RAMCloud Filesystem Design

Diego Ongaro

2010-01-05

1 Purpose

To exercise RAMCloud's API by creating a robust networked filesystem on top of RAMCloud.

2 Requirements

- POSIX-like filesystem semantics
- Highly concurrent
- Each object is self-identifying
- No off-line fsck - the filesystem must be immediately available after a crash and must not degrade in performance over time
- Atomic filesystem operations

3 Challenges

Concurrency and atomicity are difficult to achieve with no primitives that cover multiple objects or operations. In the filesystem, most of the concurrency problems are largely solved by executing multiple operations in a specific order, but dangling objects and links are difficult to prune after a client crash.

4 Table-Level Design

Each filesystem occupies exactly two tables: one for inodes and one for clients. While newer versions of an object have the same OID as previous versions, OIDs are never reused to refer to two entirely different objects.

4.1 File Objects

Each non-directory inode is represented as a file object. It contains the following:

- stat data
- an opaque blob
- references to its linking directories
- a list of client OIDs which have the file open

4.2 Directory Objects

Each directory inode is represented as a directory object. It contains the following:

- stat data
- a mapping from link names to OIDs linked, each having a distinct, optional write-lease

  The write-lease is a UNIX timestamp paired with an intent - either to add the link or to remove the link.
- references to its linking directories

4.3 Client Objects

Each client connection has an object. It contains the following:

- a keep-alive-till timestamp

  This field is indexed so that expired clients can be found quickly.
- A list of file OIDs which may be open by the client.

If the connection is active, the keep-alive-till timestamp is in the future. When a polite client disconnects, it removes its client references from any open file objects and removes its client object. Moreover, when a polite client connects or disconnects, it will query the keep-alive-till index and clean any expired clients.

5 Invariants

To maintain atomicity even across concurrent connections, the following invariants are imposed on the objects:

- An inode object is always reachable by some path or open by some client (except for the root directory).
- If a file has a reference to a linking directory, that linking directory exists and has a directory entry referring to the file.
- If a directory has a directory entry without a write lease, it refers to a file that also has a reference back to the directory.
- If a file is open by some client, that client object has the file's OID in its list of possibly open files.

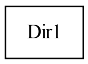
6 Complex Operations

There are five filesystem operations which make use of multiple objects, making concurrency and atomicity non-trivial: *mknod*, *link*, *unlink*, *rename*, and *readdir*.

Mutually exclusive write-leases make any combination of these operations safe to execute concurrently. Synchronized clocks are assumed.

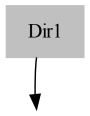
6.1 *mknod*

1. Find the directory in which we will create the node (Dir1).

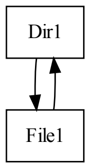


2. Reserve an OID for File1. Add a directory entry for File1 to Dir1, holding a link write-lease on it for a couple seconds.

   If the name already exists in Dir1 and has no write-lease, return EEXIST. If the name exists but has an active write-lease, retry. If the name exists but has a stale write-lease, clean it and retry.



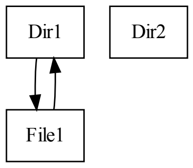
3. Create the node File1, with File1 referencing Dir1 as a parent.



4. Drop the link write-lease added in the second step from Dir1.

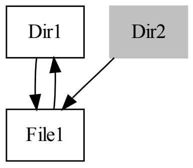
6.2 *link*

1. Find the target of the link through the given path (File1 through Dir1), and find the directory in which we will create the new name (Dir2).

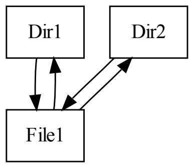


2. Add a directory entry for File1 to Dir2, holding a link write-lease on it for a couple seconds.

   If the name already exists in Dir1 and has no write-lease, return EEXIST. If the name exists but has an active write-lease, retry. If the name exists but has a stale write-lease, clean it and retry.



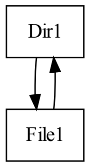
3. Change File1 to reference Dir2 as another parent.



