zFS - A Scalable Distributed File System Using Object Disks

Ohad Rodeh
orodeh@il.ibm.com

Avi Teperman
teperman@il.ibm.com

IBM Labs, Haifa University, Mount Carmel, Haifa 31905, Israel.

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 zFS high-level architecture and 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 (1) Scale from a few networked computers to several thousand machines, supporting tens of thousands of clients and (2) Be built from commodity, off-the-shelf components such as PCs, Object Store Devices (OSDs) and a high-speed network, and run on existing operating systems such as Linux.

zFS extends the research done in the DSF project [10] by using object disks as storage media and by using leases and distributed transactions.

The two most prominent features of zFS are its *cooperative cache* [8, 14] 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* (OSD). 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 zFS's various components followed by Section 4 which details zFS's architecture and protocols. Issues of fault tolerance are briefly discussed in Section 5 and Section 6 compares zFS to other file systems. We conclude with Section 7 summarizing 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:

- Creating a file system that operates equally well on few or thousands of machines
- Using off-the-shelf components with OSDs
- Making use of the memory of all participating machines as a global cache to increase performance
- Achieving almost linear scalability: 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 (*OSD*s)*[1] [1], while file management is done by other zFS components, as described in the following sections.

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

The Object Store 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 [12, 6, 2]. zFS is designed to support a low-to-medium degree of file and directory sharing. We do not claim to reach GPFS-like scalability for very high sharing situations [12].

3. zFS Components

This section describes the functionality of each zFS component, and how it interacts with other components. It also contains a description of the file system layout on the object store.

3.1. Object Store

The object store (OSD) is the storage device on which files and directories are created, and from where they are retrieved. The OSD API enables creation and deletion of objects, and writing and reading byte-ranges to/from the object. Object disks provide file abstractions, security, safe writes and other capabilities as described in [9].

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

3.2. File System Layout

zFS uses the object-stores to lay out both files and directories. We assume each directory maps to a single object, and that a file also maps to a single object[2]. A file-object contains the set of bytes that the file is comprised of. It may be sparse, containing many non-contiguous chunks. A directory contains a set of entries, where each entry contains: (a) a file name, (b) some flags, (c) a *file system pointer* fsptr that points to the location in the file-system where the file or directory resides. An fsptr is a pair *Object Store Identifier* and an *object id* inside that OSD: $\langle obs\_id, oid \rangle$.
An example is depicted in Figure 1.

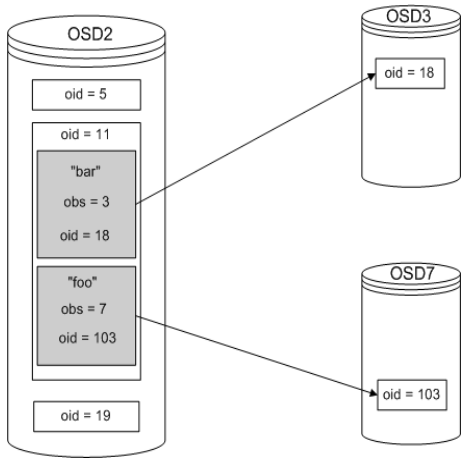

Figure 1. An example of a zFS layout on disk. There are three object stores: two, three, and seven. ObS2 contains two file-objects with object id's 5 and 19, it also contains a directory-object, number 11, that has two directory entries. These point to two files "bar" and "foo" that are located on OSD**s three and seven.**

Some file systems use different storage systems for meta-data (directories), and file data. Using Object-Stores for storing all data allows using higher level management and copy-services provided by the OSD. For example, an OSD will support snapshots, hence, creating a file-system snapshot requires taking snapshots *at the same time* from all the OSDs.

The downside is that directories become dispersed throughout the OSDs, and directory operations become distributed transactions.

3.3. Front End

The zFS front-end (FE) runs on every workstation on which a client wants to use zFS. It presents to the client the standard file system API and provides access to zFS files and directories. Using Linux as our implementation platform this implies integration with the VFS layer which also means the the FE is an in-kernel component. On many Unix systems (including Linux), a file-system has to define and implement three sets of operations.

