OpenAFS Byte-Range Locking 2007

Matt Benjamin
Linux Box Corporation

Marcus Watts
University of Michigan

Who Are We?

- Marcus: programmer employed by University of Michigan
- Matt: programmer with Linux Box, a (free/open src) software contractor in Ann Arbor, MI
- Collaborators on OpenAFS development since 2004

AFS and File Locking

AFS has supported whole-file, advisory locking only since inception

- Byte-range file locking had not been considered important, esp. for Unix clients
- Became more important for MS Windows clients (important applications [MS Office] rely on it for their operation)
- Locking enhanced in DCE/DFS (AFS 4), which also strengthened consistency, AFS 3 left behind in both areas
- OpenAFS has recently added client emulation of byte-range locking for 2 platforms

AFS and File Locking

OpenAFS has recently added client emulation of byte-range locking for 2 platforms

- Asanka Herath and Jeff Altman have byte-range file-locking emulation implemented in the Windows client
- similar behavior on Linux was achievable by delegating locks to the VFS layer
- but this doesn't really help all that many real applications (doesn't actually allow sharing the same file between clients, with full semantics)

How Locking Works Today

a. server/rpc interface

1. lock counts and a timestamp are stored as file metadata, presently these are ignored by vos operations (eg, dump)

a) OpenAFS doesn't support read-write replication

i. if it did, possibly locks should be replicated

2. SetLock, ReleaseLock, ExtendLock

a) client can release a lock it does not own

b) race prone

c) poll intensive

1) clients must make ExtendLock call not less than every 5m for each asserted lock, or it is invalidated at the server

What OpenAFS Comm. Has Wished For, Over The Years

a. byte-range lock emulation (cache managers)

1. implement byte-range locking in the cache manager, with a whole-file lock asserted on the file server

b. atomic lock up/down grades (cache managers, rxrpc, fileserver)

1. spec published by Jeff Altman

c. improved cm lock recovery (cache managers)

1. port to Unix, where applicable

d. actually track lock owners at the fileserver

What OpenAFS Comm. Has Wished For, Over The Years

`

e. "mandatory locking"

1. used to mean, emulation of Windows file consistency semantics (including mandatory locking)

2. the Unix world standardized on advisory locking

f. distributed byte-range file locking (advisory or mandatory)

1. has been viewed by the developers as unnecessary

Analysis In 2007

a. cache-manager byte-range lock emulation (It's Done)

1. completed on Windows, where it matters most (Jeff and Asanka)
2. magically avoided on Linux, could be extended to other Unixes with modern VFS implementations

b. atomic lock up/downgrades

1. critical but implementation for current Unix locking implementation is a waste of effort

Analysis In 2007

c. improved cm lock recovery specific to Windows lock emulation implementation

1. done, where it matters (Windows client)

`

d. client tracking

1. critical but implementation for current Unix locking implementation is a waste of effort

e. “mandatory locking”

1. Unix clients just don't need it

2. Windows clients can emulate it (and already do)

a) but full support for mixed Windows & Unix environments probably does require server-side support

i. (NetApp, NFSv4, etc)

Analysis In 2007

`

e. distributed byte-range file locking

1. is the real problem worth solving
2. is the general case of the “mandatory” locking
3. is more challenging but technically sound

Distributed Byte-Range Locking

1. overlaps with file consistency guarantees

    a) but is not the same thing either

2. has implications for cache implementations (block/chunk design)

3. requires fileserver to efficiently communicate richer locking and file consistency information to clients, and coordinate operations among clients

    a. we propose enhancement of callback mechanism already in AFS for this purpose

    1) logically

    2) in consideration of similar systems

    3) having considered security and performance implications

What We Are Implementing

a. distributed, byte-range file locks in the fileserver

b. proper distribution model, based on callbacks

c.. server tracking of lock clients by their unique identifiers

d. atomic up/downgrade

3. support for “delegation” (ie, leases, oplocks)

  a. because most files still aren't being read and written by 1+ clients simultaneously

  b. The Simplest Design That Could Work

    i. degrade to server-coordinated locking when 2+ clients lock in the same file

c. enhanced file consistency when byte-range locking in use, and/or under administrative control

d. Windows-compatible semantics (exclusive locks, mandatory locks)

Staged Implementation

`

a. Phase I (fairly far along here)

