order. The strong semantics of *SendToGroup* make the implementation of the group directory service simple.

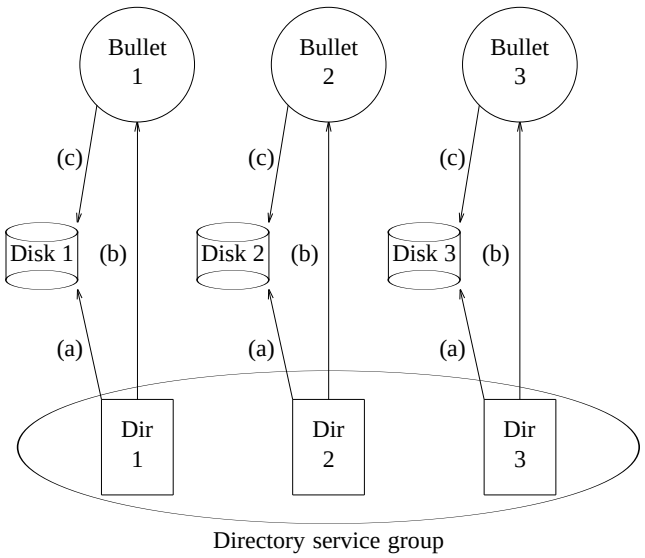

Fig. 3 Organization of the directory service based on group communication. (a) Administrative data; (b) Directories; (c) Files.

The administrative data are stored on a raw disk partition of *n* fixed-length blocks. Block 0 contains global information about the directory service and blocks 1 to $n-1$ contain the capabilities of the Bullet files storing the contents of a directory, including the sequence number of the last change. Block 0, the commit block for the group directory service is shown in Figure 4. 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. It describes the last configuration with a majority of which the server was a member.

Configuration vector

1 up?	2 up?	3 up?	Sequence number	Recovering?

Fig. 4 Layout of the commit block.

Each time an update operation is performed, a sequence number stored with the directory is increased. 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 reference to the Bullet file containing the directory and the sequence number is deleted, but the server must record 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 whether a server crashed during recovery. If this field is set, the

server knows that this has happened. In this case, it sets the sequence number to zero, because its state may be 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.

3.1. Default Operation

The protocol to keep the directories consistent and to perform concurrency control is given in Figure 5. A server in the group directory service consists of several threads: the server threads and one group thread. The server threads are waiting for a request from a client. The group thread is waiting for an internal message sent to the group. At each server 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 that 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 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.)

As in the RPC implementation, 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 a resilience degree $r = 2$, it is sufficient to check 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, as it can be 100 percent sure that it has seen all preceeding update operations.

Write operations require communication among the servers. First, the initiator generates a new *check field*, because all the servers must use the same *check field* when creating a new directory. 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.

```
Initiator:
  if (!majority()) return failure;          /* majority required          */
  if (read_operation(request)) {            /* read request?              */
      GetInfoGroup(&group_state);           /* any buffered messages? */
      buffered_seqno = buffered(&group_state);
      wait until seqno = buffered_seqno;
  } else {                                  /* write request              */
      generate check-field;                 /* for new directory          */
      SendToGroup(request, check-field, me);
      wait until group thread has received and executed the request;
  }
  send reply to client;

Group thread:
  if (group failure) {                      /* did a server fail?         */
      rebuild majority of group;            /* call ResetGroup            */
      if (group rebuild failed) enter recovery;
      GetInfoGroup(&group_state);           /* get status of rebuilt      */
      write commit block;                   /* update config vector       */
      try again;                            /* start receiving again      */
  } else {
      create directory on Bullet file;      /* use supplied checkfield */
      update cache;
      update object table;
      write changed object table to disk;   /* commit                     */
      increase_and_wakeup(seqno);
      if (sender == me) wakeup initiator;
      remove old Bullet files;
  }
```

Fig. 5 Protocol to ensure consistency of the copies of a directory.

