A Directory Service supporting Availability and Consistency

Robbert van Renesse
Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam

ABSTRACT

Object-based distributed operating systems often employ a directory service to map ASCII names onto internal object identifiers. In this paper we suggest that by extending the functionality of the directory service with support for replication and concurrency control, the rest of the distributed system can be kept relatively simple. We present the design and implementation of such a directory service for the Amoeba distributed operating system, and compare these to other existing directory services.

1. Introduction

The Amoeba distributed operating system [Mullender and Tanenbaum 1986, 1984; Van Renesse et al. 1988] is a research project investigating distributed operating systems, languages, and applications, both local and wide-area. The project originated at the Vrije Universiteit and is now being carried out jointly by teams there and at the Centre for Mathematics and Computer Science, both in Amsterdam. The system is based on the object model, and consists of four types of components: specialized services (such as the file and the time service), dynamically allocatable processors which together constitute the *processor pool*, workstations that provide the user interface, and gateways to interconnect geographically distributed Amoeba sites.

Objects are managed by specialized services, accessed using Remote Procedure Calls, and named and protected by capabilities [Tanenbaum et al. 1986]. Amoeba capabilities are not handled by the operating system, because the operating system kernels, especially those of workstations, cannot be trusted. Instead they are managed by user processes and protected cryptographically as described below.

This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.) under grant 125-30-10.

An Amoeba capability is currently a 128-bit number, consisting of four fields (see Figure 1). The *Service Port* is the (location independent) address of the service managing the object named by the capability. The *Object Number* identifies the object within the service. The *Rights Field* is an 8-bit bit map describing the access rights of the holder of the capability to the object. The *Check Field* is a 48-bit number to protect the capability against forging and tampering. (In the near future capabilities will be enlarged to 256 bits.)

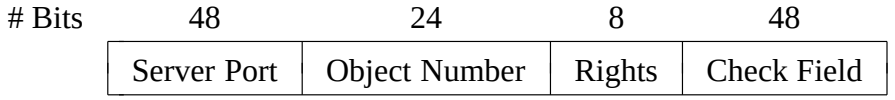

Fig. 1. An Amoeba capability.

To create an object a user sends a request to a specific service. The service allocates a free slot in its object table, and associates a 48-bit random number with this slot. A capability for the new object is created by putting the port of the service in the *Service Port*, the index into the object table in the *Object Number*, a bit map with all bits set to one in the *Rights Field*, and the random number in the *Check Field*. The capability is returned to the user. To use the object later, the user must supply the capability. By comparing the random number of the object identified by the *Object Number*, and the *Check Field* in the capability itself, the service can check whether access is allowed. The sparseness of the 128- (or soon 256-) bit capability makes it hard to guess (like a login password).

The user can give capabilities away to other users to allow sharing of objects. Usually the user wants to deny certain access rights to the other users. It is not sufficient to just clear the corresponding bits in the *Rights Field* because the other users can easily set them again. To keep users honest, the check field is changed using a one-way encryption function:

Check Field := E(random number, Rights Field)

Given the *Check Field* and the *Rights Field* (as present in the capability), it is computationally very hard to find the random number associated with the object [Evans et al. 1974]. The service, however, can easily check if the capability is authentic by recomputing the one-way function, *E*, using its copy of the random number of the object and the *Rights Field* in the capability, and comparing the result with the *Check Field* in the capability. Holders of restricted capabilities can generate further restricted capabilities by asking the service to do it for them.

The result of the one-way encryption function may be cached to make subsequent access checks fast. Note that authentication of users is never required: possession of a capability determines whether access is allowed or not. The large random

number makes forging of capabilities very hard, while the one-way encryption function prevents tampering with the access rights in capabilities.

To make the use of capabilities practical, they can be stored under ASCII names in *directories*. Directories are ordinary objects managed by the *directory service*. In its simplest form, a directory is a two-column table with the ASCII names in the first column, and the corresponding capabilities in the second. Like other objects, directories are named by capabilities. By storing directory capabilities in other directories, an arbitrary naming graph can be built. The directory service is perhaps the most important service in the system. If the service is unavailable, the users will not be able to retrieve their capabilities, and thus not be able to access objects.

