# zFS - A Scalable distributed File System using Object Disks

*O. Rodeh*[*] *A. Teperman*[*†]

## Abstract

*zFS is a research project aimed at building a decentralized file system that distributes all aspects of file and storage management over a set of cooperating machines, interconnected by a high-speed network. zFS is designed to be a file system that scales from a few networked computers to several thousand machines and to be built from commodity, off-the-shelf components.*

*The two most prominent features of zFS are its cooperative cache and distributed transactions. zFS integrates the memory of all participating machines into one coherent cache. Thus, instead of going to the disk for a block of data already in one of the machine memories, zFS retrieves the data block from the remote machine. zFS also uses distributed transactions and leases, instead of group-communication and clustering software.*

*This article describes the zFShigh-level architecture and describes how its goals are achieved.*

## 1 Introduction

zFS is a research project aimed at building a decentralized file system that distributes all aspects of file and storage management over a set of cooperating machines interconnected by a high-speed network. zFS is designed to be a file system that will

- Scale from a few networked computers to several thousand machines, supporting tens of thousands of clients.
- Be built from commodity, off-the-shelf components (PCs, ObSs) and a high-speed network, and run on existing operating systems such as Linux.

zFS extends the research done in the DSF project [7] by using object disks as storage media and by using leases and distributed transactions.

[*]IBM Labs, Haifa University, Mount Carmel, Haifa 31905, Israel. E-mail: {orodeh|teperman}@il.ibm.com.

[†]**Corresponding Author**

The two most prominent features of zFS are its *cooperative cache* and *distributed transactions*. zFS integrates the memory of all participating machines into one coherent cache. Thus, instead of going to the disk for a block of data already in one of the machine memories, zFS retrieves the data block from the remote machine. zFS also uses distributed transactions and leases, instead of group-communication and clustering software. We intend to test and show the effectiveness of these two features in our prototype.

zFS has six components: a *Front End* (FE), a *Cooperative Cache* (Cache), a *File Manager* (FMGR), a *Lease Manager* (LMGR), a *Transaction Server* (TSVR), and an *Object Store* (ObS). These components work together to provide applications/users with a distributed file system.

The design of zFS addresses, and is influenced by, issues of fault tolerance, security and backup/mirroring. However, in this article, we focus on the zFS high-level architecture and briefly describe zFS's fault tolerance characteristics. The first prototype of zFS is under development and will be described in another document.

The rest of the article is organized as follows: In Section 2, we describe the goals of zFS. Section 3 details the functionality of the various components and how they interact with each other. Issues of fault tolerance are briefly discussed in Section 4. We conclude with a summary of how combining all these components supports higher performance and scalability.

## 2 zFS Goals

The design and implementation of zFS is aimed at achieving a scalable file system beyond those that exist today. More specifically, the objectives of zFS are:

- A file system that operates equally well on only on few or on thousands of machines
- Built from off-the-shelf components with ObSs
- Makes use of the memory of all participating machines as a global cache to increase performance

- The addition of machines will lead to an almost linear increase in performance

zFS will achieve scalability by separating storage management from file management and by dynamically distributing file management.

Storage management in zFS is encapsulated in the *Object Store Devices (*ObS*s)*[1], while file management is done by other zFS components, as described in the following sections.

Having ObSs handle storage management implies that functions usually handled by file systems are done in the ObS itself, and are transparent to other components of zFS. These include: data striping, mirroring, and continuous copy/PPRC, which are done at the ObS level.

The Object Store recognizes only those objects that are sparse streams of bytes. Thus, it does not distinguish between files and directories. It is the responsibility of the file system management (the other components of zFS) to handle them correctly.

zFS is designed to work with a relatively loosely-coupled set of components. This allows us to eliminate clustering software, and take a different path than those used by other clustered file systems [9, 5, 1]. zFS is designed to support a low-to-medium degree of file and directory sharing. However, we do not claim to reach GPFS-like scalability for very high sharing situations.

## 3 zFS Components

This section describes the functionality of each zFS component, and how it interacts with other components.

### 3.1 Object Disk

The object disk (ObS) is the storage device on which files and directories are created, and from where they are retrieved. The ObS API enables creation and deletion of objects (files), and writing and reading byte-ranges from the object. Object disks provide file abstractions, security, safe writes and other capabilities as described in [6].

Using object disks allows zFS to focus on management and scalability issues, while letting the ObS handle the physical disk chores of block allocation and mapping.

[1]We also use the term *Object Disk*.