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*. If it does not succeed in building a group with a majority of the members of the original group, the remaining directory servers run the recovery protocol described in the next section.

If *ReceiveFromGroup* returns successfully, the server updates its cache, creates the new directories on its Bullet server, updates its object table, and writes the changed entry in the object table to its disk. As soon as one server has written 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 his directories.

Let us analyze the cost of a directory operation in terms of communication cost and disk operations. As in the RPC implementation, 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. Compared to the RPC implementation, the number of disk operations is smaller. The RPC implementation requires an additional disk operation to store an intentions list with updates. The number of messages in the group service, however, is higher. A *SendToGroup* with $r = 2$ requires 5 messages, whereas an RPC in Amoeba requires only 3 messages. The cost of sending a message, however, is an order of magnitude less than the cost of performing a disk operation. Thus, roughly, the performance of the group implementation is better than the performance of the RPC implementation, while providing more fault tolerance and a higher availability.

This analysis is not completely fair, however. If the RPC implementation, like the group implementation, had stored the sequence number with the directory files and thereby avoided one disk write for the commit block, the performance of the RPC implementation would have been better, as it would send fewer messages. On the other hand, if the RPC service had been triplicated, it would have been slower than the group service, because then it would have sent more messages (4 RPCs against one *SendToGroup*).

3.2. Recovery Protocol

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. The protocol for recovery of the group service is more complicated than the protocol for the RPC service, because more servers are involved. Consider the following sequence of events in a group of three servers that is up and running. Server 3 crashes. Servers 1 and 2 rebuild the group, so their *configuration vector*s have the value 110 (1 and 2 are up; 3 is down). Now, both 1 and 2 also crash. When server 1 comes up again, its vector reads 110, but on its own it cannot form a group. To execute a client update request, a majority of the servers must be up and form one group; otherwise, copies of a directory could become inconsistent, for example, in the case of a network partition.

If server 3 also comes up, its vector reads 111. At first sight, it may appear that 1 and 3 can form a group, as together they form a majority. However, this is not sufficient. Server 2, who is still down, may have performed the latest update. To see this, consider the following sequence of events just before 1 and 2 crashed. A client update request is received by 1, it successfully sends it to server 1 and 2. Now both 1 and 2 have the message buffered. It can happen that 1 crashes before processing the message, while 2 crashes after processing the message. In this case, server 2 has the latest version of the directories and thus 1 and 3 cannot form a new group and start accepting requests.

Now assume that server 2 comes up instead of server 3. The *configuration vector* of both servers 1 and 2 read 110. From this information they can conclude that 3 crashed before 1 and 2 did. Furthermore, no update can have been performed after 1 or 2 crashed, because there was no majority. Servers 1 and 2 together are therefore sure that one of them has the latest version of the directories. Thus, they can recover without server 3 and use the *sequence number* to determine who actually has the latest version.

In general, 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 are the servers that failed last.

Such an algorithm exists; it is due to Skeen [28], 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.

Figure 6 gives the complete recovery protocol. 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 have joined the group, the server leaves the group and starts all over again. It may have happened that two servers have created the group (e.g., one server on each side of a network partition) and that they both cannot acquire a majority of the members.

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 starts all over again, waiting 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.

The recovery protocol can be improved. Skeen's algorithm assumes that network partitions do not occur. To make his algorithm work under our assumption that network partitions can happen, we forced the servers that form a group with a minority of the number of servers to fail. Now the recovery protocol will fail in certain cases in which it is actually possible to recover. Consider the

```
Recovery:
  re-join server group or create it;
  while (minority && !timeout) {      /* wait for some time          */
       GetInfoGroup(&group_state);
  }
  if (minority) try again;            /* leave group and retry      */
  newgroup[me] = 1;                   /* initialize new group vector */
  SequenceNo[me] = SeqNr;             /* initialize seqno vector    */
  initialize mourned vector from configuration vector;
  for (all members in group) {
       exchange info with server s;   /* mourned set and seqno      */
       if (success) {                 /* RPC succeeded?             */
            newgroup[s] = 1;          /* add server to new group    */
            SequenceNo[s] = SeqNr;
            mourned set += received mourned set; /* take union      */
       }
  }
  last = all servers – mourned set;   /* who performed last update? */
  if (last is not subset of new group) try again;
  s = HighestSeq(SequenceNo);         /* who has has highest seqno  */
  get copies of latest version of directories from s;
  if (!success) try again;            /* succeeded in getting copies?*/
  write commit block;                 /* store configuration vector */
  enter normal operation;
```