This paper describes the design and an implementation of the Amoeba directory service. In section 2 we describe how the directory service can support the availability and mutual consistency of objects. Section 3 describes the design of the directory service, and section 4 the operations on directories. Section 5 describes its replicated implementation, and the resulting availability. Section 6 gives the conclusion.

2. Supporting Availability and Consistency

The directory service maps ASCII strings onto object capabilities. Users can store capabilities under meaningful names in directories. They can group objects by creating multiple directories, and share objects through public directories. Besides these traditional functions of directory services, the Amoeba directory service has a number of other functions. These functions deal with the mutual consistency and replication of objects of arbitrary type, and are the subject of this section.

In traditional object-based systems that support the mutual consistency of objects, each service needs to support atomic transactions [Lampson 1981], which assure the atomicity and persistence of a sequence of operations on a set of data. With multiple, concurrently executing transactions, it is guaranteed that the transactions do not interfere with each other. This design requires a large amount of complex software. Furthermore, all participants must trust a single entity, the coordinator. A crash or foul play by the coordinator may block all participants forever. Even in so-called non-blocking commit protocols, where a crash of a participant cannot block other participants, foul play can cause the eternal blocking and thus inavailability of others. In our scheme it is not possible that one or more participants block other participants.

Replication protocols in traditional systems are usually built on top of atomic transactions to ensure that the multiple copies of an object are identical. Complex software in each service is required to deal with unavailable and recovering copies. The performance of updates is generally poor.

In Amoeba a different division of complexity has led to much simpler and more efficient implementation of services. The common code for transactions and replication, which in traditional systems is part of each service, has been ‘‘factored out’’ and placed in the directory service. The only constraint placed on these services is that the

objects are immutable. That is, an update operation on an object results in a new object. Objects, as it were, are versions of data.

Before describing what functionality needs to be added to the directory service, we will briefly discuss an example of an immutable object service, namely the Amoeba file service, called the Bullet service [Van Renesse et al. 1989].

The Bullet service maintains simple files, namely sequences of bytes. After creating a file, data may be added or modified until the file is committed. An existing file may also be opened, meaning that an uncommitted *copy* is created which may be modified. After being committed, a file is immutable, and exists independently of earlier versions of the file. To make a new version of a file accessible to other users, the user updates the corresponding entry in the directory to contain the new capability. The Bullet service has no notion of commits on multiple files, nor of replication.

It is the directory service that makes the updates on a file or a set of files look indivisible, by supporting atomic operations to look up or change the capabilities of a set of files. To change a name-to-capability mapping the previous capability must also be specified so the directory service can check if any other applications have updated the file in the meantime. This serializability check implements optimistic concurrency control [Kung and Robinson 1981]. (Pessimistic concurrency control can be implemented on top of this by creating lock objects, but this is outside the scope of this paper.)

Replication of immutable objects is relatively simple. The directory service just maps a name to a set of capabilities instead of to just one. Each capability points to a distinct copy of the object, possibly geographically far apart. An update atomically replaces the set of capabilities, so users always see the old or the new set, and never a mixture. New copies can be added to the set at any time. This idea was first implemented in the Eden R2D2 directory service, although in Eden different replication protocols were used [Pu et al. 1986].

3. Design

An Amoeba directory is a two-dimensional matrix. Each row and column is labeled by an ASCII string. Within a directory all row and column names are unique. Also associated with each row is a set of capabilities. Each row maps its ASCII name to the set of capabilities in that row. The columns describe the access rights of the different groups that have access to the objects.

For example, a directory could be set up with three columns: the owner, the owner's group, and others, as in UNIX.† (see Figure 2). This directory handles the binding for two files, *README* and *x.c*, and a directory *public*. The owner of the directory has defined three UNIX-like access groups: the owner, group members, and others with whom the owner may share these objects. Each entry contains a set of three rights mask bits, for example, for read, write, and delete respectively. In our ex-

† UNIX is a Registered Trademark of AT&T in the USA and other countries.

ample, users that are in the same group as the owner of this directory are not allowed to write or delete the file *x.c*. Others are not allowed any access to this file. To prevent accidents, the owner has also turned off the delete right.

name	cap set	*owner*	*group*	*others*
README	file 1	111	100	100
public	directory	111	110	100
x.c	file 2	110	100	000

Fig. 2. An example directory. In reality the rights masks are 8 bits wide.

