Changelogs and Feeds

Nathan Rutman

1/30/08

1 Introduction

A *changelog* is a log of data or metadata changes. In general, these will track filesystem operations conveyed via one or more RPCs. Changelogs are used by *consumers* such as userspace audit logs, mirroring OSTs or files, database feeds, etc. Changelogs are stored persistently and transactionally and are removed upon completion.

There are 3 subflavors of changelogs (we intend to use the same changelog facility for all).

1. rollback (undo) logs - used for filesystem recovery

2. replication logs - used to propagate changes from a master server to a replica

3. audit logs - record auditable actions (file create, access violation, etc.)

Audit logs are presented to userspace consumers via a special transactional, readable file called a *feed*. A replication log may also be presented as a feed, if a userspace consumer is to be used. Rollback logs are only used internally.

2 Requirements

Implement changelogs as a low level service for the following use cases.

2.1 Audit logs (text)

A particular site policy requires audit logs of filesystem usage (access, create, delete, write, etc) and errors (specifically permission failures, perhaps quota overrun). Files A and B are on MDT0001, where the sysadmin has set up a Lustre audit feed with mask ALL_FILES|ALL_EVENTS. User on client 1 opens file A, reads file A, removes file A. User on client 2 attempts to open file B, but fails with permission denied. The audit feed, presented as a file under /mnt/MDT0001/.lustre/audit, is updated synchronously with event records A/open/ok, A/read/ok, A/delete/ok, B/open/EACCESS. The feed is read and piped into a regular file in /var/logs/audit where it is periodically purged by a logging daemon.

The requirements of this scenario are:

- audit log, including not only filesystem changes but also access events, atime changes, request failures.
- feed based on audit log
- feed set up (filter, retention policy)

2.2 Database

An external database is to be updated with filesystem changes for customer-specific purposes (audit, query, HSM, etc.). An audit feed is set up on each server; the feed consumer sends entries to the database back-end. The filesystem events are integrated into database, even in the event of power loss and recovery.

Entries are removed from the audit feed only after the feed consumer has indicated completion (database integration) of the entry.

Cross-server synchronization required is not required; if a single event results in two changelog entries on two servers, these need not be reconciled/recombined before submission to the consumer. However, a common identifier will indicate linked entries. For example, renaming a file from MDT0001 to MDT0002 will result in a changelog entry on each server; these will share a UUID so that the consumer (database) can act appropriately.

The requirements of this scenario are:

- audit log, including not only filesystem changes but also access events, atime changes
- feed based on audit log
- feed set up (filter, retention policy)
- shared cookie for compound transactions
- full recovery semantics on feed

2.3 Replication

A filesystem replica is defined to actively track changes in the master filesystem, to provide widely-distributed access to files and provide redundancy in case of catastrophic failure at one site (Continuity Of OPerations). The replica has a different layout than the master filesystem. Changes in data or metadata on the master produce changelog entries; these entries are re-executed on the replica to bring the replica up-to-date. The replica must remain internally consistent at all times. The replica must remain up-to-date within some time frame (e.g. new files on master are copied to replica within 5 minutes). The replica will always remain consistent with the master in the event of a cache miss (e.g. file 1 on master is modified, replica will block until extents have been copied to replica if a client on the replica tries to read it).

The requirements of this scenario are:

- Replication log, including only filesystem changes (not audit-type events, e.g. permission failures).
- Objects and byte ranges of modified data must be recorded in the OST changelogs; reintegration of the changes on the replica will result in I/O to the master requesting this data.
- Metadata changes should be completely described in the changelog so that no additional RPCs are needed during reintegration.
- Replica must remain up-to-date within a reasonable (user-definable?) time frame. Optimization: we need only propagate "final" versions of modified objects for objects undergoing rapid rewrites. (But the time limit still applies to prevent major data loss for COOP scenarios.)
- Replica must remain consistent with master on specific file access.
- Replica must always remain internally consistent (master dies, replica must roll back to an epoch boundary).
- Replica must be fully scalable (e.g. 5 widely-distributed replicas of a large master system must not significantly slow Lustre).