Fig. 6 Recovery protocol for group directory service.

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 available 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.

4. Experminental Comparison

Both the RPC and group service are operational. The RPC service has been in daily use for over two years. The group directory service has been used in an experimental environment for several months and will shortly replace the RPC version. Both directory services run on the same hardware: machines comparable to a Sun3/60 connected by 10 Mbit/s Ethernet. When a message is sent using *SendToGroup*, the Amoeba kernel uses the Ethernet multicast capability to send the message in one packet to all members in the specified group. The Bullet servers run on Sun3/60s and are equipped with Wren IV SCSI disks.

4.1. Performance Experiments with Single Client

We have measured the failure-free performance of three kinds of operations on an almost quiet network. The results are shown in Figure 7. 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 to 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 lookup operations.

Operation (# copies)	**Group** (3)	**RPC** (2)	**Sun NFS** (1)	**Group+ NVRAM** (3)
Append-delete	184	192	87	27
Tmp file	215	277	111	52
Directory lookup	5	5	6	5

Fig. 7 Performance of three kinds of directory operations for three different Amoeba implementations and for one UNIX implementation. All times are in msec.

For the append-delete test and for the tmp file test, the implementation using group communication is slightly more efficient than the one using RPC. Thus, although the group directory service is triplicated and the RPC implementation is only duplicated, the group directory service is more efficient. The reason is that the RPC implementation uses one additional disk operation to store intentions and the new sequence number. For read operations, the performance of all implementations is the same. Read operations do not involve any disk operations, as all implementations cache recently used directories in RAM, and involve only one server.

For comparison reasons, we ran the same experiments using Sun NFS; the results are listed in the third column. The measurements were run on SunOS 4.1.1 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 24 Kbyte NonVolatile RAM (NVRAM). When the server is idle or the NVRAM is full, 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 [24, 30, 31, 32].

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 the fourth column in Fig. 7). This implementation is 6.8 and 4.3 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. If the RPC service had been implemented with NVRAM, one could expect similar performance improvements.

4.2. Experiments with Multiple Clients

To determine the performance of the directory services for multiple clients we ran three additional experiments. The first experiment measures the throughput for lookup operations; its results are depicted in Figure 8. The graph shows the total number of directory lookup operations that were processed by a directory service for a varying number of clients. A rough estimate of the maximum number of lookup operations that, in principle, can be processed per second can be easily computed. The time needed by a server to process a read operation is roughly equal to 3 msec (the time for a lookup operation minus the time to perform an RPC with the server). The maximum number of read operations per server is therefore 333 per second. Thus, the upper bound on read operations for the group service using 3 servers is 1000 per second and for the duplicated RPC implementation it is 666 per second.

Neither service achieves the upper bounds, because the client requests are not evenly distributed among the servers. The first time a client performs a RPC with some service, its kernel locates the service by sending a broadcast message containing the port *p* for the requested service. Every server that listens to the port *p* answers with an RPC HEREIS message. The client's kernel stores the network address for each server that answers in its port cache and sends the request to the first server that replied. If at some point one of the servers is busy and is not listening to *p* when a request comes in, its kernel sends a NOTHERE back to the client's kernel. The client's kernel removes the server's network address from its port cache and selects another server from its port cache or locates the service again if its port cache does not contain an alternative.