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

[2] This can change in the future, to multiple objects per file.

Super Block Operations Operations that determine the behavior of the file system.

Inode Operations Operations on whole file and directory objects; e.g., `create, delete` etc.

File Operations Specific operations on files or directories; e.g., `open, read, readdir` etc.

By implementing these sets of operations and integrating them within the operating system kernel a new file-system can be created. In Linux this can be done either by changing the kernel sources or by building a loadable module implementing these operations. When the module is loaded it registers the new file system with the kernel and then the new file system can be mounted.

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

The OSD stores in memory the network address of the current holder of the major-lease. To find out which machine is currently managing a particular OSD 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 for objects residing on the OSD. 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 OSD O, first figures out who is it's LMGR. If one exists, the object-lease for obj is requested from the LMGR. If one does not exist, the requesting machine creates a local instance of an LMGR to manage O for it.

3.5. 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. This can be either the local Cache in case the FMGR is located on the client machine initiating the request, or a remote Cache otherwise.

The file manager interacts with the lease manager of the OSD 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 3.4.

3.6. 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 FE_a and the file manager (FMGR_B on machine B) realizes that the requested block resides in the Cache of machine M, Cache_m, it sends a message to Cache_m to send the block to Cache_a and updates the information on the location of that block in 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 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.

3.7. 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.

The FE sends all directory operations, asynchronous RPC style, to the TSVR and updates its internal dir-entries caches according to the results.

4. zFS Architecture

In this section we describe in detail how zFS components interact to present to the applications a file system. First we show how zFS components interconnect, following it with several protocols describing how file system operations are carried out in the zFS architecture.

4.1. zFS Component Interconnections

Figure 2 illustrates all the components of zFS. At the bottom we see several object disks and at the top we see two hosts running zFS components. The FE and Cache are situated inside the kernel while the LMGR, FMGR, and TSVR are located in a single process in user-space. It is important to emphasize that not all these components are active on all machines at all times. In the extreme case only the TSVR may be active and all other components for the files used on this particular machine run on other nodes. A socket connects the in-kernel and out-of-kernel components, and OSDs are accessed directly by the hosts.

To see the interactions between zFS components let us walk through several protocols: We start with the `read(), write()` operations following them with the `create()` and `rename()` file operations which require transactions.

4.2. Protocols

4.2.1. File Read Protocol Figure 3 shows the control and information paths for the `read(file, ...)` operation detailed below.

(a) FE looks up through the fsptr for *file*.

(b) If the read can be satisfied by locally cached file blocks (i.e., the data and read lease are locally cached) then the requested data is returned to the user and we `return`.

(c) A read request is sent to the FMGR of the file and the FE,Cache waits for the request to be satisfied.

(d) The FMGR checks and if necessary creates a read-lease for the requested blocks.

(e) The FMGR checks if other Caches hold the requested blocks of the file and does the following:

  (1) If TRUE

    (a) Forwards the above byte-range lease, the read request and the address of the requesting FE, the *requester*, to the Cache,FE on the host holding the requested blocks, *host*.

  (2) Else

    (a) Forwards the above byte-range lease, the read request and the address of the requesting FE, the *requester*, to the OSD holding the requested blocks.

(f) The FE/Cache on *host* or the OSD send the requested data blocks and the read lease to the *requester*.

We define the above scenarios as *third party communication*; i.e., the party who passed the actual data is not the same party from whom the data was requested. In Figure 3, in both cases the request is sent to the FMGR while the data arrives either from another Cache or from the OSD.

4.2.2. File Write Protocol Figure 4 shows the control and information paths for the `write(file, ...)` operation.

We assume that read operations were conducted by several users on the file and that some of its data blocks reside in several Caches. After some period of time a user wants to write to the file.