2.4 Reintegration

File or metadata is changed on caching (flash cache or proxy) server. The caching servers may have a different layout than the master servers. Client-driven changes in data or metadata on the cache produce replication changelog entries on the caching servers. When the cache is flushed, the batched changelogs are sent to the master servers to be reintegrated. Multiple changes may be merged in the cache; only the net effects need to be sent and/or reintegrated.

The requirements of this scenario are:

- Replication log, including only filesystem changes (not audit-type events, e.g. permission failures).
- The objects (or inodes) of modified data must be recorded in the changelog; reintegration of the changes on the master will result in I/O to the caches requesting these.
- Metadata changes should be completely described in the changelog so that no additional RPCs are needed during reintegration.
- Only “net” changes need to be reported/reintegrated on the master.
- Master must always remain internally consistent (if reintegration can’t complete, master must roll back to an epoch boundary).

2.5 Rollback

Master servers record filesystem changes for each epoch in a rollback log. Reintegration of changes from a writeback cache server also results in rollback log entries on the master. Reintegration fails when one of the master servers reboots in the middle of reintegration. WBC recovery mechanism (not defined in this HLD) decides that this incomplete epoch must be rolled back on all the master servers to reestablish a consistent filesystem state. The committed operations for this epoch are backed out of all master servers by undoing each operation in the rollback log (in reverse order) until the previous epoch boundary record is met.

The requirements of this scenario are:

- Rollback log, including only filesystem changes, stored locally on destination targets.
- Rollback log entries are stored in the same filesystem transaction (journalled) as the operation itself.
- Rollback log entries are created for direct client transactions as well as caching server reintegration. Note that for a caching server, a replication log is created on the cache, and during reintegration of this log a rollback log is created on the master.
- During rollback, committed log entries are rolled back in reverse order, until the previous epoch boundary is met.
- Rollback log entries must contain sufficient detail to fully undo the transaction.
- Old disk blocks are linked to the end of the rollback log so that the original data is stored without copying. When the log is eventually unlinked (rollback is no longer needed), the blocks will be freed automatically.
- Efficient storage of rollback data.

2.6 Coverage

Operations contained in changelogs

- create / delete
- rename
- hard/soft link
- permission change
- extended attribute change
- mtime, atime change
- administrative changes (quotas, filesets, log messages, filesystem proc settings?)
- access violation (audit only)
- epoch boundary
- file writes
- file reads (audit only)

It is worth noting that all the information above except file read/writes is available from the MDTs, and even then there are still the open(2) flags and mtime indications of writes. If byte range information is not required for audit, then there is no need to record (or send) audit changelogs on the OSTs. Rollback logs are still required on the OSTs to recover the original data. Replication logs are still required on OSTs to record modified byte ranges.

3 Functional specification

3.1 Implementation constraints

3.1.1 Base on llogs

Changelogs will be implemented as a flavor of Lustre llogs after relatively minor enhancements are made to the llog facility:

- multiple cancels of a single record (multiple replicators using a single changelog)
- appending old data blocks (for rollback logs)

3.1.2 Create entries only on demand

Changelog entries are created only when required by an existing consumer(s), and are canceled when that consumer(s) has finished processing the change. Filters are placed on the "front end", reducing the number of transactions entering the changelogs instead of "filtering out" entries that are already there. Filters may include operations, files (filesets), users, etc.

Audit logs are only required if audit is turned on. Replication logs are only required for replicas or caching servers. Rollback logs are required for distributed transaction recovery (currently CMD).

3.1.3 Registration

Audit and replication logs are only kept when there is at least one consumer; a registration process is therefore implied. It also follows that logs cannot be generated retroactive to registration (boundless history is not kept).

