The Design of a High-Performance File Server

*Robbert van Renesse**
Andrew S. Tanenbaum
Annita Wilschut‡

Dept. of Computer Science
Vrije Universiteit
The Netherlands

ABSTRACT

The *Bullet* server is an innovative file server that outperforms traditional file servers like SUN's NFS by more than a factor of three. It achieves high throughput and low delay by a radically different software design than current file servers in use. Instead of storing files as a sequence of disk blocks, each Bullet server file is stored contiguously, both on disk and in the server's RAM cache. Furthermore, it employs the concept of an immutable file, to improve performance, to enable caching, and to provide a clean semantic model to the user. The paper describes the design and implementation of the Bullet server in detail, presents measurements of its performance, and compares this performance with other well-known file servers running on the same hardware.

1. INTRODUCTION

Traditional file systems were designed for small machines, that is, computers with little RAM memory and small disks. Emphasis was on supporting large files using as few resources as possible. To allow dynamic growth of files, files were split into fixed size blocks scattered all over the disk. Blocks would be dynamically allocated to files, such that a large file would be scattered all over a disk. Performance suffered since each block had to be separately accessed. Also the block management introduced high overhead: indirect blocks were necessary to administer the files and their blocks. A small part of the computer's little memory was used to keep parts of files in a RAM cache to make access more efficient.

Today the situation has changed considerably. Machines have enormous RAM memories and huge disks. Files usually fit completely in memory. For example, memory sizes of at least 16 Megabytes are common today, enough to hold most files encountered in practice. Measurements [1] show that the median file size in a UNIX† system is 1 Kbyte and 99% of all files are less than 64 Kbytes. File systems, however, have

* This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.) under grant 125-30-10.
‡ Current affiliation: Dept. of Computer Science, University of Twente, Enschede, The Netherlands
†UNIX is a Trademark of Bell Laboratories.

not changed yet. Files are still subdivided into blocks. To take some advantage of new technology, the size of blocks has been increased, and the memory caches have been enlarged. This has led to a marginal performance improvement.

As part of the Amoeba distributed operating system project [2] we have designed and implemented a file server that is intended for current and future computer and disk technology. We have devoted considerable energy to making the file server fast. Since we believe we have achieved this goal, we have named it the *Bullet file server*. Among its features are support for replication, caching, and immutability.

This paper has five sections. In the following section we will present the architectural model of the Bullet file service. In section three we present the implementation of the server, including the data structures and interfaces. The performance of the file server is the subject of section four. We will compare the the performance of the Bullet file server with other files servers, such as SUN NFS. Section five contains our conclusions.

2. ARCHITECTURAL MODEL

The basic idea behind the design of the Bullet file server is to do away with the block model. In fact, we have chosen the extreme, which is to maintain files contiguously throughout the system. That is, files are contiguously stored on disk, contiguously cached in RAM, and kept contiguously in processes' memories. This dictates the choice for whole file transfer. As a consequence, processors can only operate on files that fit in their physical memory. This affects the way in which we store data structures on files, and how we assign processors to applications. Since most files (about 75%) are accessed in entirety [4], whole file transfer optimizes overall scaling and performance, as has also been reported in other system that do whole file transfer, such as in the Andrew ITC file system [5].

Another design choice, which is closely linked to keeping files contiguous, is to make all files immutable. That is, the only operations on files are creation, retrieval, and deletion; there are no update-in-place operations. Instead, if we want to update a data structure that is stored on a file, we do this by creating a new file holding the updated data structure. In other words, we store files as sequences of versions. Note that as we do whole file transfer anyway, this puts no performance penalties on the file server. Version mechanisms have positive influences on caching, as reported in the Cedar File System [6], and on replication. It also presents the possibility of keeping versions on write-once storage such as optical disks. The version mechanism is itself quite interesting, and is discussed in [7].

For most applications this model works well, but there are some applications where different solutions will have to be found. Each append to a log file, for example, would require the whole file to be copied. Similarly, for data bases, a small update might incur a large overhead. For log files we have implemented a separate server. Data bases can be subdivided over many smaller Bullet files, for example based on the identifying keys.

