Ryan Oman
Brad Bicknell
Abhishek Choudhary

CSE 490h – Distributed Systems – Project 3

Implementation Details

We implemented Optimistic Concurrency Control for Project 3. Our system has one master node (the node with address 0) and many slave nodes (nodes with addresses greater than 0). The state machines for the client and server are included in the project directory in pdf form. The nodes use the following protocols, as defined by the TXNProtocol class, to communicate with each other:

- WQ – Write Query, WF – Write Forward, CREATE
  - Payload structure: "[filename]"
- WD – Write Data
  - Payload structure: "[filename] [version] [contents]"
  - Payload structure: "[filename]"
- COMMIT_DATA
  - Payload structure: "[filename] [command type] [contents]
  - Command type is defined in Command class as static fields. The packet only has contents if it is the command type: APPEND, PUT, or UPDATE.
- COMMIT
  - Payload structure: "[number of COMMIT_DATA pkts sent]"
- START, ABORT, HB – Heartbeat
  - Empty Payload
- ERROR
  - Payload structure: "[filename] [error code]" or " [error message]"

Slave Node Logic

When slave nodes want to perform operations on files, they have to first start a transaction with the command 'txstart'. This command then sends a request to the master node letting it know that the slave wants to start a transaction. This uses the TXNProtocol.START protocol. While the transaction is starting, no other commands may be executed on the slave side. When the master node receives the start request from the slave node, it checks to see if there are any transactions currently waiting to commit that are dependent on the node requesting to start a transaction. If there are some, this means that the node trying to start a transaction had crashed, and the transactions dependent on it must abort. Once it finishes notifying any dependent transactions, the master node sends a start confirmation back to the slave node. This also uses the TXNProtocol.START protocol. Once a slave node has executed all the commands it wants to, it can execute 'txcommit', which commits the transaction to the server, and ends the transaction. Once the transaction has been committed, a new transaction must be started with 'txstart'.

Once a slave node has received the start confirmation from the master node, the user may begin executing file operations/commands. Here is a list of the procedures for each command execution:

- Create: If the File object in memory is invalidated (File.getState() == File.INV) then a create request is sent to the master node. Otherwise an error message is printed out. Once the master node returns a write data (TXNProtocol.WD), the file is created on disk locally, and the File object in memory's state is set to File.RW. The create command is also added to the transaction log in memory (TransactionLayer.txn).
- Append/Put: If the File doesn't exist locally, a write query (TXNProtocol.WQ) is sent to the master node. Once the slave node receives the write data back from the server, it updates the

file on disk, the file version in memory, and adds the write command to the transaction log in memory (TransactionLayer.txn) -> the write is not written to disk, no success message is printed.

- Get: If the File exists locally, the command is added to the transaction log in memory, otherwise a write query is sent to the master node in a similar fashion to the append and put commands.
- Delete: If the File doesn't exist locally, a write query is sent to the master node. Once the slave node receives a write data back from the server, it updates the file on disk, and then adds the delete to the transaction log in memory.
- Txabort: Send an abort request (TXNProtocol.ABORT) to the master node. Once an abort confirmation is received, reset the transaction and print out a success message.
- Txcommit: Send each command executed in the transaction log in memory to the master node, one per packet (TXNProtocol.COMMIT_DATA). Send a commit packet (TXNProtocol.COMMIT), which has the number of commands sent in step 1 of the commit in its payload. Once it receives a commit confirmation (TXNProtocol.COMMIT) back from the server, the transaction is pushed to disk, the results of executing each command are printed out, and then the cache is cleared.

Multiple slave nodes can have read/write copies of a single file out at the same time. Any possible conflicts are resolved at on a commit at the master node. To do this, the slave node keeps track of what version of a given file it starts with, and all commands executed on said file. The commands executed on a file are stored in memory, and not actually executed on disk until after the transaction commits.

Slave nodes receive write forwards (TXNProtocol.WF) when the server wants the most up to date version of a file that a node has. The slave node builds its most recent version of the file through the following steps:

1. Load the file from disk into memory (the state the slave node received it in from the master node).
2. For each write in the transaction log, the returned version number is incremented, and the write is executed on the file contents in memory.

Once the most recent version of the file has been constructed, the slave node returns the file as a write data to the master node. If this message happens to throw a timeout error, the client node does an infinite retry, since the master node has to wait until the slave node returns something for it to make progress (if the ACK message was already received by the server).

When a slave node receives an error message (TXNProtocol.ERROR), the error is simply printed out, and the command that caused it is not added to the transaction log as having executed.

When a slave node receives a heartbeat message (TXNProtocol.HB), it returns an ACK message and does nothing else.

