Ryan Oman
Brad Bicknell
Abhishek Choudhary

CSE 490h – Project 4 – PAXOS

We made several changes to our existing transaction and RIO systems in addition to implementing the PAXOS algorithm for committing changes for this project. Changes to the existing functionality provided by project 3 are outlined in the **Fixes of Known Issues** section. The addition of the PAXOS algorithm is outlined by the **PAXOS** section. System file formats and their purposes are described in the **Fixes of Known Issues**, **PAXOS**, and **System Files** sections. How timeouts in the transaction layer are handled is outlined in the **Transaction Layer Timeouts** section. State diagrams of our implementation of the PAXOS algorithm are at the end of this writeup.

Fixes of Known Issues

We made a change to the RIO layer that stores session IDs on disk. We realized that not doing this did not guarantee at-most-once semantics after reading the feedback from project 1. These are stored in the ".sessions" file on disk. We also fixed part of our handshake in the RIO layer for establishing sessions. Prior to this fix, timeouts were causing packets to become out of order and not be delivered. When a timeout occurs, that packet is returned to the transaction layer and all other waiting packets are loaded into the queue for waiting for establishing sessions. A session is then established with the server. The server returns a new session ID, which we weren't doing before. This prevents any delayed packets from being delivered on the server side from the old session. Once the new session is established, the commands in the queue are resent and their retry values are reset.

We also fixed some known issues in our transaction layer for this project. We were not handling the case if a server crashed in the middle of writing a commit to disk. Now we use the file ".wh_log", and create a redo/write-ahead log for all updates that will be performed on the files that the transaction uses. If this file exists on startup, we execute the file updates in it. The updates contained in the file are not every command in the transaction, but rather the final state of each file used by the transaction at the time the transaction commits.

Previously, we used the ".l" file to store filenames present on the system on the server. We were not storing the file version though, which presented a problem if the server crashed, restarted, and then received a commit which had dependencies on a previously committed version of the file. We added the last committed file version to this file and the last committer, so that when the server starts up, it loads each filename and its last committed version and committer.

One other previous problem mentioned in project 3's write-up is that there weren't timeouts in the transaction layer. So after a client's commit was ACKed, it would have no way of recovering if the server crashed except by restarting. For this project, timeouts are implemented by the class TimeoutManager in the transaction layer. It creates timeouts similar to the RIO layer, and stores the last sequence numbers sent for each node it communicates with. For clients, timeouts are created for the following TXNProtocols: WQ, CREATE, ABORT, COMMIT, START.

Ryan Oman
Brad Bicknell
Abhishek Choudhary

For servers, timeouts are created for the following TXNProtocols: WF. How each timeout is handled is described more in the timeouts section.

We were not using transaction ids to identify transactions in the previous project, which presents a problem if a client tries to commit a transaction twice. This also complicates things when determining whether not tx2 can commit if it is dependent on tx1, and whether or not tx1 was the last committed transaction for the dependent file. So we changed the identifier for a transaction to be a unique transaction id as opposed to the committer's address. Each transaction id starts out as a given node's address, and then is incremented by RIONode.NUM_NODES. This way all transaction ids are unique and the node's address can be pulled from the transaction id by modding by the number of nodes. Each client writes their last committed transaction id to disk (stored on clients in file ".txn_id"), so that if they crash and then they start up, they will start with a new, unique transaction id. The server takes advantage of using transaction ids by storing all transaction ids and whether they committed or aborted. This is also stored on disk, in file ".txn_log".

Another problem fixed for this project relates to tracking dependencies of a transaction. Previously, where a client got a file from was stored in memory on the server, so if the server crashed, the server would have the client abort in the best case, or allow an inconsistent commit in the worst case. This was fixed by storing where a client got a file from on the client side. When a client gets a write data back from the server, the packet contains what transaction the file came from (or 0 if it was from the server/last committed version). The client stores this and passes it to the server when it commits.