Throughout the design we have strived for performance, scalability, and availability. The Bullet file server is the main storage server for the Amoeba distributed operating system, where these issues are of high importance [8]. Performance can only be achieved if the management of storage and replication are low, and the model of contiguity and immutability corresponds to how files are usually accessed. Scalability involves both geographic scalability—Amoeba currently runs in four different countries—and quantitative scalability—there may be thousands of processors accessing files. Availability implies the need for replication.

Since these issues correlate heavily with those of the Amoeba distributed operating system, we will first devote a section to Amoeba. In this section we will also describe the naming service of Amoeba, which plays a role in how data structures, especially large ones, may be stored efficiently on immutable files [7]. In the following section we will describe the Bullet file interface.

2.1. Amoeba

Amoeba [2,3] is a distributed operating system that was designed and implemented at the Vrije Universiteit in Amsterdam, and is now being further developed there and at the Centre for Mathematics and Computer Science, also in Amsterdam. It is based on the object model. An object is an abstract data type, and operations on it are invoked through remote procedure calls. Amoeba consists of four principal components:

- Workstations
- Dynamically allocatable processors
- Specialized services
- Gateways

Workstations provide the user interface to Amoeba and are only involved with interactive tasks such as command interpretation and text editing. Consequently they do not deal with very large or dynamically changing files. The dynamically allocatable processors together form the so-called *processor pool*. These processors may be allocated for compiling or text formatting purposes, or for distributed or parallel algorithms. Among other applications, we have implemented a parallel make [10] and parallel heuristic search [11].

Specialized servers include filing servers such as the Bullet file server, and the directory server. The directory server is used in conjunction with the Bullet server. It's function is to handle naming and protection of Bullet server files and other objects in a simple, uniform way. Servers manage the Amoeba objects, that is, they handle the storage and perform the operations. Gateways provide transparent communication among Amoeba sites currently operating in four different countries (The Netherlands, England, Norway, and Germany).

All objects in Amoeba are addressed and protected by capabilities [3, 12]. A capability consists of four parts:

1) The *server port* identifies the server that manages the object. It is a 48-bit location-independent number that is chosen by the server itself and made known to the server's potential clients.
2) The *object number* identifies the object within the server. For example, a file server may manage many files, and use the object number to index in a table of *inodes*. An inode contains the position of the file on disk, and accounting information.

3) The *rights field* specifies which access rights the holder of the capability has to the object. For a file server there may be a bit indicating the right to read the file, another bit for deleting the file, and so on.

4) The *check field* is used to protect capabilities against forging and tampering. In the case of the file server this can be done as follows. Each time a file is created the server generates a large random number and stores this in the inode for the file. Capabilities for the file can be generated by taking the server's port, the index of the file in the inode table, and the required rights. The check field can be generated by taking the rights and the random number from the inode, and encrypting both. If, later, a client shows a capability for a file, its validity can be checked by decrypting the check field and comparing the rights and the random number. Other schemes are described in [12]. Capabilities can be cached to avoid decryption for each access.

Although capabilities are a convenient way for addressing and protecting objects, they are not usable for human users. For this the *directory service* maps human-chosen ASCII names to capabilities. Directories are two-column tables, the first column containing names, and the second containing the corresponding capabilities. Directories are objects themselves, and can be addressed by capabilities. By placing directory capabilities in directories an arbitrary naming structure can be built at the convenience of the user. The directory service provides a single global naming space for objects. This has allowed us to link multiple Bullet file servers together providing one single large file service that crosses international borders [13, 14].

2.2. Bullet Server Interface

The simple architectural model of the file service is reflected in its simple interface. Whole file transfer eliminates the need for relatively complicated interfaces to access parts of files. Immutability eliminates the need for separate update operators. Version management is not part of the file server interface, since it is done by the directory service [7].

The Bullet interface consist of four functions:

- BULLET.CREATE(SERVER, DATA, SIZE, P-FACTOR) → CAPABILITY
- BULLET.SIZE(CAPABILITY) → SIZE
- BULLET.READ(CAPABILITY, &DATA)
- BULLET.DELETE(CAPABILITY)

The BULLET.CREATE function is the only way to store data on a Bullet server. The SERVER argument specifies which Bullet server to use. This enables users to use more that on Bullet server. The DATA and SIZE arguments describe the contents of the file to be created. A capability for the file is returned for subsequent usage.