Master Node Logic

When a master node starts, it looks for a file with the name '.l'. In this file is the list of all files in the local directory. The cache on the server is initialized using this directory file. If the server just crashed and is now recovering, all the files in the cache will still be there, but any transaction that tries to commit, with be told to abort, since the server thinks that no one has a copy of any of the files (that is stored in memory).

When the master node receives a write query, it first checks to see if the file exists. If it doesn't exist, error code 10 is returned to the requester. If there aren't any nodes that have the file checked out at that time, the master node just returns the last committed version of the file that is stored on the master node in a write data message along with the last committed version. Then it determines if there is already a write query being executed on the same file. If there is, the write query is queued and must wait for the prior write query to finish. Otherwise, it must then send write forwards to all nodes that

have copies of the file checked out. This starts the proposal system. Essentially, each node returns a write data that has its most up to date contents and highest version number. The server also puts the last committed version and contents in as a proposal. The proposal with the highest version number is the one that gets chosen to send back to the slave node as a write data that requested the file. The master node also sends any queued requesters a write data back. The master node saves the version that each requester got sent and what node that version came from so that when the requester tries to commit, the master node will know that each requester's transaction is dependent on the transaction that it got its version from.

When the master node receives a write data, it knows that this is in response to a write query that was sent out. This is referred to as a proposal in the above paragraph. If all nodes have returned their proposals, then the best proposal can be chosen as described above; however, if there are still nodes that have not timed out or proposed a file version, then the master node waits for them.

When the master node receives an error packet, it knows that this is in response to a write query that was sent out. In this case, it means that this slave node doesn't actually have a copy of the file, and the server should update its logging appropriately to reflect that. As above, the server then either waits for more proposals, or sends back the best one to the requester.

When the master node gets a timeout on a write forward, it doesn't wait for that node to send a proposal any longer, and either waits for other nodes to send their proposals or returns the best received proposal to the requester.

When the master node receives a create request, it creates a file object in its local cache in memory, but does not write anything to disk. However, if the file already exists in its local cache, it returns the appropriate error message.

When the master node receives an abort request, it updates the cache in memory to reflect the fact that the abort requester no longer has any files checked out. It also updates the file cache so that when other transactions that may have been dependent on this one try to commit, they will be aborted. The master also checks for any transactions waiting for this one to commit and lets them know to abort also.

When a master node receives a commit data message (TXNProtocol.COMMIT_DATA), it saves the command in memory locally, and waits for the actual commit request (TXNProtocol.COMMIT). Once it receives the commit message, it checks to see that it received as many commit data messages as the commit message says it should have. If some of the packets got dropped, then the server returns an error message, asking for the user to try again. Otherwise, the server tries to determine whether or not the requested transaction can commit or not. If a transaction cannot commit, it is either because it must wait for another transaction that it is dependent on to commit or that it must abort because it is dependent on another transaction that conflicts with this one. A transaction is dependent on another transaction if the file it received came from a different transaction that has not committed yet. There are several cases that could cause the dependent transaction to abort as a result of the transaction it got the file from:

- The transaction the requester got the file from aborts
- The transaction the requester got the file from writes to the file after the requester received the file, and the requester also writes to the file.