### 3.2 Front End

The zFS front-end (FE) runs on every workstation on which a client wants to use zFS. It presents the client with the standard POSIX file system API and provides access to zFS files and directories. On any Unix system (which includes Linux), this implies integration with the VFS layer, which implies that the FE must be an in-kernel component.

### 3.3 Lease Manager

The need for a *Lease Manager* (LMGR) stems from the following facts: (1) File systems use one form or another of locking mechanism to control access to the disks in order to maintain data integrity when several users work on the same files. (2) To work in SAN file systems where clients can write directly to object disks, the ObSs themselves have to support some form of locking. Otherwise, two clients could damage each other's data.

In distributed environments, where network connections and even machines themselves can fail, it is preferable to use *lease*s rather than locks. Leases are locks with an expiration period that is set up in advance. Thus, when a machine holding a lease on a resource fails, we are able to acquire a new lease after the lease of the failed machine expires. Obviously, the use of leases incurs the overhead of lease renewal on the client that acquired the lease and still needs the resource.

To reduce the overhead of the ObS, the following mechanism is used: each ObS maintains one *major lease* for the whole disk. Each ObS also has one lease manager (LMGR) which acquires and renews the major lease. Leases for specific objects (files or directories) on the ObS are managed by the ObS's LMGR. Thus, the majority of lease management overhead is offloaded from the ObS, while still maintaining the ability to protect data.

The ObS stores in memory the network address of the current holder of the major-lease. To find out which machine is currently managing a particular ObS $O$, a client simply asks $O$ for the network address of its current LMGR.

The lease-manager, after acquiring the major-lease, grants exclusive leases on objects residing on the ObS. It also maintains in memory the current network address of each object-lease owner. This allows looking up file-managers.

Any machine that needs to access an object *obj* on ObS $O$, first figures out who is it's LMGR. If one exists, the object-lease for *obj* is requested form from the

LMGR. If one does not exist, the requesting machine creates a local instance of an LMGR to manage $O$ for it.

### 3.4 File Manager

Each opened file in zFS is managed by a single file manager assigned to the file when the file is opened. The set of all currently active file managers manage all opened zFS files. Initially, no file has an associated file-manager(FMGR). The first machine to perform an `open()` on file $F$ will create an instance of a file manager for $F$. Henceforth, and until that file manager is shut-down, each lease request for any part of the file will be mediated by that FMGR. For better performance, the first machine which performs an `open()` on a file, will create a *local* instance of the file manager for that file.

The FMGR keeps track of each accomplished `open()` and `read()` request, and maintains the information regarding where each file's blocks reside in internal data structures. When an `open()` request arrives at the file manager, it checks whether the file has already been opened by another client (on another machine). If not, the FMGR acquires the proper exclusive lease from the lease-manager and directs the request to the object disk. In case the data requested resides in the cache of another machine, the FMGR directs the Cache on that machine to forward the data to the requesting Cache.

The file manager interacts with the lease manager of the ObS where the file resides to obtain an exclusive lease on the file. It also creates and keeps track of all range-leases it distributes. These leases are kept in internal FMGR tables, and are used to control and provide proper access to files by various clients. For more details on the lease manager, see Section 3.3.

### 3.5 Cooperative Cache

The *cooperative cache* (Cache) of zFS is a key component in achieving high scalability. Due to the fast increase in network speed nowadays, it takes less time to retrieve data from another machine's memory than from a local disk. This is where a cooperative cache is useful. When a client on machine $A$ requests a block of data via $\mathsf{FE}_a$ and the file manager ($\mathsf{FMGR}_B$ on machine $B$) realizes that the requested block resides in the Cache of machine $M$, $\mathsf{Cache}_m$, it sends a message to $\mathsf{Cache}_m$ to send the block to $\mathsf{Cache}_a$ and updates the information on the location of that block in $\mathsf{FMGR}_B$. The Cache on $A$ then receives the block, updates its internal tables (for future accesses to the block) and passes the data to the $\mathsf{FE}_a$, which passes it to the client.

Needless to say, leases are checked/revoked/created by the FMGR to ensure proper use of the data.

### 3.6 Transaction Server

In zFS, directory operations are implemented as distributed transactions. For example, a create-file operation includes, at the very least, (a) creating a new entry in the parent directory, and (b) creating a new file object. Each of these operations can fail independently, and the initiating host can fail as well. Such occurrences can corrupt the file-system. Hence, each directory operation should be protected inside a transaction, such that in the event of failure, the consistency of the file-system can be restored. This means either rolling the transaction forward or backward.

