Replicated Filesystem Protocol

by Dieterich Lawson, jdlawson@stanford.edu

Introduction

This protocol outlines the behavior for a replicated file system. With this system, a client can perform various local manipulations on files (**yes, multiple files**) which are mirrored out to several servers over the network. Note that this protocol description is made specifically in reference to the C/C++ API defined at http://www.stanford.edu/class/cs244b/replFs.html/.

Definitions

A **write** is a request to modify a file made via a call to `writeBlock()`. When `writeBlock()` is called, no writes to disk are performed. Instead, writes are staged inside of a 'commit' (discussed below) in the order that they were requested. Then, when `commit()` is called, all writes in the current commit are written to the specified file, in order. Each write is given a **Write Num**, which uniquely identifies the write within that commit. Write Nums start at 0 with each commit or abort, and are incremented by one with each write. There can be at most 128 writes in one commit, so the largest Life Num possible is 127.

A **commit** is a sequence of staged writes that will be written to a specific file when `commit()` is called. Each commit is given an ID, called a **Commit Num**, which uniquely identifies the commit within the context of the associated file. Commit Nums start at 0 with the opening of a file and are incremented by one with every call to `commit()` or `abort()`. There can be only one pending commit per file, meaning that it is not possible to stage writes into different commits for the same file at the same time. When `commit()` is called, all the writes since the last call to `commit()` or `abort()` will be written on the files on disk, and a new, empty, commit will be created.

Writing

To stage a write to a file, the user should first issue a call to `writeBlock()` with the proper parameters supplied. The client will then send out a **WriteBlock** packet to all the servers, which instructs them to stage that write in their internal data structures. Note that the write will not be performed until a Commit packet is received.

Committing

To begin a commit, the user should make a call to `commit()` and pass in the appropriate file descriptor. The client will then send a **CommitRequest** to the servers containing information identifying the commit and the number of writes that the servers should have staged for that commit. If the servers have all the writes required, then they should respond with a **ReadyToCommit** packet.

The CommitRequest packet will be resent at 200 ms intervals until either a ReadyToCommit packet or a WriteResendRequest packet is received from all servers. If a server does not respond with one of those packets within 2 seconds, it is considered unreachable and the

commit will fail. Servers should ignore duplicate and irrelevant (as determined by their File ID and Commit Num) CommitRequest packets.

If the server is missing some of the writes, it must send a **WriteResendRequest** packet to the client to ask the client to resend certain writes. The WriteResendRequest contains information about the commit in question as well as a bit vector indicating which writes the client needs to resend. The bit vector will have a '1' at the nth index if the server needs the nth WriteBlock packet resent. The client should re-send all the requested WriteBlock packets.

The servers will continue to send WriteResendRequest packets at 200 ms intervals until all writes have been accounted for. Additionally, the servers should update the WriteResendRequest packets to reflect any resent WriteBlock packets they have received in the intervening time. Once all writes have been received, the server should send a ReadyToCommit packet to the client. If 2 or more seconds pass between WriteSendRequest packets from a server (or a ReadyToCommit packet) then that server is considered unreachable and the commit will fail.

Once all servers have responded with a ReadyToCommit packet, the client should send out a **Commit** packet. This packet instructs the servers to write all writes staged in the specified commit to the file on disk. After doing this, the servers should respond with a **CommitAck** packet. The Commit packet will be resent at 200 ms intervals until all servers have responded with CommitAck packets. If a server does not respond within 2 seconds with a CommitAck packet, it is considered unreachable and the commit will fail. Servers should ignore duplicate or irrelevant (as determined by their File ID and Commit Num) Commit packets.

Aborting

Aborting is used to discard any changes a user has staged via `writeBlock()` since the last commit or abort. To abort changes, the user should call `abort()` and pass in the relevant file descriptor. The client will then send out an **Abort** packet to all the servers, which signals that the currently staged changes should never be performed. The servers should discard any old housekeeping data related to the commit, and acknowledge the message with an **AbortAck** packet sent back to the client.

The Abort message will be resent every 200 ms until either all servers have acked or 2 seconds have passed. However, if 2 seconds have passed and all servers have not acked, then the call will **not** fail. Instead, the failure to ack will be ignored, and execution will continue as normal. Servers should ignore duplicate and irrelevant (as determined by their File ID and Commit Num) Abort packets.

Note that as long as the unacknowledging servers are not actually down, they do not present a problem. The abort message is really just a hint to the servers that they can discard housekeeping data to help them keep their memory footprint low and is not actually required for correctness.