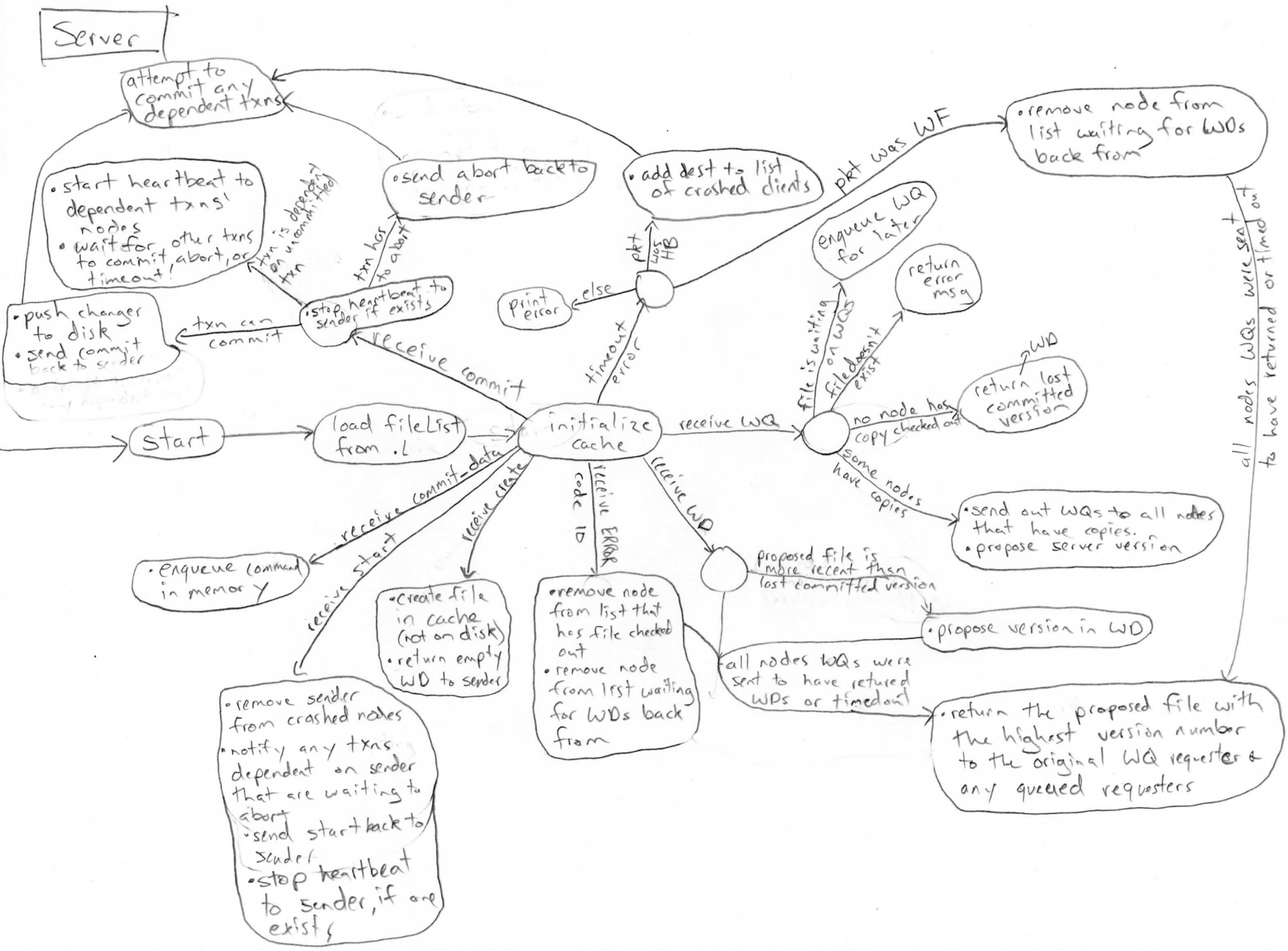
There is also the case of two transactions starting with the same version of the file, and both writing to it. In this case, whichever transaction commits first will be able to and the other one will have to abort and retry. If the transaction trying to commit is dependent on another transaction that has not committed or aborted yet, the master node starts a heartbeat protocol with the node that that transaction is on. The heartbeat protocol sends a heartbeat message (TXNProtocol.HB) to the slave node the master is waiting for every time the master receives an ACK back from a prior heartbeat message to

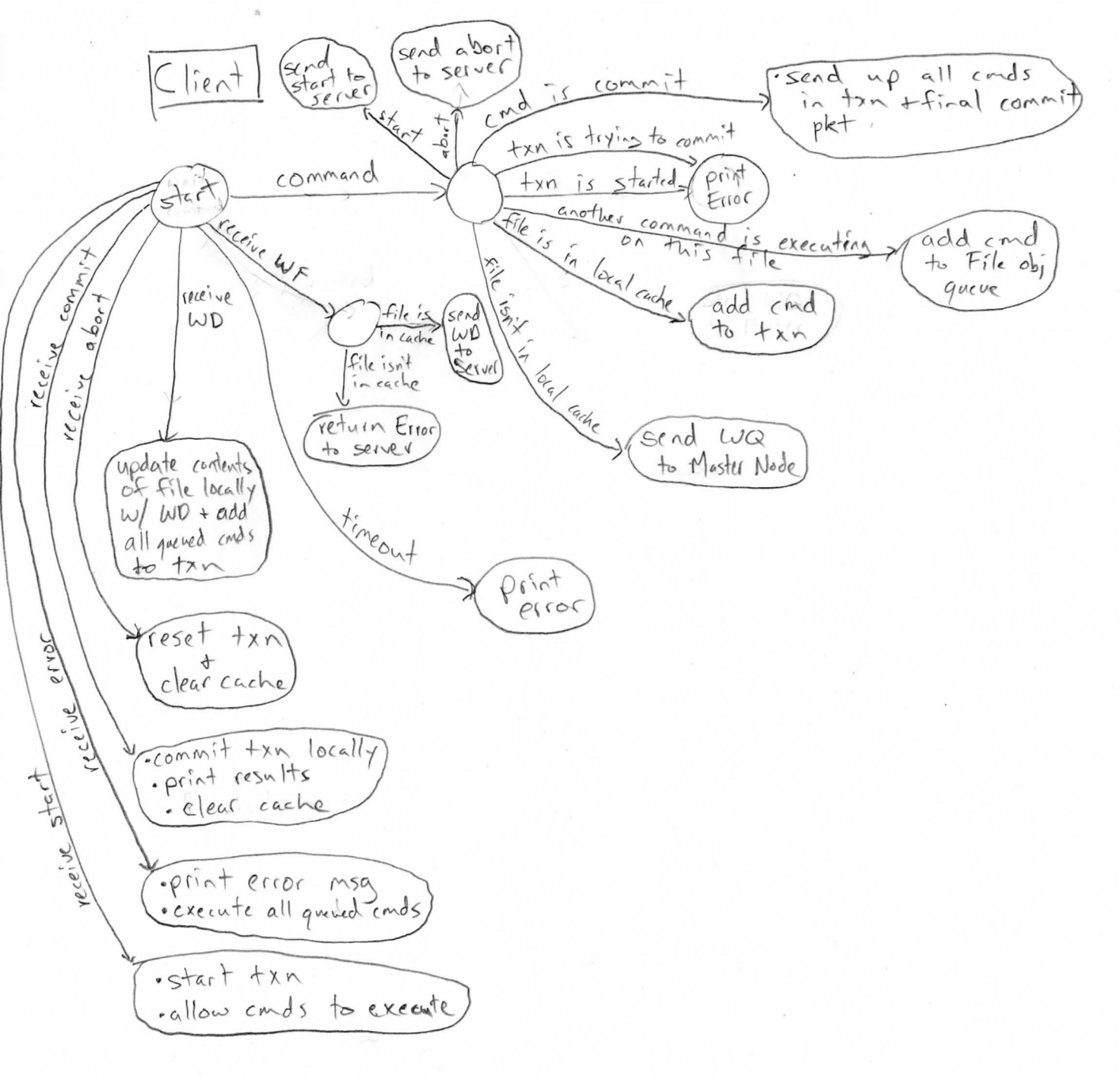
that slave node. If this process ever times out, the master node assumes that the slave node it is waiting for has crashed, and notifies any dependent transactions attempting to commit to abort.