3.1.4 Independent changelogs per server

Changelogs are per-server, and may be further restricted to a particular fileset. Changelogs from different servers / filesets will not be recombined by Lustre. Applications that require cross-server feeds have to combine separate per-server feeds within the application. Note: a Lustre-level feed combiner could be included later.

3.1.5 Feeds available on 1 client only

Feeds are FIFO files, accessed only though Lustre clients (llite), as <mntpt>/.lustre/feed/<feedname>. The feed is available only on a single client (the one that set up the feed). (A FIFO allows us to "forget" the old entries in the file and prevents the file growing without bounds.)

3.1.6 Replication only on epoch boundaries

Reintegration on a replica should only be executed on full epoch boundaries. They may be executed less frequently - no more often than every X seconds. Too-frequent reintegration will unnecessarily load the filesystem.

3.1.7 No byte range information in audit logs

Audit logs will contain only information currently available to the MDTs. This includes open()s, mtime and atime updates, but not byte ranges. This obviates the need for OST audit logs (not OST rollback or replication logs, however).

3.2 Changelog content

The following information must be stored for every recorded transaction.

1. record length
2. flags (committed, ignore, rolled_back)
3. epoch
4. transaction id (transno)
5. synchronization cookie (shared among a single distributed transaction)
6. reference fid
7. fid forward link
8. fid backward link
9. transaction type (and version)
10. transaction type-specific struct (variable length)

The reference fid is the "primary" fid a transaction is concerned with (some transactions may have multiple fids associated; primary should just be chosen consistently). The fid forward/backward links are a bookkeeping feature to simplify tracking future/past related changes, potentially for single-file rollback or replication. An llog pointer to the last llog entry for each in-memory inode is stored in the inode struct for any subsequent backward link. The forward link may be installed later (see Origination).

The transaction-specific struct includes any items necessary to undo a transaction (for undo logs), replicate a transaction on a remove server (for replication logs), or all audit data (for audit logs).

3.2.1 Replication

For MDT's, the mdt_reint_record information is sufficient (including eadata, logcookies). For OSTs, the object, object version, and extents are required. The replica servers are responsible for translating target names and layout mapping (no assumptions should be made about the replica(s) at log recording time. Adressed in Replication HLD.)

3.2.2 Rollback

Rollback is not a symmetric operation with respect to the information available in the RPC. The log entry must contains the "reverse" transaction; e.g. the "unlink fid 123" transaction must provide the "create file foo under parent fid 234 with fid 123, stripe pattern XXX, and objects[] on osts[]" instructions. (Note this still results in a new inode.) Everything must be restored during rollback. For the unlink example, the MDT must restore the ownership, striping, EA, fid, parent(s), ctime, mtime, atime, mtime of parent(s). (Performance impact - this will effectively require copying the entire inode data, plus the parent list. Potentially we can just mark the inode "pending delete" until epoch commit?)

For MDTs, a mdt_reint_rec structure is sufficient for the basic operation, but *not* the restoration of mtime, ctime, parent fid(s), etc.

1. mdt_reint_rec
2. mtime, atime of file
3. additional operation-specific info: ctime, parent dir(s), mtime, atime of parent dir(s)

RPC	undo record must restore:
unlink	full inode entry (EAs, ctime, mode, owner, etc), parent dir(s) on MDT; objects on OSTs
link	unlink inode on MDT
create	unlink inode on MDT, unlink objects on OSTs
rename	original name, parent
setattr	full inode entry, mtime
write	partially and fully overwritten blocks, mtime

For OSTs, the previous version of the objects are required.

3.2.3 Audit

For MDTs

1. mdt_reint_rec (for detailed event description)
2. consumer list (for multi-consumer audit logs)
3. parent fid (for pathname reconstruction)
4. user id: nid, pid, uid
5. mtime, atime
6. event type (e.g. create, delete, write, read, permission failure)

For OSTs, no audit logs are required (see Implementation Constraints).

3.3 Changelog entry origination

Changelogs are kept per-server for every filesystem operation taking place on that server, either initiated directly by a client or indirectly as part of a distributed transaction from another server.