An abort opens a new commit on the file, meaning that the Commit Num will be incremented by one.

Initialization

Before the system can be used, some housekeeping must be performed. Specifically, the servers must all be assigned non-conflicting IDs, and the number of responsive servers must be determined to be equal to the `numServers` parameter of `InitReplFs()`.

When `initReplFs()` is called, the client begins initialization by sending a RollCall packet out to all servers. Each server will randomly generate a proposed Server ID and respond to the RollCall with a RollCallAck packet. The client will observe these packets for 500 ms (or some other user-defined amount of time). If, after 500 ms, the client has not seen `numServers` distinct Server IDs, it will send out another RollCall packet, which will cause the servers to re-generate their Server IDs and send them out via RollCallAck packets again. If the client still has not seen the proper number of distinct server IDs after several rounds of this process, it will assume that there are not enough servers to start the system and initialization will fail.

This process handles the case where two servers randomly generate the same ID because the client will see two servers with the same ID as one server, which will cause the client to assume that there are not enough servers present and ask all servers to re-generate their IDs. While the probability of pseudo-randomly generating the identical 32-bit numbers are relatively low, this process ensures that the system will never start up with colliding IDs, and that identical IDs would have to be generated multiple times in succession for the system to fail erroneously.

Note, however, that this assumes there will never be *more* servers listening than are required for the client to successfully initialize the system. If there are more servers than necessary then the system could erroneously start up with colliding IDs.

Opening a File

When a call to `open()` is made, the client first generates a 32-bit random number that will be used as a unique ID for the file. This is called the File ID in the packet descriptions below, and will be given to the user as a ‘file descriptor’. Note that the File ID will not ever be -1, because that is the value returned from `open()` when the call fails. Calling `open()` on an already open file will should simply return the file descriptor without doing anything.

After generating the File ID, the client sends an OpenFile packet containing the File ID and filename out to all the servers. The servers should use this information to initialize any necessary internal housekeeping data, but must not actually create the file on disk until a commit is executed on that file.

The servers must acknowledge the OpenFile packet with an OpenFileAck packet sent back to the client. The OpenFile packet will be resent at 200 ms intervals until either acks are received from all servers or 2 seconds have passed. If 2 seconds pass and the client has not received acks from all the servers then the remaining servers are assumed unreachable and the call will fail. Servers should ignore OpenFile packets if they have already opened the specified file.

Closing a File

When a file is closed, an attempt will be made to commit all outstanding changes to the file. Then, the file will be closed. To close a file, the client begins a commit cycle like normal with a CommitRequest, but after all servers have signalled they are ready to commit, the client sends out a Commit packet with the Close Flag set to 1. This signals to the servers that they can close the file and release any local resources handling that file's information.

A call to `close()` will fail if some of the servers are unreachable, exactly like a call to `commit()`. Note that it is also possible to close a file via a call to `abort()`, but that ability is not exposed through the existing API. A call to `close()` can also fail if the supplied file descriptor is invalid.

Data Encoding and Transport

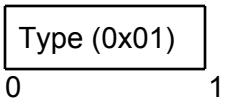
Unless otherwise specified, all numbers are 4-byte unsigned integers. All data is in big-endian byte order. All packets are sent using a UDP multicast group.

Descriptor Definitions

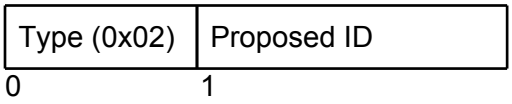
Name	Description
RollCall	Sent by the client to request that the servers identify themselves as active and provide their proposed ID to the client.
RollCallAck	Sent by the servers to identify themselves and communicate their proposed ID to the client.
OpenFile	Sent by the client to inform the servers that a file has been opened.
OpenFileAck	Sent by the servers to acknowledge an open file request from the client.
WriteBlock	Sent by the client to tell the servers to stage a write to a file.
CommitRequest	Sent by the client to check if the servers have all the writes required to perform a commit.
ReadyToCommit	Sent by the servers to inform the client that they have all required writes and are ready to perform a commit.
Commit	Sent by the client to tell the servers to perform all the writes staged in a commit. Only sent after all servers have reported that they are ready to commit.
CommitAck	Sent by the servers to inform the client that a commit was performed successfully.
WriteResendRequest	Sent by the servers to request that certain WriteBlock descriptors be resent before a commit can be made.

Abort	Sent by the client to tell the servers to abort a specific commit and discard all associated queued writes.
AbortAck	Sent by the servers to tell the client that an abort was performed successfully.