This heuristic for choosing a server is not optimal. Some clients may pick the same server, while another server is idle. From the graph one can see this happen. This possibility was also reflected in our measurements. The numbers depicted are averages over a large number of runs, but the standard deviation is high. In some runs,

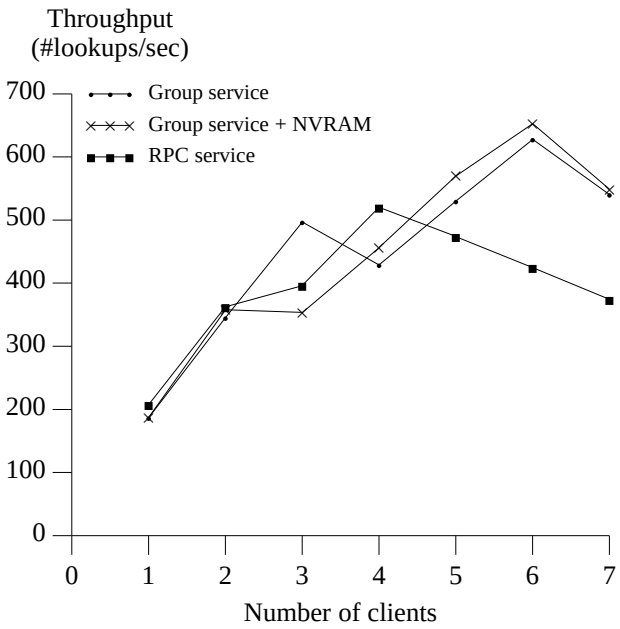

Fig. 8 Throughput in total number of lookups per second for increasing numbers of clients.

the standard deviation was almost 100 operations per second. The heuristic, however, is good enough that with higher load the clients requests are evenly distributed among the servers. One can also conclude from the graph that the RPC directory service can support fewer clients than the group service. The RPC directory service gets overloaded with 520 requests per second, whereas the group service gets overloaded with 652 requests per second.

Figure 9 shows the throughput for the append-delete test. This experiment measures the maximum number of pairs of append-delete operations that the service can support per second. Again, an upper bound can easily be estimated for each service. Processing a pair of append-delete operations takes roughly 22 msec in the group NVRAM service, 179 msec for the group service, and 187 for the RPC service. As write operations cannot be performed in parallel, the upper bounds per service are 45, 5, and 5. All three implementations reach the upper bound.

5. Discussion and Comparison

Making a fair comparison between the group directory service and the RPC directory service is hardly possible, as both services assume different failure modes. The RPC service is duplicated and does not provide consistency in the face of network partitions, whereas the group service is triplicated and does provide consistency in the face of network partitions. Furthermore, the RPC implementation employs lazy replication, whereas the group implementation employs active replication, resulting in a higher degree of reliability and availability for

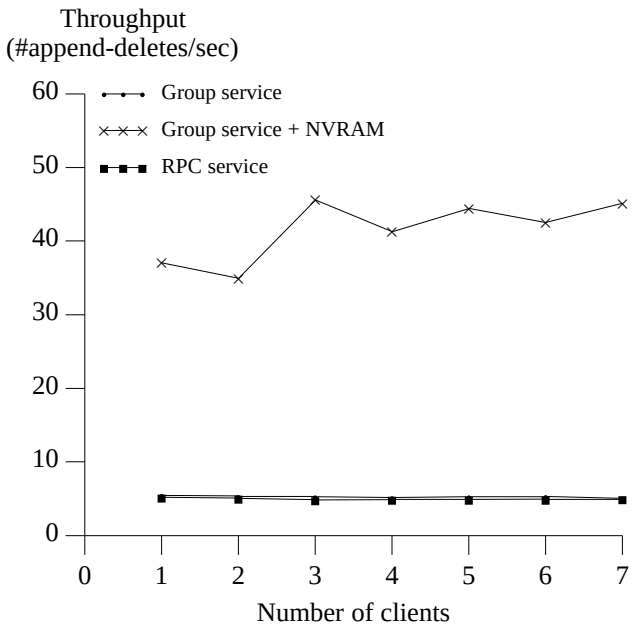

Fig. 9 Throughput in total number of append-delete operations per second for increasing numbers of clients. As append and delete operations are both write operations, the actual write throughput is twice as high.