Every file has a list of filesets of which it is a member stored in an EA. Every file is also automatically a member of the GLOBAL fileset. When audit or replication logs are set up, associated PRE and POST methods are defined for the fileset. The PRE and POST methods contain any applicable filters and describe any changelog-related actions that need to occur before the RPC is initiated (PRE) and after the RPC completes (POST). Whenever a server starts to process an RPC referencing a file, each of the file's fileset PRE and POST functions are called.

PRE is called before a request is processed. POST is called when the epoch containing the request has been globally committed.

PRE and POST methods are derived from policy information in a global database. Policies for a particular fileset are downloaded to a server and cached the first time a member file is accessed on that server.

3.3.1 Rollback

Rollback changelog entries are generated on the master or proxy servers for every transaction that results in a local disk change.

A transaction request must be reversed before it is committed in order to retrieve all the information necessary to undo the operation.

During playback of a rollback log it is unnecessary to propagate the rollback of dependent transactions to remote servers because the other servers will perform their own rollback to the same common, consistent epoch.

3.3.2 Replication

Replication changelog entries are generated on a server for every incoming RPC that results in a filesystem change, when there is at least one registered replication consumer.

Replication logs need to be generated and stored persistently as activity on the server takes place.

It is desirable to reduce the size of replication log before sending. To compact logs, earlier entries may be marked with an “ignore” flag, implying that a later operation will reverse or subsume the effect of the earlier entry, such that only the later entry need be replayed on the master. A forward pointer to the later transaction may also be stored in the earlier transaction (since we're modifying it anyhow) for efficient per-file replay.

3.3.3 Audit

Audit changelog entries are recorded only when:

1. a consumer exists
2. the operations in question meet the filter criteria.

Filter criteria may include specific files (fileset), operations, or conditions (e.g. violations). RPCs that result in items that would be filtered out are simply not recorded in a changelog log.

3.4 Filesets

Feeds are typically defined to watch a particular subset of files or directories on the filesystem. This subset is designated a *fileset.* Filesets may be used for purposes other than feeds as well (replication, mounting a subset of the namespace).

3.4.1 Implementation contraints

1. Recursive

   Directories are always considered fully recursive; any file or directory in the directory tree is part of the fileset as well.

2. Multiple Membership

   A file may be part of multiple filesets. One fileset may implicitly include other filesets. Operations on a file should affect all filesets it belongs to, and vice-versa.

3. Global

   Fileset definitions are shared between all MDTs.

3.4.2 Membership

When a new fileset is set up, the fileset name is added to a global fileset database. This is used for tracking the existence of the fileset, assigning a fileset number, and recording consumer information for the fileset. Consumer information includes audit policies for any active feeds, replication policies, and other fileset description. [?The fileset database should be stored as a regular file on an OST; this is accessable via each MDT's OSCs (do we need llite for file access?)][The fileset database should be stored in the same manner as the Fid Location Database, which is also shared between MDTs.]

When a file or directory is added to the fileset, an RPC is sent to the MDT owning the file/dir, which adds the fileset number to the file's fileset list (an extended attribute). A high-order bit on the fileset number is also set marking this file as part of the fileset definition.

Whenever a file path is traversed, parent filesets are copied to the child's fileset list (with the definition bit masked out), erasing the existing fileset numbers except where the definition bit is set. When a directory is added or removed from a fileset (via the fileset API), the inodes for subtrees under that directory must be flushed so that we re-execute path traversal on lookup. Note that if files are accessed without path traversal they are not guaranteed to be identified as part of the fileset. A file or directory explicitly removed from a fileset via the fileset API will store the fileset number with the definition bit set as well as another high-order bit signifying exclusion.

If a file is moved, it inherits fileset definitions from its new parent, but again retains those with the definition bit set. Rename does not affect the fileset list.