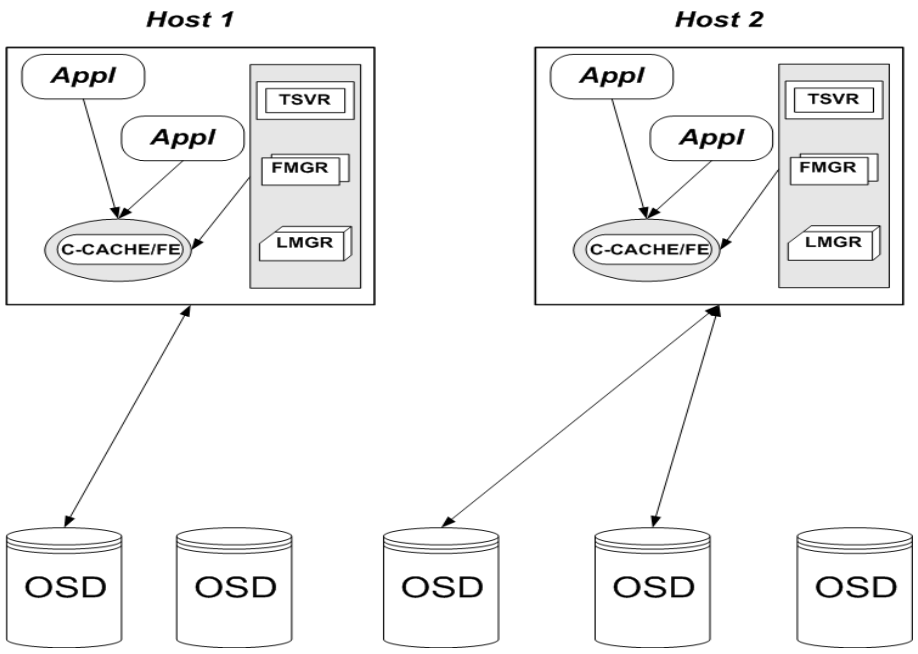

Figure 2. zFS Components

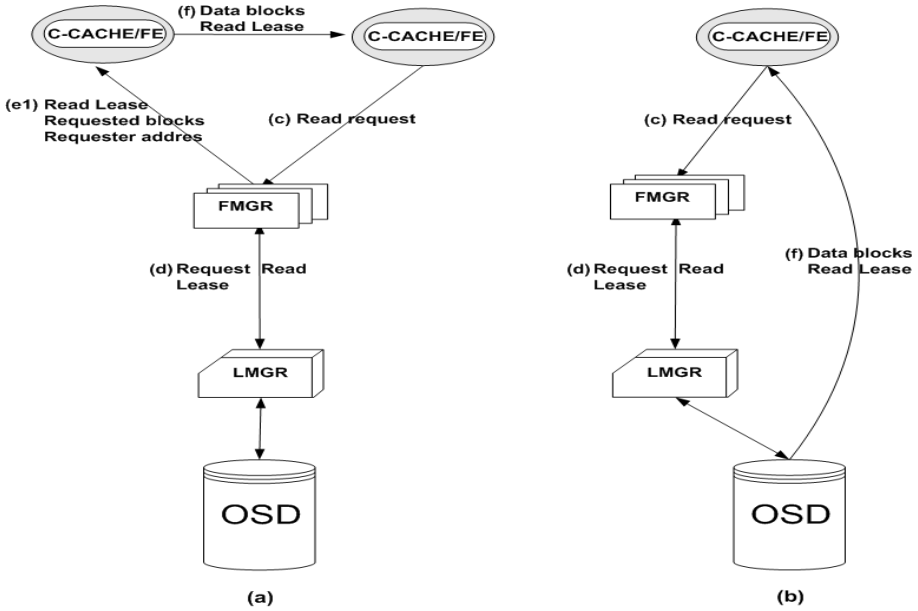

Figure 3. zFS Operations - walk through `read(file)`. (a) a cache-to-cache read (b) an OSD read

(a) FE sends the write request to $fmgr_i$

(b) $fmgr_i$ checks if other Caches hold blocks of the file and does the following:

(1) Revoke all read leases on the object (file) whose range overlaps with the write area. Messages are sent to the various Caches to that effect. Note that we *have to wait* for Acks from the Caches, otherwise some clients will read incorrect data.

(2) Revoke all overlapping `Write` leases. This may require flushing buffers to the object disk

(3) Create a write lease for the specified range.