Although the objects pointed to by the *owner*, *group*, and *other* columns is identical, the capabilities for them are not, because they have different access rights. Thus these capabilities will have different Rights and Check Fields.

A capability for a directory specifies a particular column (or columns). A capability good only for say *group* members will not return capabilities with all the rights bits on, but will restrict them according to the rights mask in the *group* column. The directory service can generate capabilities for the various columns on-the-fly as needed, but it can also decide to cache them, at its discretion.

The entry for a particular row may contain multiple capabilities, one for each replica of the object. When a new (name, capability) pair is installed, the owner can say, ''When you have time, please make *n* copies of the object.'' The services where the objects have to be generated can be specified. Having the directory service replicate objects at its leisure is called *Lazy Replication*.

When a new version of an object is available, the owner presents the directory service with a new (name, capability) pair. This action invalidates all the old replicas, which the directory service deletes by sending messages to the appropriate services. The directory service also creates new replicas. All of this work is done automatically by the directory service.

4. Operations on Directories

The bits in the *Rights Field* in a capability for a directory are defined as follows

bits 0-5 These bits tell which columns in the directory may be accessed for look-up. Bit 0 is set if the first column may be accessed, bit 1 if the second column may be accessed, etc. to a maximum of six columns. The rights mask to be used for creating the capability in a look-up operation is formed by OR-ing together the masks in the accessible columns together.

bit 6 This bit gives the right to modify the directory, that is, adding, deleting and changing rows in the directory.

bit 7 This bit gives the right to delete a directory. For safety only empty directories may be removed.

Note that other directory operations, such as listing the names of rows and columns, are available to any holder of a capability. For a complete listing of the directory operations see Figure 3.

Function	Arguments	Returns	Rights Required	Description
CREATE-DIR	Service Column Names	Directory	—	Create a directory
DELETE-DIR	Directory	—	Delete	Remove a directory, if empty
LIST-DIR	Directory	Row Names Column Names Rights Masks	—	List a directory
CREATE-ROW	Row Capability Set Rights Masks	—	Modify	Add a row to a directory
CHMOD-ROW	Row Rights Masks	—	Modify	Replace the rights masks in a row
DELETE-ROW	Row	—	Modify	Delete a row from a directory
LOOKUP-SET	Rows	Capability Sets	—	Look up the capabilities in a set of rows
INSTALL-SET	Rows Capability Sets Current Capabilities	—	Modify	Replace the capabilities in a set of rows

Fig. 3. The directory functions. A ‘‘Directory’’ is a capability set to access an Amoeba directory. A ‘‘Row’’ is a <directory, name> pair.

The CREATE-DIR function creates a directory at the suite of directory servers specified by *service*. The number and names of the columns are specified. A set of capabilities describing the directory is returned. This set is called ‘‘Directory’’ in Figure 3. The DELETE-DIR operation removes the directory but only if it is empty to prevent accidental loss of capabilities and their corresponding objects. LIST-DIR returns a list of the names of the rows and the columns, and the rights masks in the directory.

CREATE-ROW adds a row to a directory. A row is described by the capability set of the directory and the name within that directory. A capability set and a rights mask for each column is specified. CREATE-ROW is the normal way to install a new file in a directory. CHMOD-ROW changes the rights mask in an existing row, and DELETE-ROW deletes a row from a directory.

LOOKUP-SET returns for each row specified the capability set in the row, after restricting the rights according to the rights masks in the accessible columns. This is done atomically for the whole list. Instead of simple names, *path-names* may be specified. These are simple names separated by slashes, such as *a/b/c*. The effect is that if looking up *a* results in a new directory, *b* is looked up in it, resulting in yet another directory in which *c* is looked up. If either *a* or *b* is not a valid directory, an error is returned.

The INSTALL-SET function is to manipulate the capability sets in (existing) rows. For each row the following is specified:

- The capability set of the directory that contains the row;
- The name of the row within that directory;
- The new master capability set for that row;
- The Current Capability, which should be in the current master capability set.

The directory service checks, for each row, whether the Current Capability is indeed part of the current capability set or not. If not, the complete install request will not be executed. The idea is that the application specifies on what version an update is based. If another application has updated this version in the meantime (since the LOOKUP-SET), the INSTALL-SET will fail, thus preventing inconsistencies.