3.5 Feeds

Feeds are the userspace interface to an audit or replication changelog. The feed gates access to the log and translates log entries into a user-consumable form. Every audit log has exactly one feed. A replication log may have zero or one feeds (it may have an internal consumer instead).

A new audit changelog is started whenever a feed is set up via the feed API. Filtering takes place before entries enter the audit log; audit logs are prefiltered.

4 Use cases

See section 2.

4.1 File post-processing

A post-processing operation is to be started whenever a new file is created in /srv/cam1, /srv/cam2, or /srv/cam4. The post-processing operation needs the filename, and should be started when the new file is first closed.

1. Fileset 'towatch' is created using fileset API, below
2. Directories are added to 'towatch' using fileset API
3. Feed is created to watch CREATE and OPEN/CLOSE events on this fileset

4. The feed FIFO is opened by the consumer

5. The consumer loops on feed entries with a blocking read on the feed FIFO. The consumer checks to see when newly created files are first closed.

```
llapi_fileset_new("towatch");
llapi_fileset_add("towatch", "/srv/cam1");
llapi_fileset_add("towatch", "/srv/cam2");
llapi_fileset_add("towatch", "/srv/cam4");
struct feed_policy policy={fp_filtermask=FF_CREATE|FF_OPEN};
llapi_feed_new("towatch", *policy);
fd=open($MNT/.lustre/feed/towatch, O_RDONLY);
loop {
    read(fd, struct feed_entry *entrybuf);
    if entrybuf->fe_type = create then add fe_data.fid to createdlist;
    else if entrybuf->fe_type = close and fe_data.fid is on createdlist then
        postprocess(llapi_fid2path(fe_data.fid));
}
```

5 Logic specification

5.1 Llog modifications

5.1.1 Multiple cancel

An audit log driving multiple feeds requires a reference for each feed. A replication log driving multiple replicators may also benefit. For an initial implementation, making multiple copies of single-cancel llog records is sufficient. As a refinement, a list of consumers could be maintained within each llog entry to avoid multiple copies of the entry.

5.1.2 Store old data

Rollback requires restoration of an object's previous version. For OST objects, old data blocks are removed from the old inode and linked into the end of the llog's block list (and the entry is marked with the appropriate record length to "skip" the data blocks for the next entry). When the llog entry is cancelled and the log (eventually) unlinked, the blocks are freed automatically.

Under ext3, for full-block overwrites, new blocks must be allocated for the new data to replace the old block links in the file extents. The old blocks are removed from the extents and appended to the llog as above. For partial-block overwrites, the previous partial block data must be read and stored directly in the llog.

However, under ZFS, the blocks are already COW'ed and so it is only necessary to add a reference to the old blocks into the llog, or re-link them into a "to-be-purged" file.

5.1.3 Delayed send

In the case of rollback logs, entries may be removed after an epoch global commit. For audit and replication logs, entries are processed only after global commit (see below). Therefore a changelog epoch commit callback is used for every changelog. Log entries should not be sent to a replicator or placed in a feed until this callback; a "delay" bit is added to the llog entry to control this. A llog replicator can not see any entries with this bit set.

5.2 Changelog entry recording

5.2.1 Rollback

A rollback log can be stored on an external device with links pointing back to blocks on the target device. These blocks are linked into a single special hidden file(s) on the target, and removed as undo logs records are cancelled. (See Store Old Data).

Rollback records are recorded before or in the same transaction as the main RPC.

Rollback entries may be cancelled after the entry's epoch has been globally committed.

5.2.2 Audit and replication

Audit and replication changelogs may be stored on the target device or on an external device. These entries must be processed (sent to consumers) *after* a distributed transaction has fully committed. This prevents partial updates from propagating to a replica (in case master crashes and recovers). However, given that the RPC may have been destroyed by the time a distributed transaction is marked complete, the changelog entry info will need to be recorded *before* the RPC is freed.