1. rpc interfaces and implementation of the calls
2. in-memory implementation in the fileserver callbacks or saved state
3. naive cache manager implementation, no delegation
4. advisory only

Staged Implementation

b. Phase II

\`

1. afscbint.xg interfaces and implementation of the calls
2. server callback implementation (wait on lock)
3. server tools for issued-lock reporting
4. mandatory locking support (needs discussion)

(Phase II is a useful deliverable)

Staged Implementation

`

c. Phase III

1. delegation
2. persistent lock state

File Consistency

a. even if locks are advisory, consistency model is broken if a client with a write lock asserted on a range has a stale copy in cache

1. we propose to add cache flush barrier when a client releases a write lock

2. when flushing a range that is a subset of a chunk, the rest of the chunk should not be flushed

i. proposed solution: make chunks of variable size, allow the cache manager to ensure that a write-locked range is a chunk of the locked size, whatever that is; the flush-on-release can then only flush a range that was locked

File Consistency

b. apparently, the Windows client flushes all writes immediately (discards sync-on-close)

1. this could be considered a reasonable behavior for Unix clients, too

a) what are community views on this?

c. increased consistency will result in more frequent invalidates

1. one response could be more granular invalidation than whole-file data version updates

a) eg, implement a “range invalidate” callback

a) this adds complexity (and increases workload) at the file server and (to a lesser extent) the clients

Delegation

a. is our version of file leases/oplocks

b. delegation could accomplish at least two efficiencies

1. reduce server workload

    a) allow client to manage local range locks on an entire file if no other client is currently interested

2. reduce lock traffic

    a) currently, each allocated lock generates (something like) a lock extension message from the client per lock, per minute

c. introduces complexity

1. suggest model where server revokes delegations and manages all locks itself whenever multiple clients are interested

    a) minimizes complexity and optimizes for the 2 most important cases

Callbacks

a. the core function of the callback interface is to allow the server to invalidate cached file information

  1. it does this by sending an updated data version number to the client

b. we propose to use the callback channel for two new classes of operation

  1. efficiently express inter-client intent to wait on a lock, and fileserver notification when locks are granted

  2. efficiently implement delegation

    a) fileserver must be able to

      i. revoke file delegation at will, and

      ii. request any locks a client had issued to local processes while a file was delegated, so that they can be asserted on the fileserver

Callbacks (Issue)

c. the callback connection is currently not authenticated

1. this has been viewed as acceptable because in the worst case, a malicious call could force a client to request data (over an authenticated channel) unnecessarily
2. adding new responsibilities to the callback mechanism changes its security implications
3. fortunately, OpenAFS is committed to resolving this issue in rxgk, and probably, rxk5 (we are the developers of rxk5, and are in a position to do this)

Additional Considerations

a. Planned Reboots

1. planned restarts should not require revocation of all outstanding locks

a) save lock state

b. Fileserver Failure

2. the unfriendly version of a

a) how hard should the file server work to preserve locks across a crash?

c. Network Partition

1. neither client nor server has strong interest in preserving locks during an extended outage

d. Starvation (unfair lock contention behavior)

e. Replication (speculative)

Locking and Replication

a. Honeyman/Zhang work defines relationship between locking and one form of advanced replication (NFSv4-r)

1. key algorithm is advanced delegation, where file server can delegate a lock on a VFS subtree (to another file server)

a) complicated

b) efficiently supports multiple writers

c. active/passive read-write replication does not require NFSv4-r locking model

1. but probably would include lock replication

Implementation Remarks

`

a. code sharing between between fileserver and Unix cache managers can be fairly pervasive, and we are attempting to promote this

b. code sharing with Windows lock emulation code is less easy but many notions are conceptually similar (and similar to equivalent code on BSD and other Unixes, which agree on basic concepts and could almost share routines)

Implementation Remarks

c. Windows lock emulation consists of a cache of "keyed byte-range locks"

1. fileserver implementation is logically similar

a) fileserver maintains (unthreaded) AVL tree of nodes keyed by FID

d. locks are tracked as a linked list of lock segments on some node

1. on the fileserver, lock segments are associated with the client unique identifier, which is already implemented since OpenAFS 1.4.x

2. in the Unix cache manager, lock segments are associated with process ids

e. delegation looks like conditional execution of lock conflict code on the client, if delegated, or the server, if not

Questions and Discussion