We changed much of how the commit process works in addition to adding PAXOS. Previously we were sending every commit as a sequence of packets, but had no way of recovering if either node crashed during this process. If the client crashed and came back up, then tried to commit, the new commit packets would be included with the old ones, which is another reason we changed to transaction ids. We changed this process to sending only one commit packet, which contained all the commands in the transaction. We are assuming that all transactions fit into one packet for simplifying purposes. We use the CommitPacket for this purpose, which is wrapped by a TXNPacket before being handed off to the RIO layer. The CommitPacket builds a string representation of the transaction by using the buildCommit function in the Transaction class. Contents of put and append commands are transformed into byte arrays before being put in the transaction string to prevent conflicts with using delimiters in the transaction string. For the PAXOS layer, a slightly different representation of the transaction was required, so we created the toString and fromString methods. We had to do this because we couldn't wrap the CommitPacket in a PaxosPacket. This is because the payload limits on the CommitPacket are bound by the TXNPacket, which the PaxosPacket is wrapped in.

One additional thing we failed to mention in our previous write-ups is that you can send content to the put and append commands with whitespace in them, but then the contents must be surrounded by double quotes.

In addition to the many changes described above, we also made several other changes to simplify our code in the transaction layer. Other than these changes, and the addition to the PAXOS algorithm, the functionality is similar to what was described in the project 3 write-up.

PAXOS

Our implementation of PAXOS uses a leader node, which is both the distinguished learner and distinguished proposer. It is not necessary for correctness for there to be a leader; however it makes progress much more likely. Storage state is maintained through the PAXOS algorithm. The value associated with a proposal is actually a string representation of the entire transaction. Therefore, when a server learns about a chosen value, it can then execute it immediately also.

Leader Election

Each client elects his own leader on three events: node startup, timeout with current leader, and on commit/abort. The leader is elected by the client asking all server nodes what their highest known instance is. The client waits for a majority of responses and then picks the server with the highest known instance number and the lowest address. This guarantees that the leader that the client will send its commit to will know about the most recent instance that has been accepted by a majority of acceptors, meaning the leader doesn't have to ask all the other nodes for what the highest instance they know about in order to proceed.

The Proposer Layer

The proposer layer is in charge of initiating an instance of PAXOS when it receives a commit and trying to get that commit learned. The first step in this process is to send prepare requests to acceptors to learn what state the acceptors are in, and based on what it hears from the acceptors, it will make a proposition that has a chance of succeeding. It only allows one instance of PAXOS to run at any given time.

For each commit the Proposer Layer receives, it has to start a new instance of PAXOS. In this process, the proposer has to make sure that it is aware of the consensus decision made for every earlier decision, because holes in the log could lead to inconsistency. Because of this, every time a new instance of PAXOS begins the proposer asks the Learner Layer for which instances it does not have a value. Then, for those instances it sends out 'Recovery' packets (RECOVERY protocol) to the acceptors who respond with what they have accepted or what they have chosen. If PAXOS receives a RECOVERY_CHOSEN packet for a given instance it is trying to recover, it accepts that as the value. Otherwise, it waits for a majority of the acceptors to respond with a RECOVERY_ACCEPTED and the same accepted value and uses that. If it is determined that a particular instance hasn't finished, then it finishes that. We send PREPARE packets out and wait for this instance is finished. During this process, the Proposer is not sending doing prepares or proposals that deal with future commits. It is just working on old instances and learning what the group did, or helping the group finish.

Once the recovery stage completes, the Proposer sends out it's PREPARE packets to which acceptors respond with an REJECT or PROMISE and the highest proposal number and value pair they have scene thus far. If the majority reject the prepare proposal number, the proposer sends out new prepares with higher proposal numbers in hopes that this time the majority will promise. If the majority does PROMISE, the proposer sends out PROPOSE packets that come with a value (this is can be the value of the original commit that started this instance, or in the case that there is another proposer and some acceptors are already trying to learn a value for this instance, it can be that value). If the proposer finds that it takes more than 13 steps to find a majority for a given instance, it times out and starts the process over again.

When the learner has learned something for an instance, it informs the proposer layer that the instance is finished and gives it the value that it learned. If this value is the same as the commit that proposer was initially using, it removes it from the commit queue and goes on to the next commit it had received. If there are no commits, it waits until there is a commit. If that value was different, it does not remove that commit from the queue and starts a new instance with that commit as the value. This is all the proposer layer handles, as everything else is handled by the other layers as responses to whatever the proposer does.