P-FACTOR stands for Paranoia Factor. It is a measure for how carefully the file should be stored before BULLET.CREATE can return to the invoker. If the P-FACTOR is zero, BULLET.CREATE will return immediately after the file has been copied to the file server's RAM cache, but before it has been stored on disk. This is fast, but if the server crashes shortly afterwards the file may be lost. If the P-FACTOR is one, the file will be stored on one disk before the client can resume. If the P-FACTOR is N, the file will be stored on N disks before the client can resume. This requires the file server to have at least N disks available for replication. At present we have two disks.

The BULLET.SIZE and BULLET.READ functions are used to retrieve files from a server. First BULLET.SIZE is called to get the size of the file addressed by CAPABILITY, after which local memory is allocated to store its contents. Then BULLET.READ is invoked to get the contents, where &DATA is the address of the allocated local memory. Alternatively a section of the virtual address space can be reserved, after which the file can be mapped into the virtual memory of the process. In that case the underlying kernel performs the BULLET.READ function. BULLET.DELETE allows files to be discarded from the file server.

3. IMPLEMENTATION

Keeping files contiguous (*i.e.,* not splitting them up in blocks) greatly simplifies file server design. Consequently, the implementation of the file server can be simple. In this section we will discuss an implementation on a 16.7 MHz Motorola 68020-based server with 16 Mbytes of RAM memory and two 800 Mbyte magnetic disk drives. We will describe the disk layout of the file server, the file server cache, and how replication is done.

The disk is divided into two sections. The first is the inode table, each entry of which gives the ownership, location, and size of one file. The second section contains contiguous files, along with the gaps between files. Inode entry 0 is special, and contains three 4 byte integers:

1) *block size* : the physical sector size used by the disk hardware;

2) *control size* : the number of blocks in the inode table;

3) *data size* : the number of blocks in the file table;

