CS244B Replicated Filesystem

Protocol specification

Maliu2@stanford.edu

1. Packet formats and semantics

init	client send it when init() is called
initACK	server's response to init packet
open	client send it when openfile() is called
openACK	server's response to open packet
writeblock	client send it when WriteBlock() is called
check	client send it when Commit() is called
vote	server's response to check packet, indicate whether it is OK to commit
ResendRequest	server's response to check packet, if this server didn't get any writeblock in this transcation
commit	when client received voteYes from all server, it send this packet to commit this transcation
commitACK	server's response to commit packet
abort	when client received some voteNo during commit() or get called by abort(), it send this packet to abort this transcation
abortACK	server's response to abort packet
TranscationRequery	if the server received check packet, and reply voteYes for it, but it didn't get any commit/abort for 1 minute, it will send this packet to ask other servers if this transcation is committed.
TranscationResponse	Response to TranscationRequery, indicates whether that transcation is committed or abort or get nothing

How all the packets relate to others are explained in section event sequence. The fields in each packets are elaborate below. Each row is a 4-byte field.

Common fields:

Type: Packet type, the descriptor of the packet. Each type of packet has a unique number.
Client ID/Server: 32 bits unique ID for each client/server indicating the sender/receiver.
File Descriptor: The unique ID of files in each client.
Transaction ID: The unique monotonically increasing number for each client's current transaction.
Write Sequence number: The ID for each writebock() call, range from 0 to 127.

init packet:

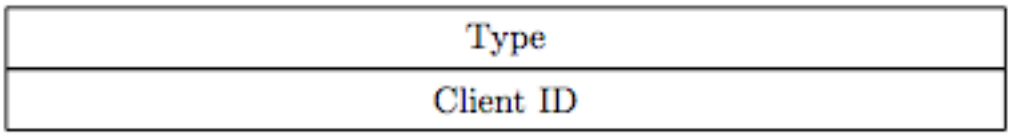

Figure 1: init Packet

initACK packet:

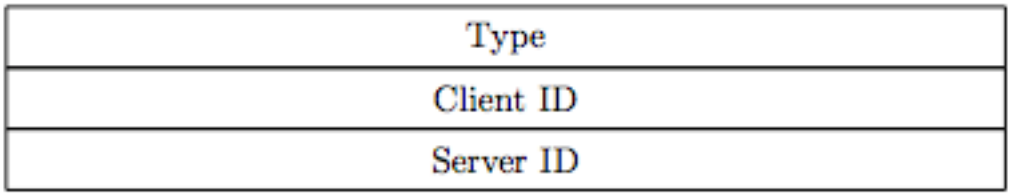

Figure 2: initACK Packet

open packet:

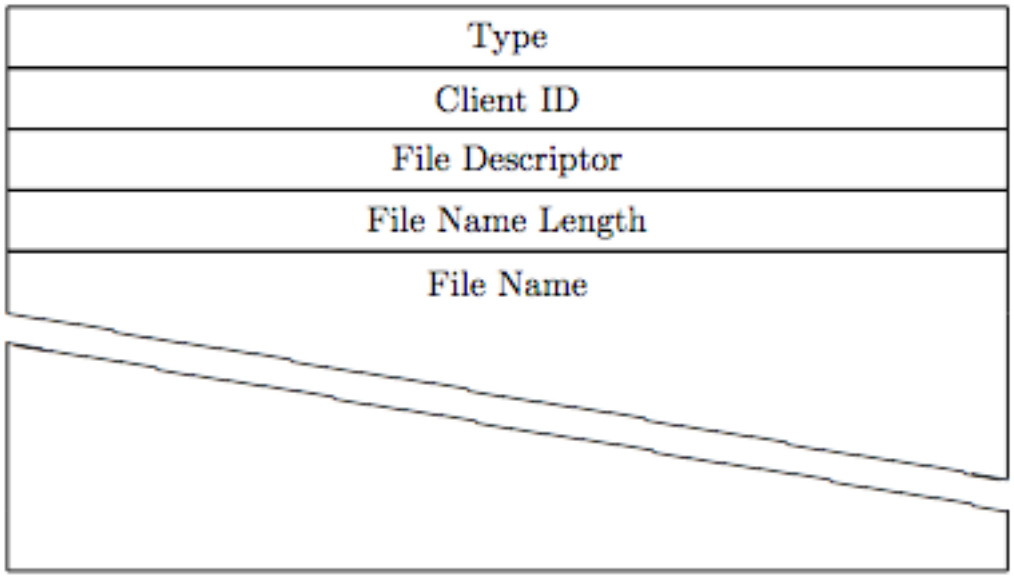

Figure 3: open Packet

openACK packet:

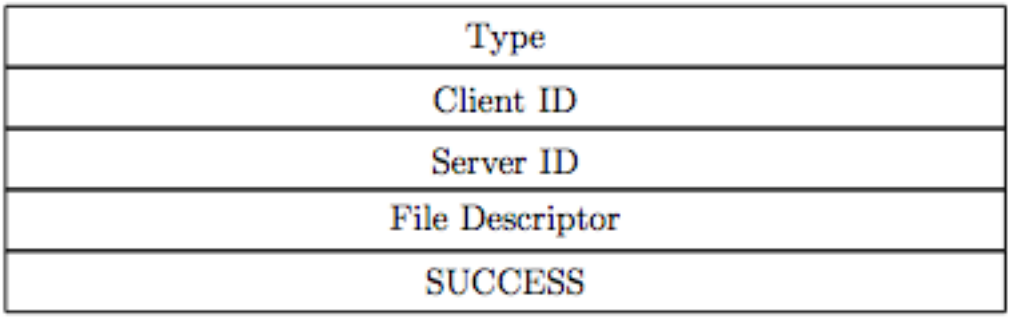

Figure 4: openACK Packet

SUCCESS field indicates whether the server has successfully open the file. If the file name is opened by other client, it will return 0.

writeblock packet:

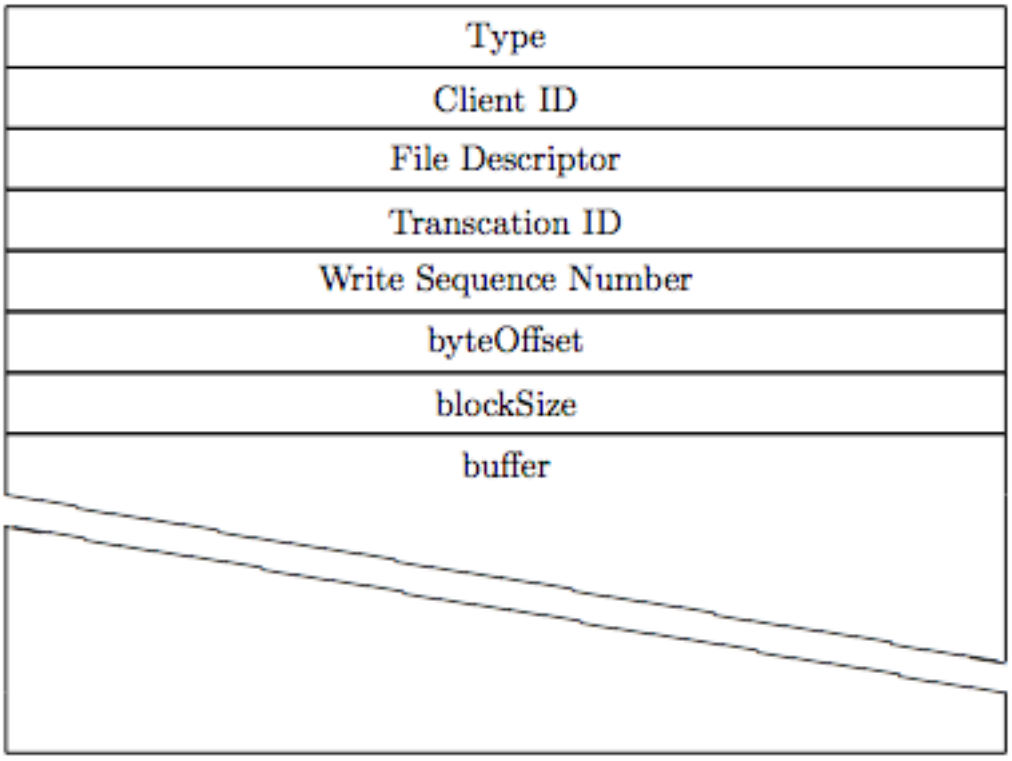

Figure 5: writeblock Packet

buffer has up to 512 bytes. Its length is saved in blockSize field.

check packet:

Type
Client ID
File Descriptor
Transcation ID
Write Sequence Number

Figure 6: check Packet

Write Sequence Number is the last writeblock packet's one.

vote packet:

Type
Client ID
Server ID
File Descriptor
Vote Yes/No

Figure 7: vote Packet

ResendRequest packet:

Type
Client ID
Server ID
File Descriptor
Write vector

Figure 8: ResendRequest Packet

Write vector is a bit vector, with the last 128 bits indicating whether the relevant sequence number's write is received.

commit packet:

Type
Client ID
File Descriptor
Transcation ID
CLOSE

Figure 9: commit Packet

CLOSE Flag is set to True if close() is called. In commit(), it's False. If CLOSE = True, server will close the file after commit.

commitACK packet:

Type
Client ID
Server ID
File Descriptor
Transcation ID

Figure 10: commitACK Packet

abort packet:

Type
Client ID
File Descriptor
Transcation ID

Figure 11: abort Packet

abortACK packet:

Type
Client ID
Server ID
File Descriptor
Transcation ID

Figure 12: abortACK Packet

TranscationRequery packet:

Type
Client ID
Server ID
File Descriptor
Transcation ID

Figure 13: TranscationRequery Packet

TranscationResponse packet:

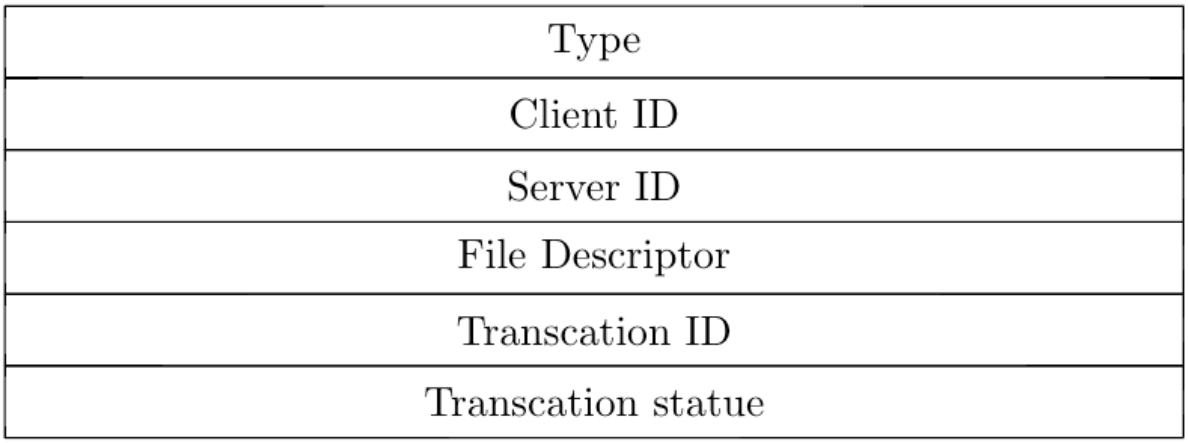

Figure 14: TranscationResponse Packet

Server ID: This Server ID is the respond server's ID.
Transaction statue: 0 if transaction is neither committed nor aborted, 1 if it is committed, 2 if aborted.

2. Event sequence

InitReplFs():

Once a client is started, this function is called by application to establish connection between this client and all servers.
First, the client will send an *init* packet in every 100ms for 3 seconds and wait for respond. All server received this packet at the first time should reply with *initACK* packet with their ID. The 30 times resending mechanism guarantees all available server should receive the *init* packet at least once, therefore the client can record all available server down. If there is no available server, the client will just return -1 and then quit execution. Otherwise, it will return 0 to serve application. The client will not be blocked by any unavailable server except all (lead to quit).
All other functions called before InitReplFs() will be regarded as invalid and directly return -1.
The InitReplFs() get called at second time or more has no effect.

Openfile():

After client called InitReplFs() or CloseFile(), it can call Openfile() to open a new file.
The client will send an *open* packet to every server. Like the init packet, it keeps resending in 3 seconds and wait for respond.
Each server received it first time will check whether it is already opened by other client. If so, it will respond *openACK* with SUCCESS flag set to False. Otherwise, it will respond with SUCCESS set to True, while using the file name for maintaining its own information, and file descriptor for later communication.
If the client receives any *openACK* packet with SUCCESS=False, it just returns -1 indicating other client is opening this file.

WriteBlock():

With a file opened in the client, it can call WriteBlock to send written data to servers. This call send write packet to all servers and wait for no response. It still uses the resend mechanism explained previously (To reduce network overhead, maybe it can be tune to resend for just 5 times for now). Also, the frequency of WriteBlock is much higher compare to all other function calls. So it shouldn't be blocking while retransmitting. The client will just check whether this write is valid and return. If it's valid, the write packet will be trigger by timer to retransmit.
I also considered to cache all write data in client and send when commit. Since it might cause some network congestion, caching may result to a long time commit and doesn't seem to be a good choice.

