Combining High Performance and Fault Tolerance in a Distributed File Server

Position paper for the 4th SIGOPS workshop on Fault-Tolerance Support in Distributed Systems

Sape J. Mullender

CWI, the Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam

Among the most reliable and fault tolerant components in a distributed system are storage systems. Obviously, reliability of storage systems belongs to the most researched issues in distributed computing.

Every distributed file system project is based on different assumptions about size, load, amount of sharing, and desirable semantics, making it hard to compare research results fairly.

The current Amoeba file server is the *Bullet File Server* [van Renesse, Tanenbaum, and Wilschut, 1989] which provides immutable files, is optimized for whole-file transfer and does caching at the file server. It has excellent performance for reading cached files ($1.5 + 1.5n$ ms for n kilobytes) and for sustained file I/O (680 kilobytes per second, both on read and write).

Although performance is excellent, there is room for improvement, especially in the area of fault tolerance, sharing semantics and caching. I am currently doing the back-of-the-envelope design for a new file server that will form the basis of both our normal file system and of a *complex-object server*[1] which is being designed by the database group at CWI. In addition to those desirable properties of fault tolerance, persistency, consistency, and availability, I am anxious to achieve even better performance than the Bullet server by extensive use of client and server caching.

This position paper presents some of our design ideas. Note that this is work in progress; that the design is only just starting and that an implementation of a design will not start for another six months or so.

I want to realise single-site file-sharing semantics, and provide sufficient handles for implementing efficient transaction-processing mechanisms: file locking (on byte ranges) and persistent, atomic write operations. To make single-file update easy, one can *open* a file, read and write the open file and *commit* the result. The opened file will be updated atomically on commit. There are three basic file types, *mutable* files, *immutable* files, and *open* files. Open files are derived from mutable or immutable files. After they're created, immutable files can only be read or deleted.

[1] A complex-object server stores structured objects which may and will contain references to other objects. Objects may range in size from a few bytes to many megabytes. A series of operations on a collection of objects must often be atomic and persistent in the face of crashes.

Files can be read and written via conventional read and write operations. Immutable files and open files can also be accessed using *mapped file I/O*. The idea is to integrate the mechanisms for caching and mapped file I/O to make file updates very efficient.

Naming and access control is done through the normal Amoeba capability mechanism [Mullender *et al.*, 1990].

The failure modes that I want to deal with are client failure, server failure, media failure, network failure, and power failure. The first three are assumed to occur as single-point failures and the file system should survive them with no loss of data, functionality, or availability. Individual operations, or ongoing transactions may need to be redone, however.

The server-failure model is *fail stop*, in the sense that servers are assumed neither to corrupt storage media when they crash nor send incorrect messages. What clients do in a crash is their own business. However, every message sent from a client to a server is checked for well-formedness and permission to access files in the manner requested. File server processes may use non-volatile RAM (NVR) in recovery from power failures, but not from server crashes: Tim Wilson's seven-minute presentation at SOSP-12 warned me to be careful trusting NVR after a software crash.

Network failures are those failures that render clients or servers unreachable from other clients or servers. These failures cannot always be coped with transparently. However, I feel that file operations should be allowed to continue whenever possible, even if they result in inconsistencies. Applications most be warned, however, of the possibility that file read operations may return stale data, that successful file write operations now may cause conflicts later and that transactions may use inconsistent or out-of-date information and fail when the partition is repaired.

If a client becomes disconnected from the storage service, it should be allowed to continue to operate in the following manner: read operations of cached data should succeed, but a warning error code should be issued if the data cannot be guaranteed to be consistent, and write operations should succeed with a warning error code that consistency and persistency cannot be guaranteed. Clients should be allowed to maintain a *stash*[2] of files (files that never disappear from the cache) so that they can continue in disconnected mode with all the files they need.

Power failure is the only non-single-point failure I intend to cope with. After power is restored, the system should come back up quickly (within a few minutes at most), and no data, acknowledgedly written before power went, should be lost.

The performance model is based on the following assumptions:

1. Both clients and servers must be able to cache (pieces of) files.
2. When a read-shared file is cached at a client site, it must be readable without any server interactions on the execution path of the read operation.
3. When a non-shared file is being read and written at a client site, reads for cached data must not require server interaction and (persistent) writes may have at most one server interaction and at most one disk write in their execution path.
4. For guaranteeing persistency of updates, at least two replicas of every file must be stored. For efficiency, I intend to allow one of the replicas to be in a client cache.
5. Where possible, I intend to exploit lazy data transfers: large-chunk transfer which results in whole-file transfer for small files, but not for large ones; temporarily using data in client caches as replicas while background server processes create stable-memory replicas.

[2] The word *stash* was first used by Birrell and Schroeder at the 1988 ACM SIGOPS Workshop on Autonomy and Interdependence in Distributed Systems' in Cambridge.