(c) If data blocks are not (fully) in the local Cache and

the data is not aligned on page boundaries then read in proper pages into the cache

(d) Write the data to the proper blocks in the Cache.

(e) Return the proper result (number of bytes written or error code as the semantics of write requires) to the FE which passes it to the user.

4.2.3. Create File Protocol Upon receiving a `create(parent_dir, fname)` from the FE, the TSVR executes the following protocol shown also in Figure 5:

1. FE receives a request to create `fname` in directory `parent_dir`. It does a lookup, which starts at a pre-configured location, this is the root directory[3]. The TSVR is consulted for missing parts of the path. Finally, FE will hold an fsptr for the directory where the file is to be created, `paren_dir`.
2. FE sends a request to TSVR to create a file named `fname` in directory `parent_dir`.
3. TSVR chooses an object-store *obs_id* on which to create the new file
4. TSVR creates a new (unused before) object-id, *oid*, for the new file and acquires an exclusive lease for the new file (obs_id, oid)
5. TSVR acquires a lease for `parent_dir`
6. TSVR writes a new entry in the parent directory: { `name = fname ; flag = InCreate; fsptr =` (obs_id, oid) }
7. TSVR creates the file at (obs_id, oid)
8. TSVR writes initial meta-data to the file: gid, uid, etc.[4]
9. TSVR overwrites the dir-entry flag with `Normal` flag.
10. TSVR returns the fsptr of the new file to the FE.

Note that no Cache is involved during file creation, since the Cache only gets involved when data is actually read by the client. Also, simultaneous requests from two clients to open the same file are serialized by the file manager which manages the file.

To acquire the leases in stages 4 and 5 a two level hierarchy is used, see Figure 6.

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

[4]In our design the file's meta-data is attached to the file's data in a special block.

Each OSD has an associated LMGR that takes the major-lease, and manages exclusive object-leases on it. Each FMGR takes an exclusive lease for an object, and allows taking single-writer-multiple-reader leases on byte-ranges in the file. Each TSVR takes the leases it needs for directory transactions, and attempts to hold on to leases for as long as possible, assuming other hosts do not request them. Only *safe caching* is done at the FE and Cache. No data that is being modified on other hosts is locally cached, this provides strong cache consistency to file-system applications.

As part of its normal operation the TSVR has to locate LMGRs and FMGRs for OSDes, files, and directories. The input for this function is the OSD identifier, or the fsptr. To locate an LMGR for a particular OSD, the OSD itself is queried for the network address of the LMGR currently managing it. If one exists, it's address is returned. If one does not exists, the TSVR takes the OSD major-lease and creates a local LMGR for it. To locate an FMGR for a particular (obs_id, oid) (1) the LMGR for *obs_id* is located (2) it is queried for the FMGR of *oid*. If none exists, the TSVR creates a local FMGR to manager the object.

If all works out, then at the end of this transaction the file has been created. Failures can occur at each of these stages. The solution is to always roll-back. This is achieved by the following Erase sub-protocol:

1. Take the parent directory lease and the exclusive object lease for (obs_id, oid)
2. Overwrite the flag in the directory entry with an `InDelete` flag
3. Erase the object (obs_id, oid)
4. Erase the parent dir-entry
5. Release the leases taken

Choosing an object-store on which to create the file is an important issue. In order to achieve good load-balancing the creator needs to take into account the set of possible OSDs and choose the best candidate. We intend for each host to keep track of a set of OSDs it works with, and monitor their usage through periodic statistics gathering. It should fill up the same OSDs to provide locality, and should also refrain from filling up any particular OSD since this degrades its performance. This, admittedly simple, algorithm requires no global knowledge. This property is important in a large scalable system.