Commit():

Commit implements 2PC. The first phase, the client sends out *CheckCommit* packet to all servers. This packet also includes the resending mechanism and wait for all respond.
The server can reply with *voteYes or ResendRequest* packet. *vote* packet with vote=yes means this server has all write in this transaction and it's ready to commit. While vote=no means this server also sees all write packets, but it's not valid due to disk/memory limitation.
ResendRequest means this server has lost some *write* packets and the vector indicates those ones missing. The client will resend all missing packet, and then come back to wait for *vote*. If

the client didn't receive *vote* when time out, it will abort this commit and return -1. If any *vote* packet has vote=no, it also aborts. If all vote=Yes, the client sends a *commit* packet with resend, waiting for every server reply with *commitACK* packet.
Server might become unavailable when the client is waiting for *commitACK* packet. The the client won't get a *commitACK* from it. However, it doesn't result in inconsistency because the client will regard that server as dead.

Abort():

If the client failed to commit and return -1, or the application want to give up all writes, the Abort() is called. Then the client will send *abort* packet to all server and wait for *abortACK* packet from all servers. Any server not responding will be regarded as dead.

Close():

The Close() function in my design is just the same procedure as the last step in Commit() – send *commit* packet and wait for *commitACK*, with the CLOSE flag sets to true.

Packet Loss and out of order: (transaction number and write number)

In the network, packet loss and delay is inevitable. I use retransmission to prevent data loss from packet loss. And each commit also confirms all *write* packet is present in all servers.
For packet delay, some *write* packet in previous transaction might arrived after the new transaction has begun. With the Transaction ID field, the server can directly discard it. All other calls like InitReplFs, OpenFile, Commit, Close are blocking so will not be affected.

Handle client unavailable:

If the client got killed, there have multiple condition to tackle.
If the client isn't in the middle of Commit()/Abort(), then there might be some uncommitted writes which should be abort, also all the servers should be able to relinquish this file's access to all other clients.
The client gets killed in the middle of commit/abort. It just sent *commit (Last step of commit call) /abort* packet to partial servers. The missing server would think it's dead and abort all writes, which would result in inconsistency in commit case.
To handle this case, server got *CheckCommit* packet but missing the commit or abort packet for 1 minutes will broadcast the *TranscationRequery* packet to all other servers. If any server respond TranscationResponse with commit or abort in transcation statue filed, it means the previous transcation is alrealy commited/aborted so this server will also follow.

3. Evaluation:

In this implementation, we simply use UDP to broadcast every packet instead of establishing TCP connection between each client and server, which doesn't scale because it's quadratic increase. TCP provides packet retransmission in transport level, but it lacks for application state, which will be explained below. So it's also not a optimization in performance on both server and client sides.

UDP pro:

UDP is lightweight and broadcast is desirable in most cases for replicated file system. So in network view, it's efficiency.
In contrast, using TCP, it means every client need to establish connection to server, which is a 1 to N mapping, so every time it broadcast a packet, there are N numbers of IO outcome. It burdens bandwidth and router's packet buffer, which lead to more congestion.

Implementing retransmission in application level rather than transport level is also an advantage to performance. If use TCP, how many connections should we establish for each pair of server and client? If there is only one, then it's blocking. All events are transmitted by stream. So it can't decide whether one complete packet has arrived unless the stream send all packets. If there is some packet loss in the middle, the server had to wait for it to start processing. This will block other events. If every event establishes one TCP connection, then the three-time handshake is a big overhead because there are so many events. Therefore, lack of application level information make TCP have some unnecessary dependence, which is similar to the limitation of CATOCS.

In summary, in both network and end point view, TCP is less efficient than UDP.

UDP cons:

UDP is just connectionless. So packets may arrive totally out of order, and also there is not guarantee packet will arrive. To handle those conditions, it makes application more complicate.

If some server/client just be killed silently, others has no way to be informed unless they found no response on later events. So every time an Open(), Commit(), Abort() or Close() happens, the client has to wait enough long time to make sure all available server get the packet. And server needs more complicated mechanism to prevent inconsistency.

In conclusion, UDP complicates the application and makes events processed slower.

4. Future Directions:

Server restart: Some server just dead. To large practical system, this is common so should be properly handle to restart them. The late join server should learn information from other server.

2PC network overhead, change to sharding/hierarchy: If there are too many servers/clients, 2PC doesn't scale well. Each commit may meet some server's problem like out of memory. Make application has very bad performance. To scale, hash file descriptor and sharding to some part of server, or make a master to controller metadata are the alternatives.

Allow multiple client simultaneously open one file: In practical, concurrent opening same file is desirable for multiple user. Using the timestamp of commit transaction is a way to resolve conflict.

Allow Multi-Thread: Both Server and Client are implemented as blocking, i.e. when it processing some packet, there packets are blocked. Next step is to spawn a thread to process the arrived one when it get a packet, and the main thread is still waiting for new packet.