The most complicated directory operation is `rename()`. This requires, at the very least, (a) locking the source directory, target directory, and file (to be moved), (b) creating a new directory entry at the target, (c) erasing the old entry, and (d) releasing the locks.

Since such transactions are complex, zFS uses a special component to manage them: a *transaction server* (TSVR). The TSVR works on a per operation basis. It acquires all required leases and performs the transaction. The TSVR attempts to hold onto acquired leases for as long as possible and releases them only for the benefit of other hosts.

### 3.7 File Creation Scenario

In this section, we describe a file creation scenario and explain how zFS components interact. This algorithm is shown in Figure 1.

**File creation - `open(path/file,...)`** We assume the normal Unix semantics that `create()` is actually `open(...,O_CREAT)`; 9i.e. open with create flag). We also use the term *file system pointer*, fsptr, to denote a pair $<$*obs_id*,*oid*$>$, which uniquely identifies a file (object).

1. FE receives a request to create `path/fname`. It does a lookup, which starts at a pre-configured location, this is the root directory[2], which is situated in a known location. The TSVR is consulted for missing parts of the path. Finally, FE will hold an fsptr for

[2]This is the only part of the file system that is located in a location that do not change.

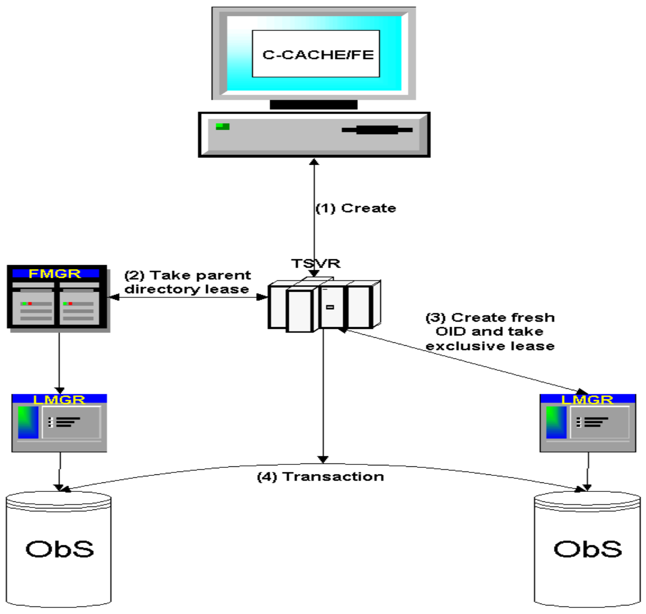
C-CACHE/FE
(1) Create
TSVR
FMGR
(2) Take parent directory lease
(3) Create fresh OID and take exclusive lease
LMGR
LMGR
(4) Transaction
ObS
ObS

Figure 1: zFS Operations - walk through `open(path/file,...)`. (1) The host sends a request to create a file. (2) The TSVR takes a lease for the parent directory from the FMGR (3) The TSVR creates a fresh object-id for the file, and takes an exclusive lease for it from the LMGR (4) The TSVR performs a transaction to create the file-object and create a new entry for it in the parent directory

the directory where the file is to be created, `path`: $path_{ptr}$.

2. FE sends a request to TSVR to create a file named `fname` in directory $path_{ptr}$.

3. The TSVR:

   (a) Takes all appropriate leases

   (b) Decides on an ObS in which to create the file

   (c) Creates a new (never used before) object_id (*oid*) for it

   (d) Performs the transaction

4. The TSVR returns the fsptr to the new file.

Note that no Cache is involved during the file creation/opening, since the Cache only gets involved when data is actually read by the client.

## 4 Handling Failures

In this section, we describe several failure scenarios which can occur during file system operation. The zFS error-recovery algorithms sketched here enable the system to recover and return to normal operating state.

If an FE fails, when it is connected to some FMGR opening file $X$, then the FE leases will expire after a timeout, and the FMGR will be able to recover control of portions of the file held by the failed FE. The same goes for any directories the FE held open.

Failure of an FMGR managing file $X$ is detected by all the FEs that held $X$ open, as well as the LMGR that managed the ObS on which $X$ was located. Once FE's range-leases expire, and cannot be refreshed, all dirty file blocks are flushed to the ObS and all file blocks are discarded. $X$ is mapped to some object $X_{obj}$ on ObS $obs_i$. Since the FMGR, does not renew its object lease on $X_{obj}$, it is then automatically recovered after a timeout by LMGR$_i$. Clients instantiate a new FMGR and once the lease for $X_{obj}$ expires, the new file-manager takes over.