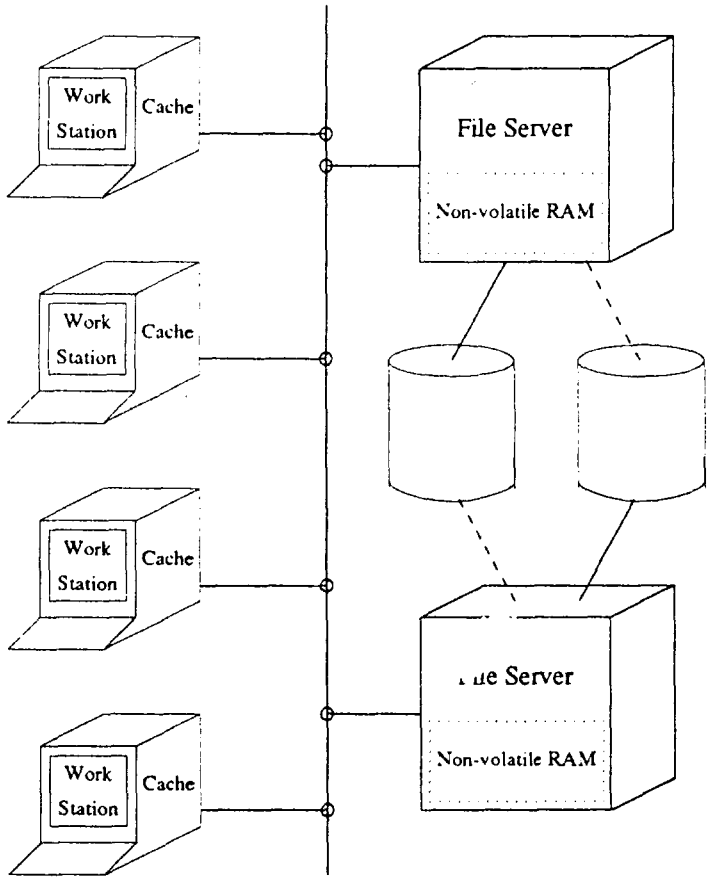

Figure 1. System structure

Points 2 and 3 essentially also say that servers have to do callbacks for data in client caches to guarantee cache consistency. Point 4 implies that client workstations will participate in a cache-coherence protocol. For robustness, the cache-coherence protocol — as well as other protocols in which client workstations participate — must be designed such that even a maliciously incorrect client workstation cannot corrupt the integrity of the file system, nor deny service to other, correctly operating workstations.

As a suitable hardware platform for a very fast highly reliable distributed file system, I propose a system structure as illustrated in Figure 1. The figure shows a number of client workstations, two file server machines and a network connecting the machines. The storage media may or may not be dual-ported to both servers; I haven't made up my mind yet what dual-porting is going to buy me. The file server machines should preferably have NVR. The client workstations have a main-memory file cache (ordinary volatile memory).

To give an idea how fast I would like operations to be, let me give the scenario I envisage for committing a mapped (open) file:

1. The application process in the client workstation sends a commit request to the file server stub on the workstation (which also acts as the client cache manager).
2. The workstation's file server takes over the memory segment containing the file's data, maps it in and pins it down. Note that no file data need be copied.
3. It then sends the data in a commit request to one of the file servers (the one that granted the lock).
4. The file server receives the data in NVR, updates the file meta data (which also resides in NVR) in an appropriately atomic manner and returns an acknowledgement to the workstation stub which sends it on to the application process.

At this point, the data is safe against single-point processor and media failures (there are two copies on independent machines) and power failures (there is one copy in battery-backed-up storage). The latency of the write operation need only be marginally more than the time it takes to do one RPC with the file data. Background processes in the file service can migrate the data to disk, freeing the NVR for new requests, and replicate the data to the other server, allowing the workstation stub to unpin its cached data.

In the normal case (a file has either a single writer or multiple readers and is already cached), cache coherence should be realized without the need for any interprocess communication in the critical path of read and write operations. The cache-coherence protocols should be designed such that workstations not obeying the protocol properly cannot disrupt service for other (honest) workstations.

The challenge of the project described above is to design a storage server that has all the desirable properties listed, but which is still conceptually simple so that one can reason about the correctness of the algorithms and so that a very fast implementation is possible.

Another challenge is to design the fault tolerance without relying too much on the correct behaviour of the client workstation. A client workstation holding an essential replica in its cache may be assumed to have written that data, so I think it is reasonable to assume the workstation can be trusted to provide the data to the file server again if that data were lost in media failure. In any case, the protocols used must be able to deal with maliciously incorrect behaviour of client workstations.

I have been inspired for these ideas by work being carried out in Cambridge by Roger Needham and his students, by Mike Burrows' ideas on locking [Burrows, 1988], by my experience with the Amoeba Bullet server [van Renesse, Tanenbaum, and Wilschut, 1989], and by the work on Coda [Howard *et al.*, 1988].

1. References

M. Burrows [1988].
Efficient Data Sharing.
Ph.D. Thesis, Cambridge University Computer Laboratory, September 1988.
Also available as Technical Report No. 153, December 1988.

J. H. Howard, M. J. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West [1988].
Scale and Performance in a Distributed File System.
ACM Transactions on Computer Systems 6 (1), 1988.

S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and J. M. van Staveren [1990].
Amoeba — A Distributed Operating System for the 1990s.
IEEE Computer 23 (5), May 1990.
To appear.

R. van Renesse, A. S. Tanenbaum, and A. Wilschut [1989].
The Design of a High-Performance File Server.
Proceedings of the Ninth ICDCS: 22–27, Newport Beach, Ca, June 1989.