the group directory service. After the RPC directory service has performed an update on a directory, the new directory is directly stored on only one Bullet file. If the Bullet server storing this file crashes before the second replica is generated, the directory will become unavailable. In the group directory service this cannot happen, because the service creates all replicas at about the same time before the client is told that the update has succeeded.

Although the group service provides a higher reliability and availability, its protocols for normal operation (no failures) are as simple as the RPC protocols. The group recovery protocols are more complex, but this is due to the fact that the group service is built out of three servers instead of out of two. A three server implementation of the RPC service would require a similar protocol for recovery as used in the group service.

The performance of the group directory service is better than the RPC directory service. This is, however, mainly due to the fact that the group implementation avoids one disk write. The RPC directory service could have been implemented in such a way that it also avoids this additional disk write. Such an implementation is likely to have the same performance as the group implementation. On the other hand, if the RPC directory service had been triplicated, it would have had to implement two-phase locking [33], resulting in higher communication overhead compared with the group implementation.

Summarizing, the group directory service is hard to compare with the RPC directory service due to differences in the failure semantics and differences in the implementations that are not related to using group communication or using RPC. However, we believe that the comparison gives enough insight in all the design choices and implementation issues that an RPC directory service with the same specification as the group directory service will be more complicated and have a worse performance than a group implementation.

There is an extensive literature on designing and building fault-tolerant systems, covering both practical and theoretical research. It is outside the scope of this paper, however, to review all this work. Instead we focus on a number of actual implementations of fault-tolerant file and directory services in systems similar to Amoeba.

Marzullo and Schmuck describe a fault-tolerant implementation of Sun's Network File System (NFS) using the Isis toolkit [34]. As the authors did not want to change the client side nor the server side of NFS, they introduced an extra level of indirection. Client processes do not talk directly with the file service, but go through an intermediate process, called an *agent*. The agents hide from the clients that the file service is replicated and use internally one of Isis's broadcast primitives to keep their state consistent. The agents update the replicas of a file using regular Sun RPC, because to employ broadcast would have meant changing the file servers to use Isis.

Harp is another approach to increase the fault tolerance of NFS [31]. Unlike Marzullo and Schmuck, the authors of Harp decided to change the file server to avoid an extra level of indirection. Harp is based on a primary copy protocol [35]. Clients communicate with one designated server, called the *primary*, to perform operations. The other servers are termed *secondaries*. On a write operation, the primary first sends the results to secondaries before sending a reply to the client. All servers store the result in NVRAM and copy the result lazily to disk to improve performance. If the primary crashes, the secondaries elect a new primary.

Another fault-tolerant file system is Coda [36]. Coda replicates files at the server side and also caches files at the client side. The clients cache whole files, so even if all servers fail, the clients are able to continue working with the cached files. If client and servers are connected, callbacks are used to keep the caches of the clients and servers consistent. The servers themselves use active replication and an optimistic variant of the read-one write-all policy to keep replicas consistent. The implementation is based on a parallel RPC mechanism that exploits the multicast capability of a network [37].

Another approach to a fault-tolerant distributed file system is Echo [38]. Like Harp, Echo uses a primary copy scheme. Unlike Harp, it does not perform replication at the file level, but at the level of an array of disk blocks. One of the reasons for doing so is that Echo uses *multiported* disks, which can be accessed by multiple servers. The multiported disks and replication at the level of disk blocks allow a primary to continue working even if all secondaries have failed. The primary can directly

write to all disks without having to go through the secondaries.