Proposer Layer Protocols

- PAXOS PREPARE – Contains an instance number and proposal number and no value
- PAXOS PROPOSE – Contains an instance number, proposal number and a value
- PAXOS PROMISE – Contains an instance number and proposal number
- PAXOS REJECT – Contains an instance number
- PAXOS RECOVERY_CHOSEN – Contains an instance number and proposal number and value
- PAXOS RECOVERY_ACCEPTED - Contains an instance number and proposal number an 'accepted' value for that instance.
- PAXOS RECOVERY_REJECTED - Contains an instance number and no value since nothing was learned for that instance

Proposer Layer Log Files

- ".prepare_version" – each time the proposal number is incremented, the curVersion is also incremented, so that and delayed packets from prior broadcasts of prepares (rejects or promises) don't factor into a new broadcast of prepares when determining whether or not a majority has returned. Has following format: "[curVersion]".

The Learner Layer

The learner layer contains functionality for both the distinguished learner (leader) and other learners. Upon receiving an accept message the leader increments a running count for the instance number and proposal number combination in the message. If this count is equal to or greater than the majority of the acceptors, the leader will learn the proposal. A leader learns a proposal by writing the value of the proposal to disk using the transaction layer, as well as

adding the proposal to a list of learned proposals. Once the leader has learned the proposal and written its learned proposals to disk, the learner sends learn messages to every other learning node and the client node that committed. When a learner receives a learn message, it will learn the proposal if the proposal has not already been learned.

The learner layer keeps track of holes in proposal instance numbers. For example if a learner has received proposals with instance numbers 1 - 10 and 12 - 15. The learner will not learn proposals 12 to 15 until it has learned 11. To account for node crashes, out of order proposals are written to an "out of order" log on disk so they can be recovered. The learner layer also responds to the acceptor layer during recovery, by reporting learned proposals for the given instance number from the acceptor.

Learner Layer Protocols

- PAXOS ACCEPT - contains the instance number, proposal number, and value for the accepted proposal. Sent from an acceptor to the distinguished learner.
- PAXOS LEARN - contains the instance number, proposal, number and value for the learned proposal. Sent from the distinguished learner to all other learners.

Learner Layer Log Files

- ".learned" - contains every proposal (instance number, proposal number, and value) that have been learned by the learner layer. The log file is replaced by the learned proposals in memory every time a new proposal is learned. The log file is read when a node recovers from a crash. This allows the learner to keep state across failure and know about all chosen transactions that it has learned about when it starts up. Each line has the following format: “[instanceNum]|[proposalNum]|[toString of TXN]”
- ".outOfOrder" - contains every proposal that has been accepted by a learner, but not learned because the proposal instance numbers are out of order. The log contains the instance number, proposal, number and value for every out of order proposal. It has the same format as the “.learned” file.

Acceptor Layer

The Acceptor Layer is in charge of responding to messages sent by the proposal layer to help the group come to a consensus. An acceptor can receive prepare messages, propose messages, and recovery messages.

Upon receiving a prepare message, the acceptor will respond with a promise message or a reject message. If the acceptor can promise not to accept a smaller value for the instance given in the message, then the acceptor responds with a promise message to the leader. If the acceptor has already accepted a proposal for this instance, or the proposal number for the instance is less than what has been promised by the acceptor, a reject message is sent to the leader. Before sending promise messages, the acceptor writes all of its promises to a log file on disk.

After receiving a proposal message, the acceptor will either respond to the leader with a reject message or an accept message. It will accept it if the proposal number is not below a proposal number it promised to pick above. If the proposal number is too low, it rejects.

An acceptor layer receives a recovery message when there is a proposer layer starting a new instance and filling holes. The recovery message that it receives comes with an instance number that the proposer layer is trying to learn the value for. If the layer has 'Accepted' a value for that instance, it responds with a RECOVERY_ACCEPTED to the proposer. If it has learned a value for that instance, then it responds with a RECOVERY_CHOSEN message. This does not have to do with acceptor layer functionality, it is just information that the proposer uses.

Acceptor Layer Protocols

- PAXOS PROMISE – contains the instance number and promised proposal number. These messages are sent from the acceptor to the distinguished proposer.
- PAXOS ACCEPT – contains the instance number, proposal number, and proposal value of the accepted proposal. These messages are sent from the acceptor to the distinguished learner.
- PAXOS RECOVERY_ACCEPTED – contains the instance number, proposal number, and proposal value of the accepted proposal. These messages are sent from the acceptor to the distinguish proposer after receiving a PAXOS RECOVERY message.
- PAXOS RECOVERY_CHOSEN – contains the instance number, proposal number, and proposal value of the learned (chosen) proposal. In this case the acceptor will have asked the learner layer for a learned proposal with a given instance number. These messages are sent from the acceptor to the distinguished proposer after receiving a PAXOS RECOVERY message.