Choosing a fresh object-id needs to be carefully done. We use a special object *C* on each object-store *O* that holds an object-id counter. The counter is initiated when zFS formats the object-disk; it is set to an initial value of *I*. Objects with names *oid*s smaller than *I* are special zFS auxiliary objects. Objects with larger *oid*s are regular files and

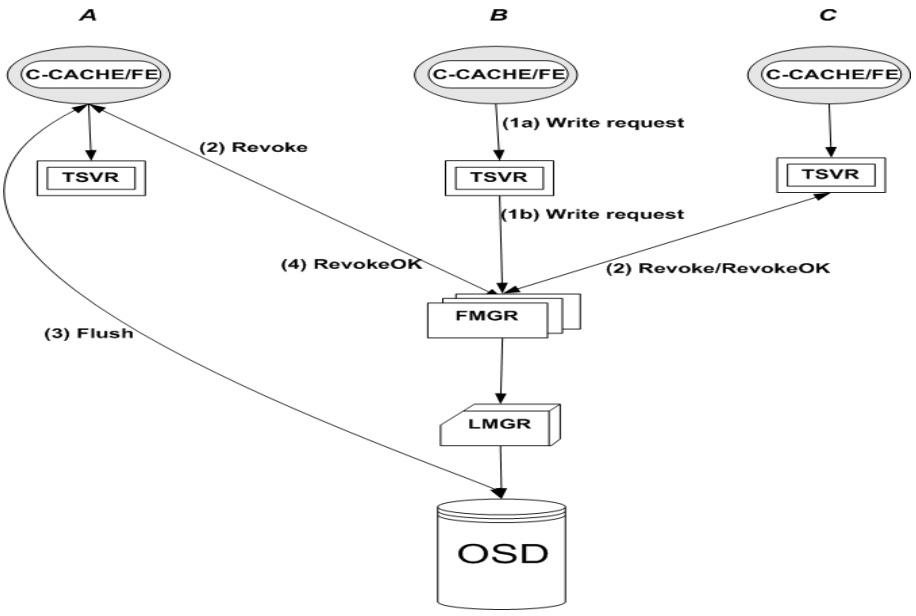

Figure 4. zFS Operations - walk through `write(file)`. Host *B* requests to write to a byte-range. The FMGR revokes existing read/write leases for the same range on other machines. Machine *A* has dirty data on the range, and needs to flush. Machine *B* can immediately return a Revoke_Ok. *B* then reads the relevant block(s) directly from the cache of *A*. No need to go to OSD.

Table 1. Example run of an interrupted create.

Host	operation
A	create dir-entry (fname, `InCreate`, (obs_id, oid))
A	`Create_object`$((obs\_id, oid))$
B	dir-entry flag := `InDelete`
B	`Delete_object`$((obs\_id, oid))$
B	erase dir-entry(fname,(obs_id, oid))

directories. The LMGR for *O* provides an interface for taking ranges of fresh object names. It takes the exclusive lease for *C*, reads the current value, increments it by *R*, and writes it to disk. This provides a range of *R* fresh names. Any application that needs fresh names can use this LMGR for that purpose.

This protocol may seem overly complex for a simple, basic operation, such as create. To see how this works, we shall walk through an example of failure scenario.

Assume host *A* starts a `create(`$(parent\_dir, fname)$ and fails after step 7. Host *B* comes along, attempts to read the dir-entry in *parent_dir* and sees the `InCreate` flag. It then initiates the Erase sub-protocol. The list of actions can be seen in Table 1.

We are careful to include in stage 6 of the create protocol the fsptr of the new file-object, although it has not been created yet. It is possible to first create the object, and then link it into the directory structure. However, if the host fails, a dangling object will remain on the OSD. Since there are no links to this object, it will never be erased. It is also possible to fail after the first phase of the protocol. Thus storing fsptr is important for the erase operations since it is required to finish the operation in case of failure.

Using fresh (never used before) object names is crucial. For example, assume that a create object operation uses object id *i*, and fails after stage 6. Later, a create operation reuses *i* and creates a file with id *i* in another directory. Even later, Erase is activated to erase the initial failed create-object, and erases *i*. This sequence of events will corrupt the file-system, with the second create dir-entry pointing to a non-existent object.