A fault-tolerant directory service is described by Mishra, Peterson, and Schlichting [39]. This directory service also uses active replication, but it is based on the assumption that operations are idempotent. It uses Psync's protocols to enforce an ordering on the messages sent by the servers [11]. To enhance concurrency of the directory service operations, the directory service uses the partial ordering and the property that some operations are commutative (e.g., list directory and lookup entry). To be able to recover, the servers checkpoint their state to non-volatile storage. Using the checkpoint and the partial order among messages, the service can reconstruct the state before a failure.

Daniels and Spector describe an algorithm designed for replicated directories [40]. Their algorithm is based on Gifford's weighted voting [41]. The algorithm exploits the observation that many operations on a single directory entry can be performed in parallel if the operations access different entries. Simulations done by the authors show that the additional cost for their algorithm is low, while it provides better performance.

Baker et. al. have simulated and analysed the performance impact of NVRAM in two configurations: file caches in client workstations and write buffers in file servers [32]. The latter approach is comparable to our use of NVRAM in the directory service. Their measurements indicate that the addition of one-half megabyte of NVRAM can decrease the number of disk acceses by 20 to 90% (in extreme cases). The results suggest that even at today's prices of NVRAM (four to six times more expensive than DRAM) the use of NVRAM in file servers can be cost-effective. A reimplementation of Amoeba's Bullet file service using group communication as well as NVRAM is certainly feasibly.

6. Conclusion

We have tried to support the claim that a distributed system should not only support RPC, but group communication as well. Group communication allows simpler and more efficient implementations of a large class of distributed applications. As an example to demonstrate the claim we looked in detail at the design and implementation of a fault-tolerant directory service. Although a completely fair comparison is not possible, due to differences in failure semantics and differences in the implementations unrelated to using group communication and RPC, we nevertheless claim that the directory service using group communication is not only easier to implement, but also more efficient.

Another important conclusion of our study is that disk operations are the major performance bottleneck in providing fault tolerance. By using a relatively new but already wide-spread technology, NVRAM, the performance of the directory service for update operations improves by an order of magnitude. The group directory service allows for 627 lookup operations per second and 88 update operations per second.

Acknowledgements

Henri Bal, Leendert van Doorn, Elmootazbellah Nabil Elnozahy, Robbert van Renesse, Mark Wood, and Willy Zwaenepoel provided comments on drafts of this paper.

References

1. R.F. Rashid, ''Threads of a New System,'' *Unix Review* **4**(3), p. 37 (August 1986).
2. J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch, ''The Sprite Network Operating System,'' *IEEE Computer* **21**(2), pp. 23-36 (Feb. 1988).
3. R. Pike, D. Presto, K. Thompson, and H. Trickey, ''Plan 9 from Bell Labs,'' *EKUUG Summer 90*, London, pp. 1-9 (July 1990).
4. 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).
5. 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).
6. A.D. Birrell and B.J. Nelson, ''Implementing Remote Procedure Calls,'' *ACM Trans. Comp. Syst.* **2**(1), pp. 39-59 (Feb. 1984).
7. A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal, ''Parallel Programming using Shared Objects and Broadcasting,'' *IEEE Computer* **25**(8), pp. 10-19 (Aug. 1992).
8. R. van Renesse, ''The Functional Processing Model,'' Ph.D. Thesis, Vrije Universiteit, Amsterdam (1989).
9. M.F. Kaashoek and A.S. Tanenbaum, ''Group Communication in the Amoeba Distributed Operating System,'' *11th Int'l Conf. on Distributed Computing Systems*, Arlington, Texas, pp. 222-230 (20-24 May 1991).
10. K.P. Birman and T.A. Joseph, ''Exploiting Virtual Synchrony in Distributed Systems,'' *Proc. Eleventh Symposium on Operating Systems Principles*, Austin, TX, pp. 123-138 (Nov. 87).
11. 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).
12. D.R. Cheriton and W. Zwaenepoel, ''Distributed Process Groups in the V kernel,'' *ACM Trans. Comp. Syst.* **3**(2), pp. 77-107 (May 1985).
13. S. Navaratnam, S. Chanson, and G. Neufeld, ''Reliable Group Communication in Distributed Systems,'' *Proc. Eighth International Conference on Distributed Computing Systems*, San Jose, CA, pp. 439-446 (June 1988).
14. P.A. Barrett, A.M. Hilborne, P. Verissimo, L. Rodrigues, P.G. Bond, D.T. Seaton, and N.A. Speirs, ''The Delta-4 Extra Performance Architecture (XPA),'' *Proc. 20th International Symposium on Fault-Tolerant Computing*, Newcastle, UK, pp. 481-488 (June 1990).
15. R. Curtis and L. Wittie, ''Global Naming in Distributed Systems,'' *IEEE Software* **1**(4), pp. 76-80 (July 1984).
16. M.D. Schroeder, A.D. Birrell, and R.M. Needham, ''Experience with Grapevine: The Growth of a Distributed