RollCall

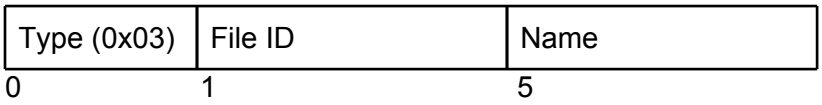


RollCallAck



Proposed ID The proposed Server ID for the server sending the message.

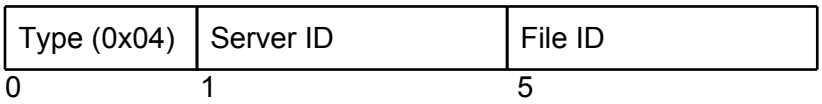
OpenFile



File ID A new File ID for the file being opened.

Name The name of the file encoded in an ASCII string. Does not have to be null terminated. Can be at most 128 bytes long.

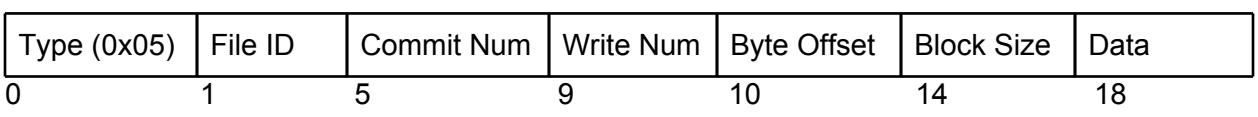
OpenFileAck



Server ID The ID of the server sending the acknowledgment

File ID The ID of the file that was opened.

WriteBlock



File ID ID of the file being written to

Commit Num The Commit Num of the commit that this write is part of

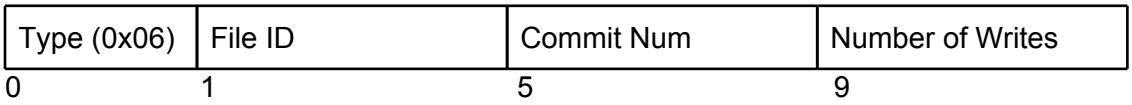
Write Num 1-byte unsigned integer encoding the Write Num of this write. Can be at most 128.

Byte Offset The offset into the file, in bytes, where this write should take place

Block Size The size in bytes of the data to be written

Data The data to be written

CommitRequest

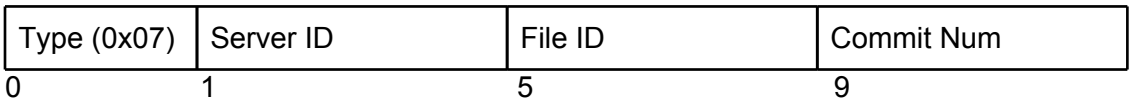


File ID ID of the file to which changes are being made

Commit Num The Commit Num identifying the commit to be performed

Number of Writes 1-byte unsigned integer encoding the number of writes that the servers should have staged for this commit. Will be at most 128.

ReadyToCommit

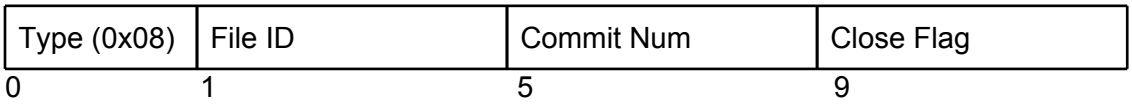


Server ID The ID of the server that is ready to commit

File ID The ID of the file to which changes will be made

Commit Num The Commit Num of the commit to be performed

Commit



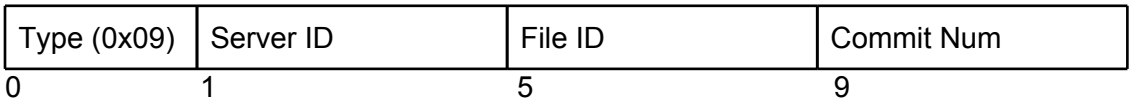
File ID The ID of the file that changes should be written to.

Commit Num The Commit Num of the commit to be performed.

Close Flag A 1-byte unsigned integer indicating whether or not the specified file should be closed after this commit is performed. Will be 1 if the file should be closed, and

0 otherwise.

CommitAck

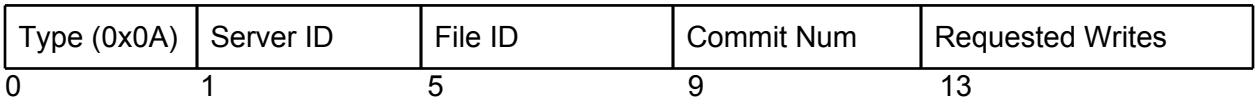


Server ID The ID of the server acknowledging the commit.