Acceptor Layer Log Files

- ".acceptor_record" – This keeps track of the values that have been accepted for every instance. If a node crashes and then comes back up, it can remember if it accepted something for some instance and not accept anything else from delayed/resent packets. Each line has the following format: "[instanceNum]|[proposalNum]|[value]".
- ".promises" – This keeps track of all promises made by the acceptor so that if the node crashes and comes back up, it can remember all promises it has made and prevent any proposal with a proposal number that is too small from being accepted. Each line has the following format: "[instanceNum] [highestProposalNumPromised]".

Transaction Layer Timeouts

Server Timeouts:

- WF – Write Forward
    - When a write forward times out, the server removes that node from the nodes it is waiting on. Then the server checks to see if it waiting on any other writes forwards. If it isn't, then it will respond to the client who requested the file with

the highest version of the file it knows about. Otherwise it waits for all other write forwards to either return or timeout.

- PAXOS – Election and Prepare
    - When an election times out, the transaction that is trying to start is aborted.
    - If a prepare times out, it resends the prepare messages with whatever the current proposal number and value is. It resends proposals if there is a majority of promises already, since the value and proposal number could have changed if some acceptors responded with reject messages.

Client Timeouts:

- WQ – Write Query, CREATE
    - When a write query times out, an error message is printed out, and the command is not executed. If the server didn't crash, and was just delayed, and a write data gets returned, it is ignored by the client, since it will have the sequence number of the write query that was originally sent, which timed out.
- COMMIT, ABORT
    - When a commit or abort time out, the client prints out an error message saying so. If the server didn't crash but is just taking a long time, and returns a confirmation of a commit or abort, the client still executes the transaction and prints out a success message. -> should change to infinite retries?
- START
    - When a start times out, the client prints out an error message, and user must type the command again.