If none of these problems are present, then the requested commit may proceed. The master node pushes all the changes to disk for the transaction. Once the changes have been pushed to disk, the master node sends a commit confirmation message to the slave node (TXNProtocol.COMMIT). After it executes the requested transaction, it also attempts to execute any transactions that are dependent on the transaction just committed.

Assumptions/Known Issues

- Slave nodes perform infinite retries when sending error or write data messages, since the RIO layer only allows one way data streaming, and thus, each direction on the between the master and slave nodes are on different sessions. This means that if the master node receives an ACK from the client for a write query, it won't retry or timeout after that.
- Server failures aren't handled in every case. If a server crashes in the middle of pushing a commit to disk (for commits that update multiple files), it could leave the overall file system in an inconsistent state.
- Assume that all slave nodes will attempt to commit, abort, or crash at some point. Transactions dependent on unfinished transactions on nodes still responding to heartbeats will wait until the transaction it is dependent on either commits, aborts, or the node doesn't respond to a heartbeat.


Server
start
load fileList from .L
initialize cache
receive commit
receive WQ
receive commit_data
receive create
receive start
receive ERROR code ID
receive WD
timeout error
prt was HB
prt was WF
else
print error
txn is dependent on uncommitted txn
txn has to abort
txn can commit
file is waiting on WQs
file doesn't exist
no node has copy checked out
some nodes have copies
all nodes WQs were sent to have returned or timed out
attempt to commit any dependent txns
start heartbeat to dependent txns' nodes
wait for other txns to commit, abort, or timeout
send abort back to sender
add dest to list of crashed clients
stop heartbeat to sender if exists
push changes to disk
send commit back to sender
remove node from list waiting for WDs back from
enqueue WQ for later
return error msg
return last committed version of WD
send out WQs to all nodes that have copies
propose server version
enqueue command in memory
create file in cache (not on disk)
return empty WD to sender
remove node from list that has file checked out
remove node from list waiting for WDs back from
proposed file is more recent than last committed version
propose version in WD
all nodes WQs were sent to have returned WDs or timed out
return the proposed file with the highest version number to the original WQ requester & any queued requesters
remove sender from crashed nodes
notify any txns dependent on sender that are waiting to abort
send start back to sender
stop heartbeat to sender, if one exists


Client
send start to server
send abort to server
start
abort
command
cmd is commit
send up all cmds in txn + final commit pkt
txn is trying to commit
txn is started
print Error
another command is executing on this file
add cmd to File obj queue
file is in local cache
add cmd to txn
file isn't in local cache
Send WQ to Master Node
receive WF
file is in cache
send WD to server
file isn't in cache
return Error to server
receive WD
update contents of file locally w/ WD + add all queued cmds to txn
receive commit
receive abort
receive error
receive start
timeout
print error
reset txn + clear cache
•commit txn locally
•print results
•clear cache
•print error msg
•execute all queued cmds
•start txn
•allow cmds to execute