5. Implementation

Because the directory service is a very important service within Amoeba, a reliable implementation is required. The directory service is duplicated to achieve this reliability. It consists of two almost identical processes, called *servers*, which run on different processors. They communicate to keep their directories consistent. A crash of one of the servers does not lead to unavailability of the service. It is also possible to bring a server down, install new code, and boot it again without the service being interrupted.

The directory service uses two Bullet file servers and two disk servers for writing raw disk blocks. In practice the disk servers are the same ones that the Bullet servers use, albeit using different disk partitions. The servers use Bullet files to store the directories, one copy on each server. The raw disk partitions are used to store administrative data. The resulting configuration is shown in Figure 4. Currently the disks are 800 Mbytes each, of which the directory servers use one megabyte for the administrative data. The processors used are Motorola 68020s and are connected by an Ethernet. The

Bullet servers each use 16 Mbytes of RAM memory for whole file caching, whereas the directory servers only use 2 Mbytes.

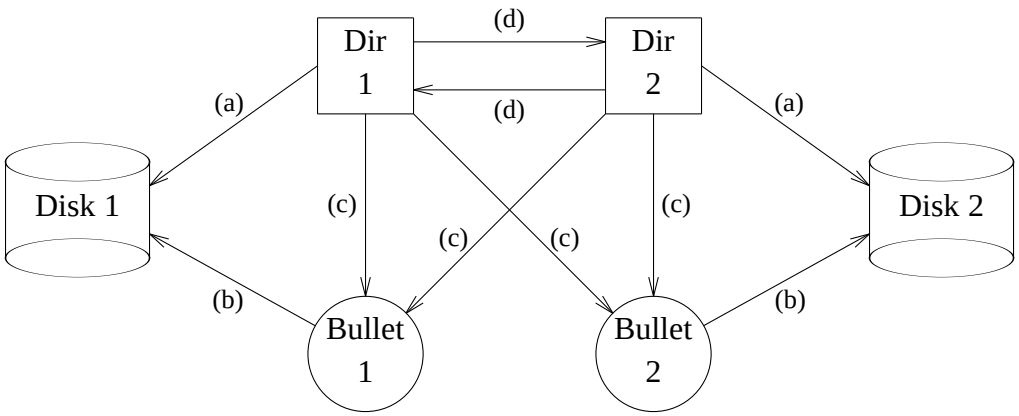

Fig. 4. The configuration. The boxes are the directory servers, while the circles represent the Bullet file servers. The arrows imply the client/server relationship. (a) Administrative Data; (b) Directories; (c) Files; (d) Intentions and other internal messages.

The administrative data is stored on a raw disk partition of n fixed-length blocks. Blocks 1 to $n-1$ contain a table of (Capability 1, Capability 2) pairs. The *Object Number* in a directory capability indexes into this table. The capabilities in the table point to two identical Bullet files that store the directory and the random number for access protection. An update of the directory means that the capabilities have to be replaced by the capabilities for the new Bullet files. These capabilities are actually overwritten, which is why raw disk and not Bullet files are used (Bullet files are immutable).

Block 0 is the *commit block* (see Figure 5). *Alone* is a boolean flag that indicates whether the server is functioning alone or together with another server. During normal operation the flag should be *false*, but if this server knows for sure that the other server is unavailable, *Alone* will be set to *true*. If *Alone* is true, the server can perform operations for users without the consent of the other server. When the other server recovers, the flag is set to false again.

Each time an update operation is performed, the *Sequence Number* is increased. The Sequence Number is sometimes required in recovery operations to tell which of the two servers have the most recent version of the administration file. The rest of the commit block is an intentions list that may or may not have been performed yet. Each entry in the list specifies the object number of the directory to be replaced, and the capabilities of the Bullet files that store the new directory. This information makes updates to the administrative data crash resistant.

The protocol that is used to keep directories consistent is depicted in Figure 6. In this protocol the replication and concurrency control are combined. The *initiator* is the

Alone	Sequence Number	# Intentions
Object number	Capability 1	Capability 2
.	.	.
.	.	.
.	.	.

Fig. 5. Layout of the commit block.

directory server that got a user request that involved the update of one or more directories. Read operations (LIST-DIR and LOOKUP-SET) can be handled by either server without the need for communication between the servers. The initiator records that it is busy, and creates the new versions of the directories. It creates, in its own internal RAM memory, a new version of the commit block with the capabilities of the new directories in the intention list and an increased Sequence Number.

