Distributed Replicated File System

Final Report

Yunlu Li yunluli@stanford.edu

Section 1. Protocol Specification

The protocol describes the basic algorithm design of the distributed replicated file system, and specifies format of different packets used for communication between participating clients and servers. Note: this differs from the previous version submitted.

General Specifications:

1. Each server generates a 32bit unsigned integer randomly as its ID upon start. It keeps the ID for its entire life time. Since server number is limited to 16, the chance for ID conflict is negligible.
2. Packet retry interval is 500ms, and num of retry is set to 10.
3. numServers is specified at init phase, and is adjusted by client in following operations. It is decreased if client detects that some servers are offline.
4. The system supports at most one open file by a single client at a time.
5. Client is responsible for maintaining the write sequence number. It is a non-decreasing unsigned integer.
6. Server caches all uncommitted staged writes in memory, and only flushes that to disk if client calls commit(). This simplifies the abort() logic.

Function Semantics

int InitReplFs(unsigned short portNum, int packetLoss, int numServers);

This function does not involve any communication between client and server. It just does basic initialization work on the client side. It sets the count of available servers, regardless of how many servers are actually available. It also sets the packet drop rate, and creates communication socket for the client.

int OpenFile(char *name);

Client is responsible for generating a non-negative integer as file descriptor when it calls the function. It sends out an OPEN packet with the file name to servers.

When server receives an OPEN packet, the logic it should follow is quite subtle. It first checks whether there's already an open file. If so, it then checks whether there's uncommitted writes for that open file. If not, server acknowledges this OPEN packet, closes the previous file, and keeps track of the new file. If the commit log is non-empty, server would just fail the open request.

Upon receiving acknowledgements from servers, client then gathers OPENACK packets. It retries 10 times. If number of OPENACK packets received from different servers is less than numServers, OpenFile() should return -1.

If successful, i.e. number of different OPENACK received from different servers equals to numServers, the OpenFile() returns the integer generated to the client as a legal open file handle.

The semantics above ensures that only if all servers available acknowledge the same client, this function will return successfully. If multiple clients try to open a file at the same time, at most one could succeed.

If client tries to open the same file again, OpenFile() could use states stored locally to return the same file descriptor.

int WriteBlock(int fd, char *buffer, int byteOffset, int blockSize);

Client sends out WRITE packet to inform servers to stage the writes. Both client and server keep a local copy of the write for retransmit and commit purpose.

The function returns -1 if the client violates the maximum block size, file size, or number of staged writes before Commit().

int Commit(int fd);

Client first sends out CHECK packet to check whether all servers have all staged writes. Servers respond with CHECKRES packet. The missing field in the packet is a 64-bit bitmap indicating the missing records. If it's all zero, it means no missing writes on the server side, otherwise client needs to resend some writes. Sequence numbers of missing writes are encoded in the bitmap.

The process above will repeat until the client has no CHECKRES response with non-zero missing. It then sends out COMMIT packet. Server responds with COMMITABORTACK upon successful commit. Client retries 10 times and waits for COMMITACK from servers. It then adjusts numServers according to number of COMMITABORTACK packets received.

Server should ignore COMMIT packet with invalid file descriptor. Commit() returns -1 directly if fd passed in is invalid.

int Abort(int fd);

Client sends ABORT packet 10 times and waits for COMMITABORTACK packet from server. It then adjusts numServers according to number of COMMITABORTACK packets received.

Abort() returns -1 if fd passed in is invalid.

int CloseFile(int fd);

CloseFile() is very similar to Commit(). Client just goes through the check and commit phase. It also needs to clear the commit log.

Server doesn't need to differentiate Commit() and CloseFile(). Packets sent to server in these two cases are all just check and commit packets. This is related to the logic of open() on the server side.

Client should also adjust numServers in this case, so that subsequent OpenFile() won't fail with weird errors.

CloseFile() returns -1 if fd passed in is invalid.

Packet Descriptions

Packet Type	**Description**
OPEN	Sent by client to request opening a file
OPENACK	Sent by server indicating the open request is successful
WRITE	Sent by client to transmit a staged write to server
CHECK	Sent by client to check whether server has all staged writes
CHECKRES	Sent by server indicating check result. Missing writes are encoded in the bitmap sent back in this packet.
COMMIT	Sent by client to request server to flush all staged writes to disk
COMMITABORTACK	Sent by server indicating successful commit/abort
ABORT	Sent by client to request server to abort all staged writes

Packet Details

Each row of the table takes 4 bytes.