File ID The ID of the file that changes were committed to

Commit Num The Commit Num of the commit that was performed.

WriteResendRequest



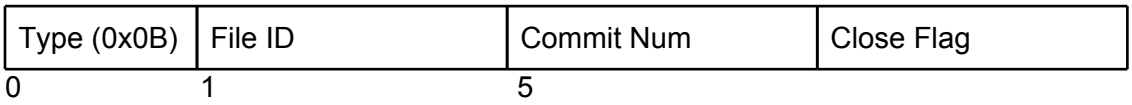
Server ID The ID of the server sending the request.

File ID The ID of the file that the writes were made to

Commit Num The Commit Num of the commit that contains the writes

Requested Writes A 128-bit (16 byte) bit-vector encoding the numbers of all requested writes. A ‘1’ at the nth index in the bit-vector indicates a request to resend the nth write. A ‘0’ at the nth index indicates that the requesting server does not need the nth write resent.

Abort



File ID The ID of the file on which a commit is being aborted

Commit Num The Commit Num of the commit to abort

Close Flag A 1-byte unsigned integer indicating whether or not the specified file should be closed after this abort is performed. Will be 1 if the file should be closed, and 0 otherwise.

AbortAck

Type (0x0C)	Server ID	File ID	Commit Num
0	1	5	9

Server ID The server sending the acknowledgement

File ID The file on which a commit was aborted

Commit Num The Commit Num of the aborted commit

Evaluation

There are several key points that differentiate the approach taken here from a more traditional reliable-transport approach. Some of the most interesting are:

Fast Writes, Slow Commits
If we were using reliable transport then our writes would have to be acknowledged before the Write() method could return, making Write() much slower. Our current approach is to just ‘fire and forget’, meaning that calls to Write() are a lot faster at the expense of slower calls to Commit(). This isn’t necessarily a better approach, but could be appropriate for write-heavy applications with fairly frequent commits.
Less Complex Protocol
If we used a reliable transport protocol, then that would offload a lot of the complexity of the protocol onto the transport. Many of the ack packets could be done away with, and the protocol as a whole would be much simpler. Honestly, I think having the reliability mechanisms built into the protocol itself is a useful teaching tool, but other than that is mostly a bad idea because it reinvents the wheel and makes the protocol more complex. An argument could be made that having reliability in the higher-level protocol makes it more tunable and configurable, but the reliable transport protocol should be tunable as well if you know what you’re doing.

More Robust System
Many of the attempts at reliability in this protocol are relatively primitive and tie up the client’s resources. Using an established reliable transport protocol would be building on the work of hundreds of people across decades, making it a far more reliable choice. Additionally, some transport protocols have lower-level support in the routing system for their reliability mechanisms, meaning that the client’s resources aren’t tied up as much in resending packets, waiting for acks, etc...

Future Directions

There are many possible extensions and refinements that could be made to the system as is. Some of the most interesting/important are:

Server Recovery & Fault Tolerance

Currently, when a server dies, it crashes the system and is gone forever without the chance to come back online. In the ‘real world’, servers would be failing and coming back online all the time, and a more flexible stance towards the number of servers, along with server recovery features would be very helpful in scaling this system.

Inter-Node Gossip
If the network became large enough, it could be useful to incorporate inter-node ‘gossip’ as a way to pass information around. For example, if you suspect that a node is dead you could ask some of its nearest neighbors when the last time they heard from that node was. In a different vein, if some servers have updates that other servers need, they could pass them around amongst themselves without having to poll the client every time.

Reading from the Servers
Currently the only way to read the replicated files is by accessing the servers’ file systems. It would be nice if the client could somehow query the servers for files & info over the network.

Memory Management
Once you start manipulating very large files or lots and lots of files, holding everything in memory could become a problem. A mechanism that caches the commits on disk could be useful. Alternatively, commits could be optimistically written to disk and rolled back from a disk-cached backup copy of the file if Abort() was called. The approach would have to be tuned to the specifics of the use cases.

More Robust Write Error Checking
If write sizes start getting large, then a mechanism to ensure that the write gets broken up into smaller chunks, and that those chunks are properly transmitted would be very useful. This could involve a checksum system and a more pessimistic commit protocol.

Privacy & Security
Currently, any server can jump in and start listening to updates without either identifying itself or even announcing that it’s listening. Additionally, any client can begin broadcasting to a set of servers. Some privacy and security controls would be very necessary for a real-world deployment.