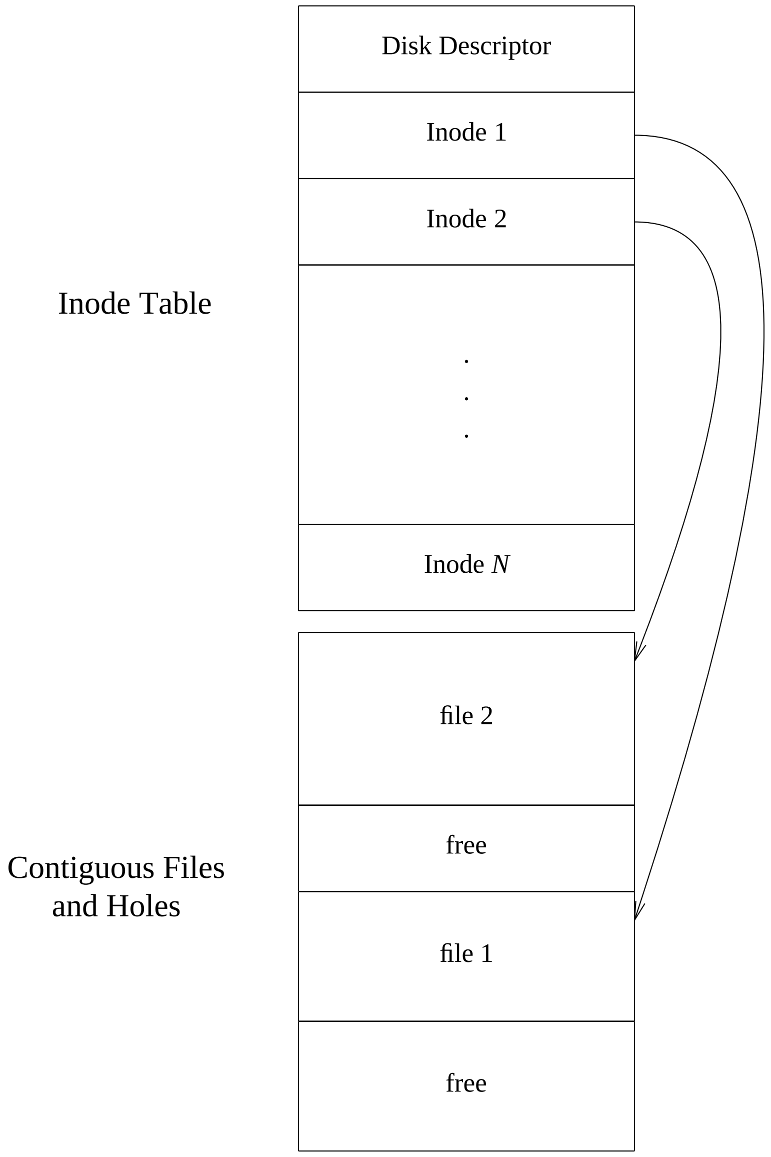

Fig. 1. The Bullet disk layout.

The remaining inodes describe files. An inode consist of four fields:

1) A 6-byte random number that is used for access protection. It is essentially the key used to decrypt capabilities that are presented to the server.

2) A 2-byte integer that is called the *index*. The index has no significance on disk, but is used for cache management and will be described later.

3) A 4-byte integer specifying the first block of the file on disk. Files are aligned on blocks (sectors).

4) A 4-byte integer giving the size of the file in bytes.

When the file server starts up, it reads the complete inode table into the RAM inode table and keeps it there permanently. By scanning the inodes it can figure out which parts of disk are free. It uses this information to build a free list in RAM. Also unused inodes (inodes that are zero-filled) are maintained in a list. While scanning the inodes, the file server performs some consistency checks, for example to make sure that files do not overlap. All of the server's remaining memory will be used for file caching. At this time the file server is ready for operation and starts awaiting client requests.

A separate table in RAM maintains the administration of the cached files. The entries in this table have a slightly different format from those on the disk, and are called *rnodes*. An rnode contains the following information:

1) The inode table index of the corresponding file;

2) A pointer to the file in RAM cache, if present;

3) An *age* field to implement an LRU cache strategy.

The free rnodes and free parts in the RAM cache are also maintained using free lists.

Client requests basically come in three varieties: reading files, creating files, and deleting files. To read a file the client has to provide the capability of the file. The object number in the capability indexes into the inode table to find the inode for the file. Using the random number in the inode, and the check field in the capability, the right to read the file by this client can be checked. Next the index field in the inode is inspected to see whether there is a copy of the file in the RAM cache. If the index in the inode is non-zero, there is a copy in the server's RAM cache. The index is used to locate an rnode, which describes where to find the file in memory. Since the file is contiguously kept in RAM, it can be sent to the client in one RPC operation. The *age* field is updated to reflect the recent access of the file.

If the index in the inode is zero, the file is not in memory and has to be loaded from disk. First the memory free list is searched to see if there is a part large enough to hold the file. If not, the least recently accessed file is removed from the RAM cache, found by checking the *age* fields in the rnodes. This is done by re-claiming the rnode, freeing the associated memory, and clearing the index field in the corresponding inode, repeating until enough memory is found. Then an rnode is allocated for this file, and its fields initialized. The index field in the inode of the file is set to the rnode index. Then the file can be read into the RAM cache. The client read operation can now proceed as with files that were already cached.

Creating files is much the same as reading files that were not in the cache. A large enough part of cache memory has to be allocated to hold the file, after which it can be filled with data specified by the client. Also, an inode and a free part in the disk data section have to be allocated. For this we use a first fit strategy. In our implementation we use a write-through caching scheme, that is, we immediately write the file to disk. The new inode, complete with a new random number, is immediately written as well. For this the whole disk block containing the inode has to be written. The new random number is used to create a capability for the user, which is returned in a reply message.

In our hardware configuration we have two disks that we use as identical replicas. One of the disks is the main disk on which the file server reads. Disk writes are performed on both disks. If the main disk fails, the file server can proceed uninterruptedly by using the other disk. Recovery is simply done by copying the complete disk. The P-FACTOR in the create operation is used to determine where in the execution to send a reply back to the client.