4. Drop the link write-lease added in the second step from Dir2.

6.3 *unlink/rmdir*

1. Find the file and its containing directory through the given path (File1 through Dir1).



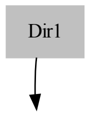
2. Take an unlink write-lease for a couple seconds on the directory entry in Dir1.

   If the directory entry is gone, return ENOENT. If it has an active write-lease, retry. If it has a stale write-lease, clean it and retry.

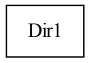
3. Drop reference to Dir1 from File1.

   If File1 has no more references to any directories, as we assume in the diagram below, just remove it entirely.

   If File1 is in fact a directory (*rmdir* operation), ensure it has no directory entries.

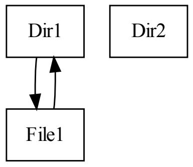


4. Remove the directory entry for File1 from Dir1, dropping the unlink write-lease added in step 2.



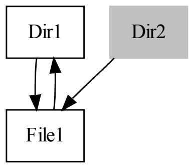
6.4 *rename*

1. Find the target of the rename through the given path (File1 through Dir1), and find the directory in which we will create the new name (Dir2).



2. Add a directory entry for File1 to Dir2, holding a link write-lease on it for a couple seconds.

   If the name already exists in Dir1 and has no write-lease, return EEXIST. If the name exists but has an active write-lease, retry. If the name exists but has a stale write-lease, clean it and retry.

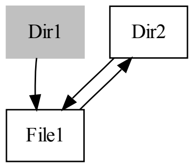


3. Take an unlink write-lease for a couple seconds on the directory entry in Dir1.

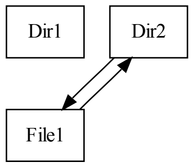
   If the directory entry is gone, revert the previous step and return ENOENT. If it has an active write-lease, retry. If it has a stale write-lease, clean it and retry.

   (This step and the previous can be done in either order. Choose the directory with the smaller OID first to avoid deadlock.)

4. Change File1 to reference Dir2 as its parent.



5. Remove the directory entry for File1 from Dir1, dropping the unlink write-lease added in the third step.



6. Drop the link write-lease added in the second step from Dir2.

   (This step and the previous step can be done in either order.)

6.5 *readdir*

readdir doesn't wait on leases, since that could get expensive in a large directory. Any links that are being created (link write-leases) should be treated as non-existent. Any links that are being removed (unlink write-leases) should be treated as existent. POSIX allows *readdir* implementations plenty of freedom to choose whether to include files being created and removed, and this is the safe thing to do with clients possibly crashing.

Any expired leases should also be cleaned.

7 Maintenance

7.1 Inode Table

Several operations above have the client clean a directory entry with an expired write-lease. Moreover, all path traversals also clean any expired write-leases they encounter. The procedure to do so is as follows:

1. Obtain a new write-lease on the directory entry (with the same intent as was previously there).

2. Read the inode pointed by the directory entry.

3. If it does not exist or does not have a reference back to the linking directory, delete the directory entry. If it does exist and has a reference back to the linking directory, remove the write-lease.

(In order to purge all expired write-leases, one can run the UNIX *find* utility on the root of the filesystem. This results in a *readdir* call for every directory, which will in turn clean any stale directory entry write-leases. This is probably a waste of effort, though.)

7.2 Clients Table

Stale clients will be cleaned every time a client connects or disconnects. The procedure to do so is as follows:

1. For each file OID which may be open by the client, remove any reference to the client OID from the file, deleting the file if necessary.

2. Delete the client object.

(In order to purge all expired clients and open files they refer to, one can simply connect and disconnect. Again, this is probably a waste of effort.)

8 OID Reservation

The *mknod* operation assumes that it is possible to reserve an OID for a file without utilizing any additional space in the table. This client-side OID allocation is implemented in a separate module.

An object at a well-known location keeps the next unreserved OID, starting at 0. Each client requests a few OIDs at a time by bumping up this number. It is then guaranteed that no other cooperating client will use these OIDs.

If a client crashes, at most a few OIDs are wasted and will never be reclaimed. Since the range of OIDs is so large, this should be acceptable for most applications.