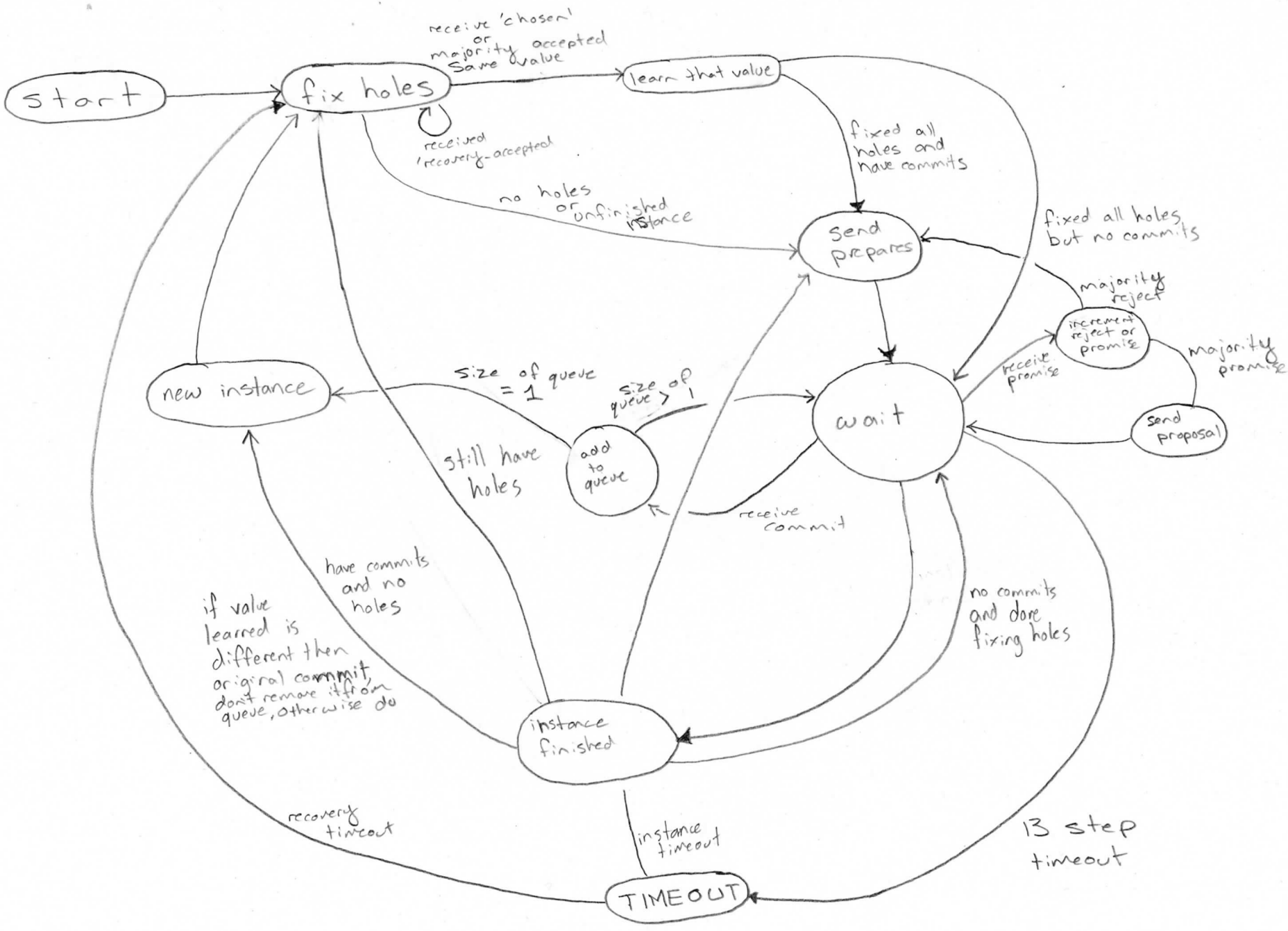
Non-PAXOS System Files

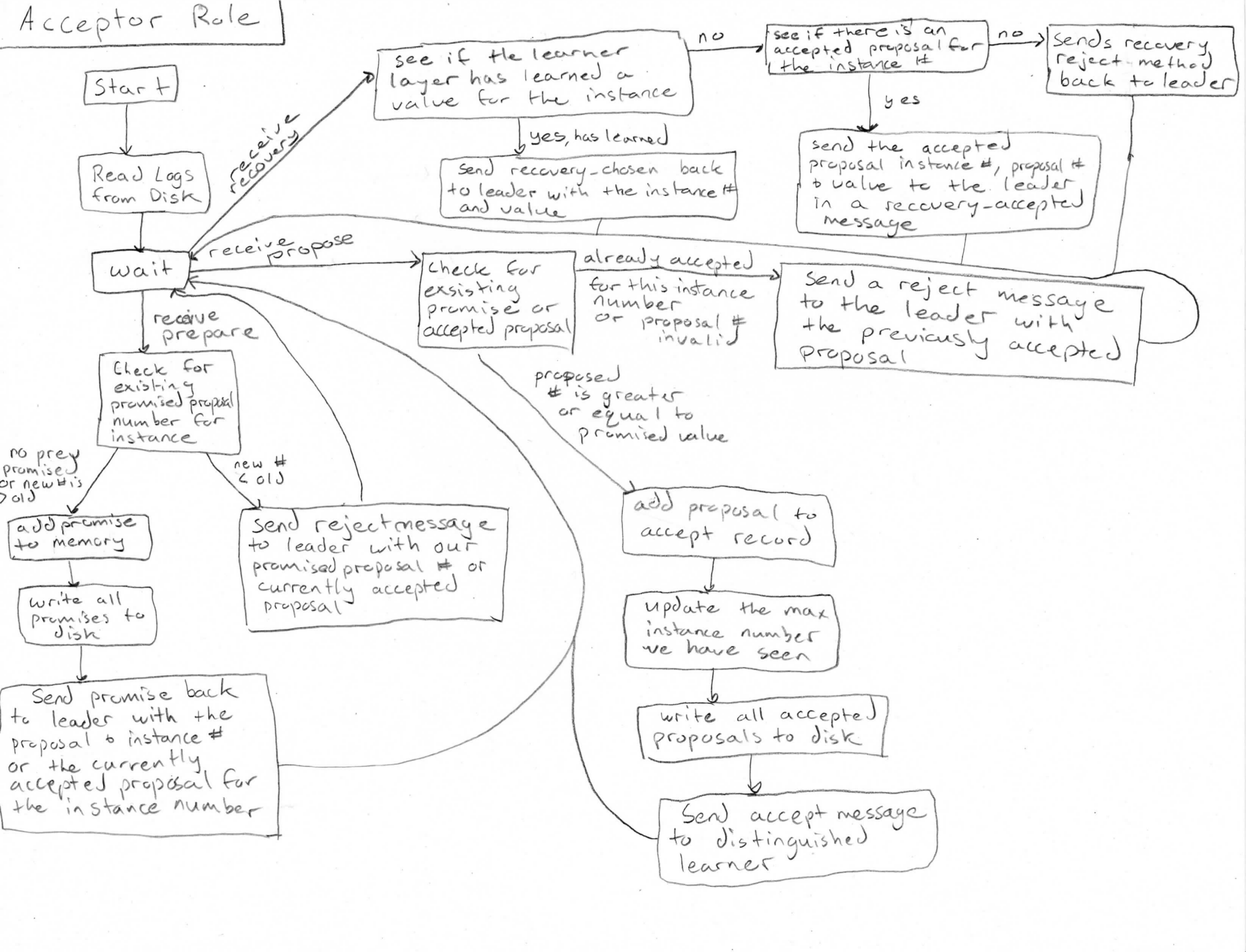
- ".txn_id" – Stored on clients. Has the following format: "[lastCommittedTxnId]"
- ".txn_log" – Stored on servers. Each line has the following format: "[txnID] [1 if committed, 0 if aborted]"
- ".txn_seq" – Stored on all nodes. Each line has the following format: "[nodeAddr] [lastSeqNumSent]". Used for timeouts on the transaction layer.
- ".sessions" – Stored on all nodes. Each line has the following format: "[nodeAddr] [currentSessionID] [lastSeqNumReceived]". Used by the RIO layer to build InChannels on startup.