Deleting a file involves checking the capability, freeing an inode by zeroing it and writing it back to the disk. If the file is in the cache, the space in the cache can be freed. The disk free list in RAM has to be updated to include the part previously occupied by the file.

At first glance, it appears that storing files contiguously in memory and on disk is very wasteful due to the external fragmentation problem (gaps between files). However, the fragmentation in memory can be alleviated by compacting part or all of the RAM cache from time to time. The disk fragmentation can also be relieved by compaction every morning at say 3 am when the system is lightly loaded.

However, it is our belief that this trade-off is not unreasonable. In effect, the conscious choice of using contiguous file may requiring buying, say, an 800 MB disk to store 500 MB worth of files (the rest being lost to fragmentation unless compaction is done). Since our scheme gives an increased performance of a factor of 3 (as described in the next section) and disk prices rise only very slowly with disk capacity, a relatively small increment in total file server cost gives a major gain in speed.

The complete code of the file server is less than 30 pages of C. The MC68020 object size of the server, including all library routines, is 23 Kbytes. This small and simple implementation has resulted in a file server that has been operational flawlessly for over a year. The most vulnerable component of the server is the disk, but because of its replication, the complete file server is highly reliable.

4. PERFORMANCE AND COMPARISON

Figure 1 gives the performance of the Bullet file server. In the first column the delay and bandwidth for read operations are shown. The measurements have been done on a normally loaded Ethernet from a 16 MHz 68020 processor. In all cases the test file will be completely in memory, and no disk accesses are necessary. In the second column a create and a delete operation together is measured, and the file is written to both disks. Note that both creation and deletion involve requests to two disks.

Delay (msec)

File Size	READ	CREATE+DEL
1 byte	2	94
16 bytes	2	97
256 bytes	2	98
4 Kbytes	7	109
64 Kbytes	109	331
1 Mbyte	1970	2700

(a)

Bandwidth (Kbytes/sec)

File Size	READ	CREATE+DEL
1 byte	0.5	0.01
16 bytes	8	0.16

256 bytes	110	3
4 Kbytes	559	37
64 Kbytes	587	193
1 Mbyte	520	379

(b)

Fig. 2. Performance of the Bullet file server for read operations, and create and delete operations together. The delay in msec (a) and bandwidth in Kbytes/sec (b) are given.

To compare this with the SUN NFS file system, we have measured reading and creating files on a SUN 3/50 using a remote SUN 3/180 file server (using 16.7 MHz 68020s and SUN OS 3.5), equipped with a 3 Mbyte buffer cache. The measurements were made on an otherwise idle processor on a normally loaded Ethernet. To disable local caching on the SUN 3/50, we have locked the file using the SUN UNIX *lockf* primitive. The read test consisted of an *lseek* followed by a *read* system call. The write test consisted of consecutively executing *creat*, *write*, and *close*. The SUN NFS file server uses a write-through cache, but writes the file to one disk only. The results are depicted in Fig. 2.

Delay (msec)

File Size	READ	CREATE
1 byte	13	228
16 bytes	10	230
256 bytes	11	229
4 Kbytes	20	246
64 Kbytes	316	1284
1 Mbyte	6230	32755

(a)

Bandwidth (Kbytes/sec)

File Size	READ	CREATE
1 byte	0.1	0.004
16 bytes	1.5	0.067
256 bytes	23	1
4 Kbytes	204	16
64 Kbytes	203	50
1 Mbyte	164	31

(b)

Fig. 3. Performance of the SUN NFS file server for read and create operations. The delay in msec (a) and bandwidth in Kbytes/sec (b) are given.

These measurements include both the communication time and the file server time. Since Amoeba uses a dedicated processor for the file server, it is impossible to separate communication and file server performance. Observe that reading and creating 1 Mbyte NFS files result in lower bandwidths than reading and creating 64 Kbyte NFS files. The Bullet file server performs read operations three to six times better than the SUN NFS file server for all file sizes. Although the Bullet file server stores the files on two disks, for large files the bandwidth is ten times that of SUN NFS. For very large files (> 64 Kbytes) the Bullet server even achieves a higher bandwidth for writing than SUN NFS achieves for reading files.