Thus a two-part write is used for audit and replication changelogs:

1. PRE: record changelog entry (setting "delay" bit)
2. commit RPC
3. POST: propagate changelog entry to consumers (unset "delay" bit, RPC is sent to llog replicators)

The entries can be enabled for propogation after some delay past the commit (it is acceptable for audit and replication to trail changes with some time lag). Therefore entries will be enabled when the server determines the entry's epoch has been globally committed. Multiple committed messages may be collected into a single RPC to llog replicators to reduce network load.

After recovery, servers will eventually establish whether an epoch has been globally committed; the audit and replication entries will wait for that event.

Replication and audit logs are only kept when there is at least one registered consumer. Replication and audit logs may have multiple consumers; entries are cancelled when the all of the registered consumers have processed the entry. See 5.1.1.

5.3 Fileset API

Start a new fileset definition:

```
int llapi_fileset_new(char *fileset);
```

Add a file or directory to a fileset:

```
int llapi_fileset_add(char *fileset, char *filename);
```

Directories added to a fileset refer to the entire subtree. Moving a file out of a subtree removes it from the fileset. A file or directory explicitly added to a fileset retains its membership if renamed.

Remove a file or directory from a fileset:

```
int llapi_fileset_remove(char *fileset, char *filename);
```

A child file or directory may be removed from a fileset that includes the parent. This has the effect of pruning a subtree out of the fileset tree.

Destroy the fileset :

```
int llapi_fileset_destroy(char *fileset);
```

A fileset cannot be destroyed while in use by a feed, mount, etc. (EBUSY).

Retrieve status or other info about a fileset:

```
int llapi_fileset_getinfo(char *fileset, struct fileset_info *info)
```

5.4 Feed API

Feeds provide userspace access to a server changelog. A single user process (consumer) may access a feed. Feeds are transactional and persistent; feed entries are guaranteed to be replayable in the event of a server restart, from the point where the consumer last indicated completion.

5.4.1 Feed content

Feed entries will be packed binary data, with the form

```
struct feed_entry {
    _u32 fe_len;         total record length
    _u32 fe_type;        transaction type
    _u64 fe_seq;         local feed sequence number
    _u64 fe_cookie;      synchronization cookie (for distributed events)
    _u64 fe_time;        event time, server-local
    _u32 fe_result;      return code (0=success)
    void *fe_data;       transaction type-specific struct
```

Transaction types:

```
enum {t_create, t_unlink, t_open, t_close, t_read, t_write, t_attrib, t_rename, t_link
```

Transaction type-specific struct contains event-specific data. (Ideally this will contain enough data for a userspace filesystem replicator; in many cases a MDT_REINT structure would be sufficient.) For example:

```
struct feed_entry_open {
    ll_fid fid;           (see lustre_idl.h)
    ll_fid parent_fid;
    nid_t clientnid;
    _u32 fsuid;
    _u32 fsgid;
    _u32 cap;
    _u32 flags;
    _u32 mode;
    _u32 filename_len;
    char *filename;
```

Feed content examples (expressed in human-readable form, produced by a Sun provided feed consumer demo utility):

```
logid=1 cookie=0 type=OPEN rc=0 name=/etc/passwd fid=23a87346:003d source=cli1@tcp0
logid=2 cookie=0 type=UNLINK rc=-EACCESS fid=23a87346:003d source=cli2@tcp0 uid=joe
```

5.4.2 Feed setup

A feed is created (and available) on a single Lustre client as a FIFO file. New feeds are defined through lfs or a direct call to the liblustre c library (llapi).

```
int llapi_feed_new(char *fileset, struct feed_policy *policy)
```

starts a new audit feed. <fileset> is a previously defined fileset or a server name. Filesets are defined via the Fileset API. Special user permissions are required to start an audit log; else EPERM is returned. The new feed is created as a FIFO at $MNT/.lustre/feed/<feedname>, where <feedname> is <fileset>[_xx], with increasing numerical xx if the name already exists.