OPEN

Packet Type (0x00)
File Descriptor
File Name (128 bytes maximum) …

OPENACK

Packet Type (0x01)
Server ID
File Descriptor

WRITE

Packet Type (0x02)
File Descriptor
Write sequence number
Byte Offset
Block Size
Data (512 bytes at most)

…

CHECK

Packet Type (0x03)
File Descriptor
Number of Staged Writes

CHECKRES

Packet Type (0x04)
Server ID
File Descriptor
64-bit Bitmap Showing Missing Writes

COMMIT

Packet Type (0x05)
File Descriptor

ABORT

Packet Type (0x06)
File Descriptor

COMMITABORTACK

Packet Type (0x07)
Server ID
File Descriptor

Section 2. Evaluation

This section evaluates this approach to replication versus using reliable transport.

Merits

1. Group delivery semantics

Our replicated file system makes intensive use of UDP group multicast for clients and servers to update shared state. It's possible to do reliable delivery to a group of nodes much more efficiently than TCP's point-to-point acknowledgement.

2. Out of order delivery
   Our application is not sensitive to out-of-order delivery of packets. For example, write requests can come out of order and the phase 1 of commit operation will ensure all available servers have consistent copy of commit log. By using UDP transport mechanism, we reduced the application latency significantly. Especially for those write intensive applications, our file system could be very efficient. The overhead added by TCP could reduce the write QPS we can tolerate.

Disadvantages

1. Reliability
   Since the packet drop rate could be high for UDP (up to 10%), we are forced to implement wait and retry logic on the client side. That makes a lot of operations like open file, commit and abort changes block for a potentially long time (up to 5s in this design). The reliable transmission ensured by TCP could make these operations much faster.
2. Congestion control
   One of the greatest merits of TCP is congestion control. Our UDP transport channel could be overloaded and increase the packet drop rate even further. In that case, we may either need to increase the retry timeout, making the system even slower, or experience more failures.

Section 3. Future Direction

1. OPENACKFAIL packet to speed up the OpenFile() API
   The current semantics of the OpenFile() API makes the client blocks for 5 seconds if it’s not able to gather enough OPENACK packets. An OPENFAILACK packet could be added to speed up the process. If one of the servers detects it has an open file already, it could respond to the open request with a ‘reject’ message so that the OpenFile() call on the client side could fail immediately.

2. Deal with server come and go
   In real life, it’s very common for servers to come and go all the time. If not dealt with correctly, fresh servers could confuse clients and cause byzantine failures to the system. This protocol design only keeps track of server number on the client side, since there’s the assumption that servers won’t come back to life after they die. In order to deal with server reboot, client needs to keep a list of trusted servers and verify packets received.

3. Deal with client failure
   This design doesn’t deal with client failure at all, whereas it’s a must for real life deployment. For example, if a client makes several write requests to available servers and dies, the servers won’t be useful to other clients. That’s determined by the open() semantics on the server side. Servers won’t acknowledge new open request if it has open file with uncommitted changes. Another example is, if client fails after sending commit packet but before gets any commitack response, it’s hard to determine which server has which version of the file.

4. Back off retry
   Current retry on the client side are made in equal time intervals, i.e. every 500ms in this design. However, in real life, there should be a back off mechanism. Retry is needed because the network is somewhat overloaded, or has limited bandwidth, so that the packet drop rate is high. A constant retry will harm the network even further, and this accumulated effect is very bad.

5. Tradeoff between consistency and availability
   This design favors consistency absolutely and sacrifices availability in most of the cases, including open file, close file, commit and abort changes, although write request completes very fast. Real system has customized specs, so might favor availability over consistency in some cases.

6. Support of concurrent clients
   The current system only supports one open file by one single client at a time, which is very inefficient. In real systems, servers are usually able to handle concurrent clients, although write conflicts might happen if two or more clients open the same file at same time. At least, it's better to support different clients operating on different files.

7. Communication between servers
   Current design doesn't support communication between servers. This mechanism might be useful to deal with scenarios like server id conflicts. That's especially important if number of servers goes up to, for example, several thousands.