System,'' *ACM Trans. Comp. Syst.* **2**(1), pp. 3-23 (Feb. 1984).

17. B.W. Lampson, ''Designing a Global Name Service,'' *Proc. Fifth Annual Symposium on Principles of Distributed Computing*, Calgary, Canada, pp. 1-10 (Aug. 1986).
18. D.R. Cheriton and T.P. Mann, ''Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance,'' *ACM Trans. Comp. Syst.* **7**(2), pp. 147-183 (May 1989).
19. P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA (1987).
20. J.N. Gray, ''Notes on Database Operating Systems,'' pp. 393-481 in *Operating Systems: an Advanced Course, Lecture Notes in Computer Science*, Springer Verlag, New York (1978).
21. B.W. Lampson, ''Atomic Transactions,'' pp. 246-265 in *Distributed Systems - Architecture and Implementation, Lecture and Notes in Computer Science*, Springer Verlag, Berlin (1981).
22. S.B. Davidson, H. Garcia-Molina, and D. Skeen, ''Consistency in Partitioned Networks,'' *ACM Computing Surveys* **17**(3), pp. 341-370 (Sept. 1985).
23. F. Cristian, ''Understanding Fault-Tolerant Distributed Systems,'' *Commun. ACM* **34**(2), pp. 56-78 (Feb. 1991).
24. 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).
25. S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, S. Tao, and A. Tully, ''Principal Features of the VOLTAN Family of Reliable Node Architectures for Distributed Systems,'' *IEEE Trans. on Computers* **41**(5), pp. 542-549 (May 1992).
26. 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).
27. 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).
28. D. Skeen, ''Determining the Last Process to Fail,'' *ACM Trans. Comp. Syst.* **3**(1), pp. 15-30 (Feb. 1985).
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. 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).
31. 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).
32. 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).
33. K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, ''The Notion of Consistency and Predicate Locks in a Database System,'' *Commun. ACM* **19**(11), pp. 624-633 (Nov. 1976).
34. 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).
35. P. Alsberg and J. Day, ''A Principle for Resilient Sharing of Distributed Resources,'' *Proc. Second International Conference on Software Engineering*, pp. 627-644 (Oct. 1976).
36. M. Satyanarayanan, ''Scalable, Secure, and Highly Available Distributed File Access,'' *IEEE Computer* **23**(5), pp. 9-22 (May 1990).
37. M. Satyanarayanan and E.H. Siegel, ''Parallel Communication in a Large Distributed Environment,'' *IEEE Trans. on Computers* **39**(3), pp. 328-348 (March 1990).
38. 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).
39. 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, pp. 42-52 (Oct. 1989).
40. 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).
41. D.K. Gifford, ''Weighted Voting for Replicated Data,'' *Proc. Seventh Symposium on Operating System Principles*, Pacific Grove, CA, pp. 150-159 (Dec. 1987).