5. DISCUSSION AND CONCLUSIONS

The simple architectural model of immutable files that are kept contiguous on disk, in memory, and on the network, results in a major performance boost. Whole file transfer minimizes the load on the file server and on the network, allowing the service to be used on a larger scale [5]. Replication for availability is relatively easy. The simple implementation of the server also renders high availability. Immutability turned out to be a satisfactory model for most of our storage requirements. Recently we have implemented a UNIX emulation on top of the Bullet service supporting a wealth of existing software.

Client caching of immutable files is straightforward. Checking if a cached copy of a file is still current is simply done by looking up its capability in the directory service, and comparing it to the capability on which the copy is based.

Currently we are investigating how the Bullet file server and the Amoeba directory service can cooperate in providing a general purpose storage system. Goals of this research are high availability, consistency, performance, scalability, and minimal tradeoffs between these goals. We have extended the interface to allow generating a new file based on an existing file, such that for a small modification it is not necessary any longer to transfer the whole file, while it also allows processors with small memories to handle large files. Files still have to fit completely in the file server's memory such that performance is not affected.

6. REFERENCES

[1] Mullender, S. J. and Tanenbaum, A. S., ''Immediate Files,'' Software—Practice and Experience, Vol. 14, No. 4, pp. 365-368 (April 1984).

[2] 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 (March 1986).

[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

(October 1984).

[4] Ousterhout, J. K., Costa, H. Da, Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G., ‘‘A Trace-Driven Analysis of the UNIX 4.2 BSD File System,’’ Proc. of the 10th Symp. on Operating Systems Principles, pp. 15-24, Orcas Island, WA (December 1985).

[5] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J., ‘‘Scale and Performance in a Distributed File System,’’ ACM Trans. on Computer Systems, Vol. 6, No. 1, pp. 51-81 (February 1988).

[6] Gifford, D. K., Needham, R. M., and Schroeder, M. D., ‘‘The Cedar File System,’’ Comm. ACM, pp. 288-298 (March 1988).

[7] Renesse, R. van and Tanenbaum, A. S., ‘‘Consistency and Availability in the Amoeba Distributed Operating System,’’ VU Technical Report, Vrije Universiteit, Amsterdam (To Appear).

[8] Renesse, R. van, Staveren, J. M. van, and Tanenbaum, A. S., ‘‘The Performance of the Amoeba Distributed Operating System,’’ Software—Practice and Experience, Vol. 19, No. 3, pp. 223-234 (March 1989).

[9] Renesse, R. van, Staveren, J. M. van, 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 (October 1988).

[10] Baalbergen, E. H., ‘‘Design and Implementation of Parallel Make,’’ Computing Sys- tems, Vol. 1, No. 2, pp. 135-158 (Spring 1988).

[11] Bal, H. E., Renesse, R. van, and Tanenbaum, A. S., ‘‘Implementing Distributed Algo- rithms Using Remote Procedure Calls,’’ Proc. of the 1987 National Computer Conf., pp. 499-506, Chicago, IL (June 1987).

[12] Tanenbaum, A. S., Mullender, S. J., and Renesse, R. van, ‘‘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).

[13] Renesse, R. van, Tanenbaum, A. S., Staveren, J. M. van, and Hall, J., ‘‘Connecting RPC-Based Distributed Systems Using Wide-Area Networks,’’ Proc. of the 7th Int. Conf. on Distr. Computing Systems, pp. 28-34, West Berlin (September 1987).

[14] Renesse, R. van, Staveren, J. M. van, Hall, J., Turnbull, M., Janssen, A. A., Jansen, A. J., Mullender, S. J., Holden, D. B., Bastable, A., Fallmyr, T., Johansen, D., Mullender, K. S., and Zimmer, W., ‘‘MANDIS/Amoeba: A Widely Dispersed Object-Oriented Operating System,’’ Proc. of the EUTECO 88 Conf., pp. 823-831, ed. R. Speth, North-Holland, Vienna, Austria (April 1988).