If the server thinks it is not alone it will have to get the consent of the other server. It sends the new commit block to the other server and waits for a reply. If the request is rejected, this means that the other server also wants to do an update. The initiator undoes all operations and tries again. To prevent a second clash, one of the servers will always wait a second before retrying.

If no reply arrives within a certain time, the initiator makes sure that the other server has really crashed. A reliable method is to reset the processor that runs the server, thus killing the server. When the initiator knows for sure that the other server is unavailable, it sets the *Alone* flag in the RAM copy of the commit block.

Now, in either case, it writes the commit block to disk. At this point the operation has officially succeeded, and a reply is sent to the client. The initiator and the other server, if alive, have some administrative affairs left to deal with. The other server still has to write the commit block to disk. Both have to execute their intentions in their capability pair table on disk. The old Bullet files still need to be removed. This is always done by the same server unless it is unavailable.

Recovery involves copying the administration file and comparing the Sequence Numbers. When they are equal, both servers can continue operation as usual. Using Markov analysis the availability of the directory service has been calculated to be at least $p + p^2 - p^3$ [Van Renesse and Tanenbaum], where p is the combined availability of a single directory, Bullet, and disk server, thus leading to very good availability if the availability of the components is reasonably high.

The directory service also applies Lazy Replication internally to improve performance and update availability. Instead of creating the two replicas for directories at once, it initially only create one copy. Thus between the time the user requests an operation and the time that the reply is received, only one Bullet file is created, and one

```
Initiator:
       busy := TRUE;
       create new directories on Bullet files;
       Sequence Number := Sequence Number + 1;
       put Bullet capabilities in intention list;
       IF (NOT Alone) THEN
          send intention to Other Server;
          IF (rpc-failure) THEN
             reset Other Server;
             Alone := TRUE
          FI
          IF (reply = rejected) THEN
             remove newly created Bullet files;
             Sequence Number := Sequence Number - 1;
             IF (me = 2) THEN wait 1 second FI
             try again
          FI
       FI
       write commit block to disk;
       send reply to client;
       execute the intentions;
       IF (me = 1 OR Alone) THEN remove old Bullet files FI
       busy := FALSE

Other Server:
       IF busy THEN
          RETURN rejected
       ELSE
          RETURN ok;
          write commit block to disk;
          execute the intentions;
          IF (me = 1) THEN remove old Bullet files FI
       FI

Recovery:
       IF (NOT Alone) THEN
          copy administration file;
          tell Other Server to clear Alone flag;
             (if the Sequence Numbers are the same)
          IF (NOT ok) THEN try again FI
       FI
```

```
continue operation as usual
```

Fig. 6. The duplication protocol.

block is written to disk (the commit block). Later, when the directory service has some spare time, the other replica is generated. Since the directory service uses a large cache, the crash of a Bullet server containing the only copy of a directory does not necessarily affect availability.

6. Comparison with Other Directory Services

Many directory services have been built. Functionally, they all associate a certain key with a value. Keys are usually ASCII names, and values are often objects (usually files) or object identifiers such as capabilities. In this section we will discuss some of the more interesting systems, and how they compare with the Amoeba directory service.

The CAP computer system [Wilkes and Needham 1979] is a capability-based computer and operating system. Directories map ASCII names to single capabilities. Directories have four columns, with rights masks in each column. Unlike Amoeba, CAP processes never see capabilities; capabilities are managed by the CAP system and users manipulate them indirectly. There is no support for replication or concurrency control among objects.

R2D2 is the Eden directory service [Pu et al. 1986], which, like in Amoeba, maps ASCII names to sets of capabilities. Unlike in Amoeba, the directory service does not support protection. Like in CAP, capabilities are managed by the system. Directories are replicated using an algorithm called Regeneration. Other than mapping names to sets of capabilities, the directory service supports neither replication nor atomic updates. These functions are available through separate software systems, however.

COSMOS [Nicol et al. 1987], CFS [Gifford et al. 1988], and FUSS [Mullender and Tanenbaum 1985], all support immutable files, which are identified by an object identifier and a version number. Not specifying the version number results in a default version, usually the latest. They do not use a directory service for version management, replication, or atomic updates of multiple objects, although COSMOS and FUSS have separate mechanisms available.