Failure of an LMGR$_i$ is detected by FMGRs that hold leases for objects on ObS$_i$. Upon failure to renew a lease, FMGR informs all FEs that received leases on the file, to flush all their data to disk and release the file. Subsequently, the client instantiates a new LMGR which attempts to take the ObS$_i$ lease. Once the old lease expires, this is possible, and operations on ObS$_i$ can continue.

An ObS failure is catastrophic, unless it is replicated, or, unless the file-system is intelligent enough to reconnect to it when it comes back up.

## 5 Summary

Building a file system from the components described above is expected to provide high performance and scalability due to the following features:

**Separation of storage from file management** Caching and metadata management (path resolution) are done on a machine that is different from the one storing the data – the object disk (ObS). Dynamic distribution of file and directory management across multiple machines is done when files and directories are opened. This offers superior performance and scalability, compared to traditional server-based file systems. For low-sharing scenarios, each file-manager will be located on the machine using that file.

This provides good locality. Because multiple machines can read and write to disks directly, the traditional centralized file-server bottleneck is removed. File system recovery can occur automatically; whenever a directory transaction fails, the next client to access the directory will fix it.

**Cooperative caching** The memories of the machines running the cooperative cache process are treated as one global cooperative cache. Clients are able to access blocks cached by other clients, thereby reducing ObS's load and reducing the cost of local cache misses.

**Lack of dedicated machines** Any machine in the system, including ones that run user applications, can run a file-manager and a lease-manager. Hence, machines can automatically get exclusive access to files and directories when they are the sole users. Furthermore, any machine in the system can assume the responsibilities of a failed component. This allows online recovery from directory system corruption (by failed transactions). The lease mechanism employed in zFS ensures that, in the event of failure, zFS will operate correctly. Thus, in zFS, there is no centralized manager and no centralized server which are a single point of failure.

**Use of Object Disks** The use of object disks greatly enhances the separation between management and storage activities. It relieves the file system from handling meta-data chores of allocating/removing and keeping track of disk blocks on the physical disk. Assuming discovery support will be added to ObSs, similar to what SCSI provides today, zFS clients will be able to discover online the addition of ObSs. Using load statistics, available from the ObS interface, will allow intelligent determination of file-placement.

All hard-state is stored on disk; hence, the rest of the file system can fail all at once without corrupting the file system layout on disk.

## 6 Comparison to Other File Systems

The full article will compare our work with Coda [3], Intermezzo [4], Lustre [1], xFS and XFS [2], StorageTank [5] and GPFS [9].

### Acknowledgements

We wish to thank Alain Azagury, Ealan Henis, Julian Satran, Kalman Meth, Liran Schour, Michael Factor and Zvi Dubitzky from IBM Research for their participation in useful discussions on zFS.

Thanks are due also to the DSF [7] team, Zvi Dubitzky, Israel Gold, Ealan Henis, Julian Satran, and Dafna Sheinwald on whose work zFS is based.

## References

[1] Lustre techincal project summary. Technical report, Cluster File Systems, Intel Labs, June 2001. `www.clusterfs.com`.

[2] T. Anderson, M. Dahlin, J. Neffe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File System. *ACM Transactions on Computer Systems*, February 1996.

[3] P. J. Braam. The coda distributed file system. June 1998.

[4] Braam, P.J., Callahan, M., and Schwan, P. The intermezzo file system. August 1999.

[5] Burns, Randal. *Data Management in a Distributed File System for Storage Area Networks.* PhD thesis, Department of Computer Science, University of California, Santa Cruz, 2000.

[6] Vladimir Dreizin, Noam Rinetzky, Ami Tavory, and Elena Yerushalmi. The Antara Object-Disk Design. Technical report, IBM Labs in Israel, Haifa University, Mount Carmel, 2001.

[7] Zvi Dubitzky, Israel Gold, Ealan Henis, Julian Satran, and Dafna Scheinwald. DSF - Data Sharing Facility. Technical report, IBM Labs in Israel, Haifa University, Mount Carmel, 2000. See also http://www.haifa.il.ibm.com/projects/systems/dsf.html.

[8] Edya Ladan, Ohad Rodeh, and Dan Tuitou. Lock Free File System (LFFS). Technical report, IBM Labs in Israel, Haifa University, Mount Carmel, 2002.

[9] Schmuck, Frank and Haskin, Roger. Gpfs: A shared-disk file system for large computing clusters. January 2002.