4.2.4. Rename Protocol `rename(src_dir, src_name, trg_dir, trg_name)` operation works as follows:

1. Take leases for the source and target directories
2. Check that the target entry has correct type: a file can only be copied onto a file, and a directory can only be copied onto a directory.
3. Overwrite the flag in the source dir-entry to `RenameFrom`(trg_dir, trg_name)

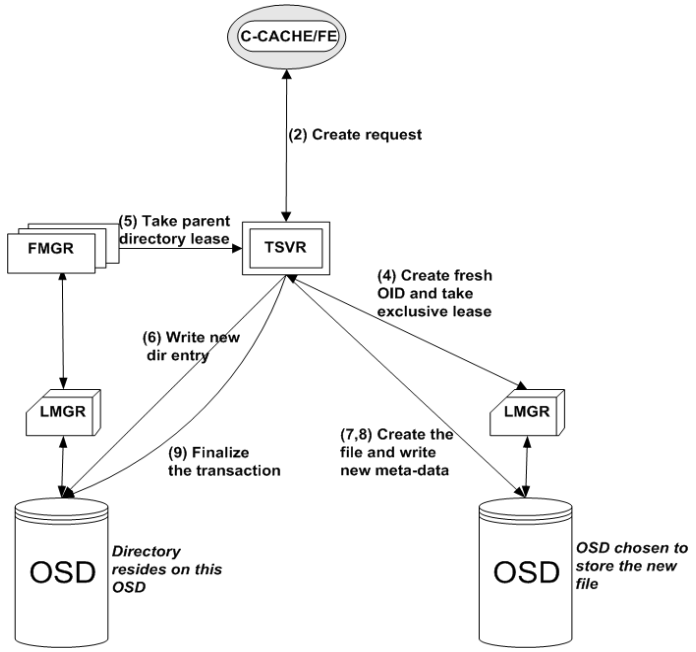

Figure 5. zFS Operations - walk through `create(parent_dir/fname,...)`**. (2) The host sends a request to create a file. (4) The** TSVR **creates a fresh object-id for the file, and takes an exclusive lease for it from the** LMGR **(5) The** TSVR **takes a lease for the parent directory from the** FMGR **(6) The** TSVR **write a new temporary directory entry on the object disk holding the parent directory (7,8) The** TSVR **creates the file and writes the initial meta-data (9) The** TSVR **finalizes the transaction.**

4. Create the target entry if it does not exist. Overwrite the flag to `RenameTo`(src_dir, src_name)
5. Overwrite the source dir-entry flag to `RenameFromSucc`(trg_dir, trg_name)
6. Overwrite the target dir-entry flag to `Normal`
7. Erase the old entry (including the log)
8. Release the leases taken

The no-return stage is step 5. If the initiating host fails prior to that phase, the rename operation needs to be rolled back. If that step is passed, any other host must help roll it forward.
To roll the rename back we do:

1. Take leases for the source and target directories
2. Overwrite the source dir-entry flag with `Normal`
3. Erase the target dir-entry
4. Release the leases taken

To support complete roll-back, we must log into the dir-entries participating in the operation complete information on the rename source and target. The largest log component is the file name which can be any sequence of 1 to 255 bytes.

An example of an interrupted rename operation where the initiating host fails before reaching the no-return stage is depicted in Table 2. Host *A* performs stages one and two and fails. Host *B* stumbles onto the source dir-entry, reads that there is an in-flight rename operation, and determines that it should be rolled back. It then converts the source dir-entry to `Normal` and erases the half-baked target entry.

5. 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 OSD on which X was located. To ensure that in