<policy> is a structure containing

```
struct feed_policy {
    _u32 fp_filtermask;
    _u32 fp_entry_timeout;
    _u32 fp_abort_timeout;
    _u32 fp_abort_size;
    int  fp_flags;
```

<filtermask> is a bitwise event mask:

mask bit	description	masked types
FF_CREATE	new file creation	t_create
FF_WRITE	file modify/append	t_write
FF_READ	file read	t_read
FF_OPEN	open/close	t_open, t_close
FF_ATTRIB	file attribute / EA change	t_attrib
FF_DELETE	file removal	t_unlink
FF_LINK	soft/hard link	t_link
FF_RENAME	rename	t_rename
FF_FILE	all of the above (shortcut)	t_create, t_unlink, t_open, t_close, t_read, t_write, t_attrib, t_rename, t_link
FF_ADMIN	administrative event	t_admin
FF_ERR	report failed requests also	err & (t_create, t_unlink, t_open, t_close, t_read, t_write, t_attrib, t_rename, t_link, t_admin)
FF_REPLICATE	all events related to fileset replication	t_create, t_unlink, t_write, t_attrib, t_rename, t_link

Retention policies (see Feed I/O below) may include:

- entry_timeout - automatically cancel each feed record after X seconds (0=off, default=0).
- abort_timeout - abort recording and destroy the feed after X second timeout (0=off, default=0).
- abort_size - abort recording after the number of unconsumed records exceeds a maximum (0=off, default=1000).
- flags
  - FG_RATELIMIT - don't report multiple consecutive similar entries.

The feed setup remains persistent across reboots (in e.g. a feed database), until it is explicitly destroyed:

```
int llapi_feed_destroy(char *feedname)
```

Feed setup info and status is available for retrieval from an existing feed:

```
int llapi_feed_getinfo(char *feedname, struct feed_policy *policy, struct feed_status
```

Two functions for converting FID to filename or full path name are also provided. Full path name is at the filesystem's disgression for hard-linked files.

```
int llapi_fid2file(ll_fid fid, char *filename);
int llapi_fid2path(ll_fid fid, char *pathname);
```

5.4.3 Feed I/O

The feed output data stream is presented as a FIFO file (see mkfifo) under $MNT/.lustre/feed/<feedname>. A feed may be open(2)ed by only a single reader at a time. Feed entries are retrieved using read(2) on the file. Reads will block until new entries are available, or poll(2) or select(2) may be used.

Feed entries are removed from the feed according to the following policy:

The file descriptor will make available only full feed entry records. With each read(2) (or close(2)) of the fd, all of the entries from the *previous* read(2) are marked as consumed. For example: read1 reads fe_seq 1-5, nothing is marked consumed; read2 reads fe_seq 6-7, seq 1-5 are marked consumed; close1, seq 6-7 are marked consumed.

Upon recovery, we restart the feed from the first unconsumed entry. The feed consumers are responsible for skipping any replayed feed entries they may have already processed (identified by repeated fe_seq).

5.4.4 Feed entry ordering

For consumers combining the data from multiple feeds (e.g. database), partial ordering of operations is required. Since clients synchronize the epoch among servers for sequential transactions, using the epoch is sufficient to order "before relations", while unrelated transactions may be reported in a random order. Distributed transactions are linked via a common cookie contained in all affected changelogs and reported in all affected feeds. Consumers are reponsible for linking/ignoring distributed transaction entries internally.

6 State management

6.1 State invariants

6.2 Scalability & performance

6.3 Recovery changes

6.4 Locking changes

6.5 Disk format changes

6.6 Wire format changes

6.7 Protocol changes

6.8 API changes

6.9 RPCs order changes

7 Alternatives

Feeds may be defined as regular files instead of FIFOs, which allows multiple readers and multiple clients. However, this complicates entry cancellation (we need an explicit cancel), but more significantly the file would grow without bound. The beginning of the file would be empty where entries were cancelled out of it, but the file size would still have to grow for seek(), etc. A FIFO presents a less confusing picture to users.

8 Focus for inspections