Grapevine [Birrell 1982; Schroeder 1984] and its successor Clearinghouse are directory services that span the world. These systems are not object-based, but map names of users, institutions, or services to information about these entities, such as user passwords, mail addressing, and resource location. The directory graph is a tree, of which only the top levels (near the root) are replicated. Consistency among the replicas is not guaranteed.

7. Conclusion

The Amoeba file system is a radical departure from traditional distributed file systems. It consists of dumb but extremely fast file servers, which store immutable files, and intelligent directory servers, which handle naming, protection, replication, and consistency control in a clean and integrated way.

The directory servers can also be used to manage objects other than files. This ability means that the implementers of new object types do not have to write code to handle naming, protection, replication, and consistency. The directory service does it all for them. We consider this approach a significant advance over previous systems, in which these functions had to be implemented over and over again for each new object type introduced into the system.

Not only has this design led to a highly flexible and general system, it has also led to a very efficient one. The divison of labor described in the paper has resulted in a file system capable of transfering 677 Kbytes/second. Further performance results have been presented in [Van Renesse et al. 1988, 1989].

8. References

Birrell, A. D., "Grapevine: An exercise in distributed computing," *Comm. ACM*, Vol. 25, No. 4, pp. 260-274, Apr. 1982. Cited on page 11.

Evans, A., Kantrowitz, W., and Weiss, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM*, Vol. 17, No. 8, pp. 437-442, Aug. 1974. Cited on page 2.

Gifford, D. K., Needham, R. M., and Schroeder, M. D., "The Cedar File System," *Comm. ACM*, pp. 288-298, Mar. 1988. Cited on page 11.

Kung, H. T. and Robinson, J. T., "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems*, Vol. 6, No. 2, pp. 213-226, June 1981. Cited on page 4.

Lampson, B. W., "Atomic Transactions," pp. 246-265, in *Distributed Systems — Architecture and Implementation*, Springer Verlag, West Berlin, 1981. Cited on page 3.

Mullender, S. J. and Tanenbaum, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, Vol. 8, No. 5-6, pp. 421-432, Oct. 1984. Cited on page 1.

Mullender, S. J. and Tanenbaum, A. S., "A Distributed File Service Based on Optimistic Concurrency Control," *Proc. of the 10th ACM Symp. on Operating System Principles*, pp. 51-62, Orcas Island, WA, Dec. 1985. Cited on page 11.

Mullender, S. J. and Tanenbaum, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, Vol. 29, No. 4, pp. 289-300, Mar. 1986. Cited on page 1.

Nicol, J. R., Blair, G. S., Shepherd, W. D., and Walpole, J., "An Approach to Multiple Copy Update Based on Immutability," *Proc. of the IFIP Conf. on Distr. Processing*, Amsterdam, Oct. 1987. Cited on page 11.

Pu, C., Noe, J. D., and Proudfoot, A., "Regeneration of Replicated Objects: a Technique and Its Eden Implementation," *Proc. of the Int. Conf. on Data Engineering*, pp. 175-186, Feb. 1986. Cited on pages 4, 11.

Schroeder, M. D., "Experience with Grapevine," *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 3-23, Feb. 1984. Cited on page 11.

Tanenbaum, A. S., Mullender, S. J., and Van Renesse, R., "Using Sparse Capabilities in a Distributed Operating System," *Proc. of the 6th Int. Conf. on Distr. Computing Systems*, pp. 558-563, Cambridge, MA, May 1986. Cited on page 1.

Van Renesse, R. and Tanenbaum, A. S., "Lazy Replication," VU Technical Report, Vrije Universiteit, Amsterdam, To Appear. Cited on page 9.

Van Renesse, R., Tanenbaum, A. S., and Wilschut, A., "The Design of a High-Performance File Server," *Proc. of the 9th Int. Conf. on Distr. Computing Systems*, pp. 22-27, Newport Beach, CA, June 1989. Cited on pages 4, 12.

Van Renesse, R., Van Staveren, J. M., and Tanenbaum, A. S., "The Performance of the World's Fastest Distributed Operating System," *ACM Operating Systems Review*, Vol. 22, No. 4, pp. 25-34, Oct. 1988. Cited on pages 1, 12.

Wilkes, M. V. and Needham, R. M., "The Cambridge CAP Computer and Its Operating System," North-Holland, Amsterdam, 1979. Cited on page 11.