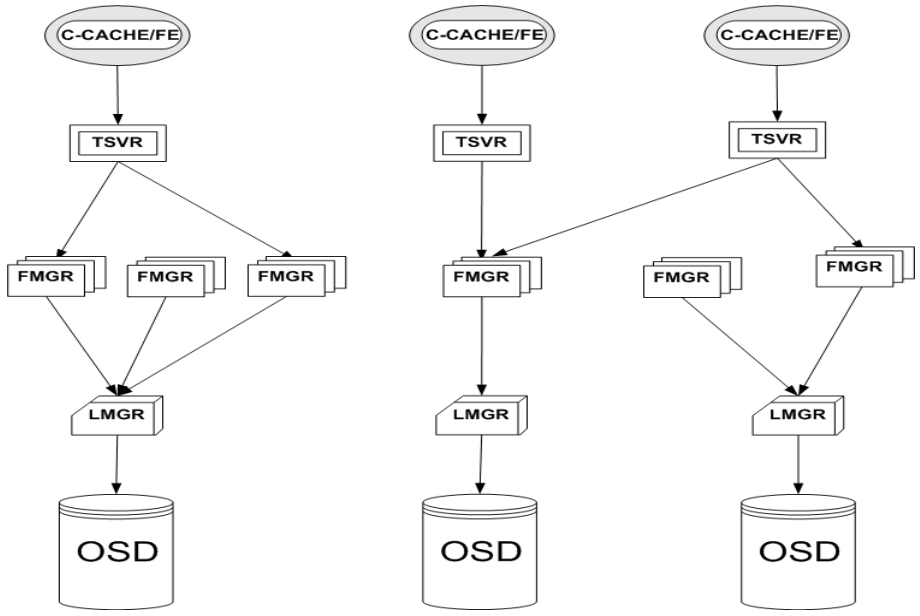

Figure 6. The hierarchy of leases

Table 2. Example run of an interrupted rename.

Host	operation on source	operation on target
A	flag := `RenameFrom`(*trg_dir*,*trg_name*)	
A		flag := `RenameTo`(*src_dir*,*src_name*)
B	flag := `Normal`	
B		erase dir entry

such case all dirty pages are saved on the OSD, zFS uses the following mechanism. zFS limits the number of dirty pages in each host's Cache to N, where N is the number of pages which can safely be written to OSD in time T. Assuming lease renewal time is set to *Renew_Time* ($Renew\_Time >> T$), if a lease renewal does not arrive in $Renew\_Time - T$ after the last renewal, then all dirty pages are written to the OSD.

Once FE's range-leases expire and cannot be refreshed, all file blocks are discarded. Suppose X is mapped to some object X_{obj} on OSD 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 OSD_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 OSD_i lease. Once the old lease expires, this is possible, and operations on OSD_i can continue.

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

6. Comparison to other File Systems

There are many file-system to compare to and we cannot include here a complete list. Therefore, we focus here only on few examples. We observed that none of these file systems use cooperative caching.

6.1. Coda

Coda [4] is a file-system designed for disconnected operation. It separates the file-system into servers that maintain FS volumes, and remote clients that connect to the system. The clients may work disconnected from the system, and wish to reconnect and synchronize their local work with FS state. To allow efficient disconnected operation, Coda relaxes consistency requirements. It optimistically assumes that clients work on their own separate home-directories, and that there is very little sharing. Hence, a client may take its files, disconnect, make changes, and after a few hours reconnect and send its new file versions

back to the servers. This works well up to a point. If there is file or directory sharing, relaxed consistency will result in inconsistency. Each client will take a directory or file home, and make incompatible changes. To work around this, the file system has a mechanism to detect inconsistency upon client reconnection, and either merges changes together, or asks the client to manually merge incompatible files.

In contrast, zFS does support disconnected operation, and enforces strong cache consistency. The challenge is to make such a system scale and work efficiently.

6.2. Intermezzo

Intermezzo [5] is a recent rewrite of Coda so that its local operations will be on par with local file-systems such as ext2 [7].

In zFS, we would like to achieve good performance for local operations, when we are working against a locally cached file or directory; i.e., to be as fast as local file-systems as well.

6.3. Lustre

Lustre [2] is a SAN file-system[5] that is built out of three components: *clients*, *Cluster control-system*, and *Storage Targets* where a SAN connects them together.

The *clients* see a cluster file-system with standard POSIX semantics.

Cluster Control Systems manage name-space and file system meta-data coherence, security, cluster recovery, and coordinate storage management functions. Cluster control systems do not handle file data and direct clients to perform file I/O directly and securely with storage targets.

Storage Targets store persistent data and participate in management functions. Targets are OSDs and are furthermore, programmable, allowing the execution of downloaded modules.

Lustre is being constructed, and is an open-source project.

zFS does not assume that OSDs are programmable. However, it does assume a non-standard locking interface provided by the OSDs. Lustre uses consensus-style methods to achieve file-system coherency in caching and locking, whereas zFS uses OSD based leases.