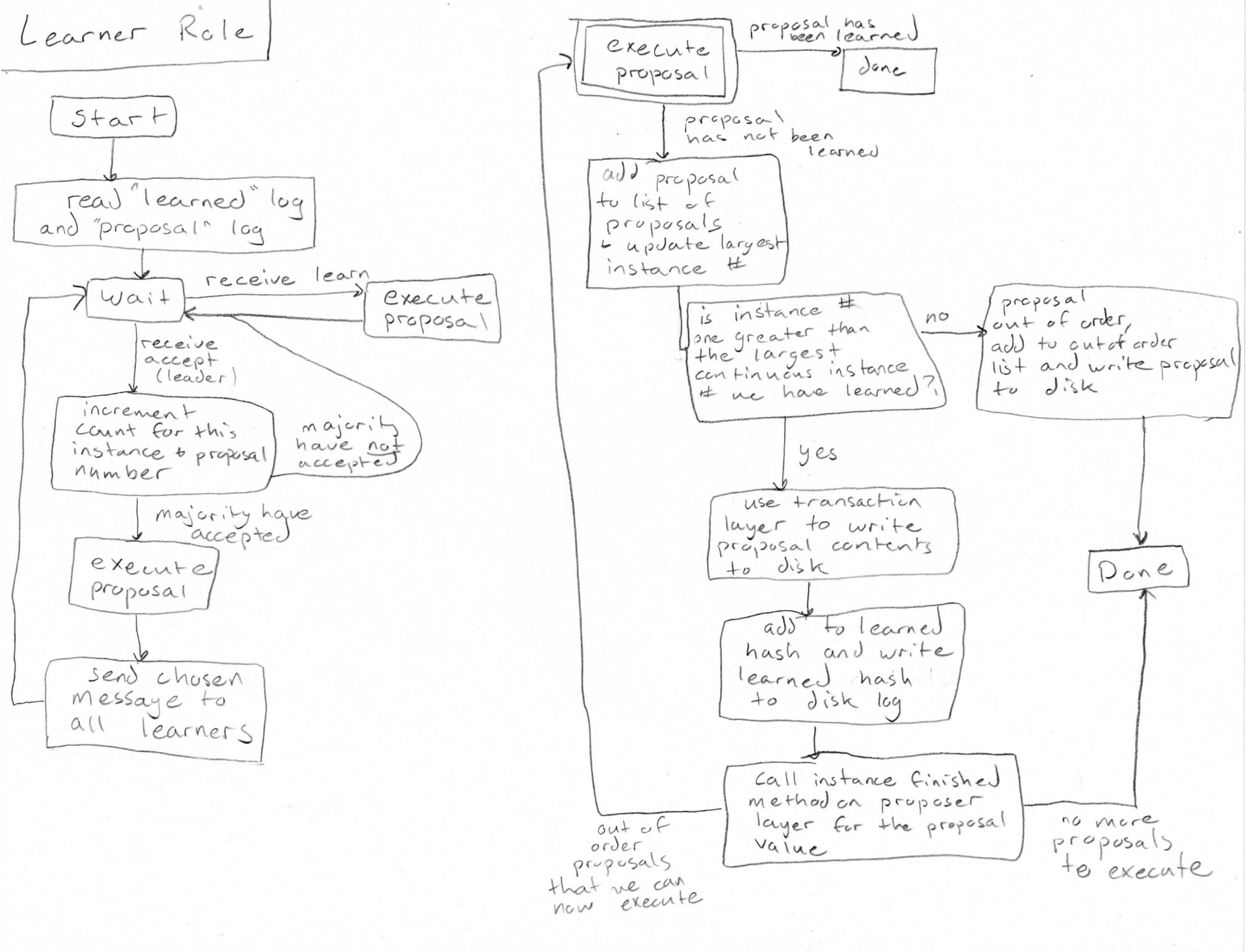
Known Issues and Assumptions

- 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.
- Assume that progress can be made on a commit. Proposers retry infinitely until they learn about their proposal being chosen. Clients also retry infinitely, electing new leaders and resending their commit or abort.
- Known to be unstable with a high failure rate.

- With high packet loss rate, may descend into state where infinitely retries, but never gets anywhere. We didn't have enough time to track down this bug.
- When run multiple times without deleting the "storage" directory, may lead to slightly unusual behavior.
- Assume there is a known bound on the total number of nodes in the system that is specified by RIONode.NUM_NODES
- Assume that our string representation of an entire transaction can fit in the payload of a TXNPacket and a PaxosPacket.


start
fix holes
receive 'chosen' or majority accepted same value
learn that value
received 'recovery-accepted'
no holes or unfinished instance
fixed all holes and have commits
send prepares
fixed all holes but no commits
majority reject
increment reject or promise
receive promise
majority promise
send proposal
wait
new instance
size of queue = 1
size of queue > 1
add to queue
still have holes
receive commit
have commits and no holes
if value learned is different than original commit, don't remove it from queue, otherwise do
no commits and done fixing holes
instance finished
recovery timeout
instance timeout
13 step timeout
TIMEOUT


Acceptor Role
Start
Read Logs from Disk
receive recovery
see if the learner layer has learned a value for the instance
no
see if there is an accepted proposal for the instance #
no
Sends recovery reject method back to leader
yes
yes, has learned
Send recovery_chosen back to leader with the instance # and value
send the accepted proposal instance #, proposal # & value to the leader in a recovery_accepted message
wait
receive propose
Check for existing promise or accepted proposal
already accepted for this instance number or proposal # invalid
Send a reject message to the leader with the previously accepted proposal
receive prepare
Check for existing promised proposal number for instance
proposed # is greater or equal to promised value
no prev promise or new # is > old
new # < old
add promise to memory
Send reject message to leader with our promised proposal # or currently accepted proposal
add proposal to accept record
write all promises to disk
update the max instance number we have seen
Send promise back to leader with the proposal & instance # or the currently accepted proposal for the instance number
write all accepted proposals to disk
Send accept message to distinguished learner


Learner Role
Start
read "learned" log and "proposal" log
Wait
receive learn
execute proposal
receive accept (leader)
increment count for this instance & proposal number
majority have not accepted
majority have accepted
execute proposal
send chosen message to all learners
execute proposal
proposal has been learned
Done
proposal has not been learned
add proposal to list of proposals & update largest instance #
is instance # one greater than the largest continuous instance # we have learned?
no
proposal out of order, add to out of order list and write proposal to disk
yes
use transaction layer to write proposal contents to disk
add to learned hash and write learned hash to disk log
call instance finished method on proposer layer for the proposal value
out of order proposals that we can now execute
Done
no more proposals to execute