6.4. xFS and XFS

xFS [3] is a network file system which attempts to distribute all aspects of file operations over multiple machines connected by a network. The goal of xFS is similar to zFS, achieving high availability, performance and scalability. However, xFS management distribution policy is static while zFS uses dynamic distribution which is sensitive to network load.

XFS [13] is a *local* file-system built by SGI, that scales to large file-systems, and is reputed to provide good scalability and performance for local and SMP systems.

zFS is designed to achieve these goals, but in a more demanding distributed settings. We do not expect to achieve XFS's excellent local performance in the near future.

6.5. StorageTank

StorageTank [6], much like Lustre, is a SAN file-system built in IBM, where there are *clients*, *meta-data servers*, and OSDs. StorageTank currently works with standard SCSI disks over a SAN. New design is underway to enable it to use object-stores. For a fair comparison we assume a future system that works with OSDs.

StorageTank clients and OSDs are connected directly by a SAN, allowing the efficient movement of bulk data. Meta-data servers are connected to the rest of the system through a different, IP network. Meta-data servers maintain all file-system meta-data state, and are further responsible for all coherency, and management issues. Meta-data servers may be clustered for better scalability.

zFS does not use clustering technologies and does not rely on different SAN/IP networks. The disadvantage is the added overhead of directory operations being distributed transactions instead of local operations on a meta-data server.

6.6. GPFS

GPFS [12] is a file-system built by IBM for high-end super-computers. The largest installation we know of is comprised of 512 compute nodes, and 1024 disks. The system can handle traditional work loads, with little sharing, however, it is designed to handle well large scientific applications which use large files with heavy write-sharing.

In contrast with zFS, GPFS is a fully developed file-system, with years of experience and with some of the largest installations in existence. However, whereas GPFS uses standard consensus style solutions to address failures, zFS attempts a different OSD based scheme. Furthermore, GPFS uses standard disks, whereas zFS uses OSDs. We expect comparative performance for low to medium sharing situations.

6.7. StorageNet

StorageNet is a file-system under development in IBM research. It shares some features with zFS: use of object-based storage, and high scalability requirements. It uses no

[5] we use the term "SAN file-system" to denote a file system which uses SAN as its block device server.

file-servers, the file-system is comprised solely of clients and object-stores. While the file-systems are similar in some respects, their goals are very different.

StorageNet is focused on high-scalability, WAN operation, and security; while zFS targets strong cache-consistency, and safe distributed transactions.

7. 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 (OSD). 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 OSD'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 OSDs, similar to what SCSI provides today, zFS clients will be able to discover online the addition of OSDs. Using load statistics, available from the OSD 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.

Acknowledgements

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

Thanks are due also to the DSF [10] team, Zvi Dubitzky, Israel Gold, Ealan Henis, Julian Satran, and Dafna Sheinwald on whose work zFS is based.

References

[1] SCSI Object-Based Storage Device Commands (OSD). Working Draft - Revision 6. See ftp://ftp.t10.org/t10/drafts/osd/osd-r06.pdf.

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

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

[4] P. J. Braam. The Coda Distributed File System. June 1998.

[5] Braam, P.J., Callahan, M., and Schwan, P. The InterMezzo File System. August 1999.

[6] 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.

[7] Card, R., Ts'o, T., and Tweedie, S. The Design and Implementation of the Second Extended Filesystem. In *Dutch International Symposium on Linux*, December 1994.

[8] M. D. Dahlin, R. Y. Wang, and T. E. A. D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *OSDI*, 1994.

[9] V. Dreizin, N. Rinetzky, A. Tavory, and E. Yerushalmi. The Antara Object-Disk Design. Technical report, IBM Labs in Israel, Haifa University, Mount Carmel, 2001.

[10] Z. Dubitzky, I. Gold, E. Henis, J. Satran, and D. 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.

[11] E. Ladan, O. Rodeh, and D. Tuitou. Lock Free File System (LFFS). Technical report, IBM Labs in Israel, Haifa University, Mount Carmel, 2002.

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

[13] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, 1996.

[14] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, and H. M. Levy. Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System. Technical report, Department of Computer Science and Engineering, University of Washington.