**Meeting Notes 05-04-2022**

**Progress:** Read many papers on distributed concurrency testing: model checking, randomized testing and search-based testing and incorporated feedback into background. I realize that the current method of reordering messages is not enough. Based on the papers and the ripple code, this will likely not expose any concurrency bugs. I would like to get your thoughts on whether I should implement some or all of the following features.

1. Using delays, I feel is not sufficiently concrete enough to cause consistent reordering of messages. I want to try out a more direct scheme where I collect an inbox of enabled events and prioritize the events.
   1. Try this out if I have time. See POS.
2. Inspired by the symmetry used in the FlyMC paper and the identical roles in ripple I would like to assign dynamic node ids based on when the nodes first enter the new consensus round. This should produce better results even with the delay encoding.
   1. Won’t influence much, state symmetry does. Takes too long, don’t implement.
3. I think I can improve this further by changing ripple code. Calls to start round are initiated by a client command or peer message from the scheduler instead of by the node itself.
   1. Try this out.
4. One step further is controlling the timerEntry calls in the individual nodes. Nodes only proceed and send consensus messages at timerEntry calls. Validation messages can occur slightly after due to building the ledger first. Judging by the ripple code, I think postponing messages to after timerEntry calls is also a promising technique and allows.
   1. Leave this for future work.

Ripple bug:

Check updating state on receiving message and try to mess with how it updates that state in the case of asynchronous execution, so lagging and receiving old messages.

* Validations.h line 616. Checks whether a received validation is stale or current. Stale validations are validations that are signed more than 3 minutes prior to the ledger under consideration’s close time or more that 5 minutes after the ledger under consideration’s close time or seen more than 3 minutes after the ledger under consideration’s close time.
  + Could remove one condition and see how it affects the network
* PeerImp.cpp line 1640. onProposeSet -> checkPropose:2638 -> NetworkOPs.cpp::processTrustedProposal:1817 -> RCLConsensus.cpp::peerProposal:935 -> Consensus.h::peerProposal:698  
  Stores the PeerPosition in recentPeerPositions. Keeps at most 10 proposals per peer **What if less are stored?** Proposals are ‘played back’ on the start of a round. Goes through all proposals in recentPeerPositions.  
  -> Consensus.h::peerProposalInternal:718  
  Return value determines whether to relay the proposal. Replaces (or adds) proposal to currPeerPositions. If proposal is bowOut remove peer’s votes from disputes and proposal from currPeerPositions and add peer to deadNodes. **What if we no longer properly remove bowed out peers’ votes?** If this proposal is initial it records closeTime vote. **What if we change the closeTime vote also if a particular proposeSeq is exceeded?** Tries to find the proposed txSet. If not available it will call gotTxSet when it gets it. If available updateDisputes  
  -> Consensus.h::updateDisputes:1674 -> Goes through all disputed transactions and updates the votes for the peer based on the presence of a transaction in the set.  
  -> DisputedTx.h::setVote:146 -> **What if we no longer track old votes correctly? If node changes to yes, it normally removes one nay and adds one yay. How can we change this to** Updates the number of yays and nays on a disputed transaction based on the peer’s vote.
* Consensus.h::GotTxSet:837 is not checked or feed so could be used to clog the network. When it receives a transaction set it checks whether it already has it and if a peer is still proposing it, before updating disputes. **What if the already has it check is removed? It updates disputes for peers that that have this txSet in the currPeerPositions. For this to have an effect, there must be txSet in currPeerPositions. We would like this to be an earlier peer proposal. Remove check for higher proposeSeq. gotTxSet gets called from peerProposalInternal if this txSet was not proposed before, but it is known? gotTxSet gets called from receiving the txSet from a peer if it was not seen before and not available. One goal is to get enough peers to bow out in one round. Will this stop consensus from progressing? Not necessarily, 0 participants will result in ConsensusState::Yes. Otherwise, 80% or more of the current proposers need to agree with the peer. It counts the current proposers as the peers in ‘currPeerPositions’. bowOut results in removal from currPeerPositions. shouldPause() looks at the amount of laggards in the UNL to determine whether it should pause. Cycles through quorum percentage, min 80%. This only happens when a node is ahead.**
* PeerImp.cpp::onMessage(statusChange):1736 -> Bookkeeping with regard to previousLedgerHash and closedLedgerHash if outOfSync or changeLedger. Updates min and max ledger seq, checks if peer is tracking (<24 is diff is converged, >128 is diverged). Publishes status change to subscribers.

How to create a fork

* Set validation threshold to 20: Equates to removing validation mechanism
* Reach consensus on two conflicting proposals
  + Achieved through earlier proposeSet message arriving later.
  + 3 say tx1: account 1 sent to account 2
  + 2 say tx2: account 1 sent to account 3
* 80% of UNL should agree on proposeSet how? Heavy dropping
  + 1 and 2 should see 1, 2, 3 and 4 proposing tx1. 3, 4 and 5 should see 2, 3, 4 and 5 proposing tx2 in the final proposing round to reach threshold.
  + 2, 3 and 4 should change their proposal. Internal state:
    - 2 should change from tx2 to tx1
    - 3 and 4 should change from tx1 to tx2
  + ProposeSeq = 0:
    - [1, 3, 4] tx1, [2, 5] tx2
      * 1 sees: [2, 4] (tx1: 2, tx2: 1) stays tx1
      * 2 sees: [1, 4] (tx1: 2, tx2: 1) goes tx1
      * 3 sees: [2, 5] (tx1: 1, tx2: 2) goes tx2
      * 4 sees: [2, 5] (tx1: 1, tx2: 2) goes tx2
      * 5 sees: [2, 3] (tx1: 1, tx2: 2) stays tx2
    - No process declares consensus
  + ProposeSeq = 1:
    - [1, 2] tx1, [3,4, 5] tx2
      * 1 sees: [3\_0, 5], still has [2, 4] (tx1: 4, tx2: 1) consensus tx1
      * 2 sees: [1, 3\_0, 5], still has [4] (tx1: 4, tx2: 1) consensus tx1
      * 3 sees: [1, 4, 5], still has [2] (tx1: 1, tx2: 4) consensus tx2
      * 4 sees: [1, 3, 5], still has [2] (tx1: 1, tx2: 4) consensus tx2
      * 5 sees: [1, 3, 4], still has [2] (tx1: 1, tx2: 4) consensus tx2
  + Two different txSets are agreed upon. Many different interleavings that result in this.
  + Without proposeSeq check bug: Network partition.
  + ProposeSeq = 0:
    - [1, 3, 4] tx1, [2, 5] tx2
      * 1 sees: [2, 3, 4, 5] (tx1: 3, tx2: 2) stays tx1
      * 2 sees: [1, 3, 4, 5] (tx1: 3, tx2: 2) goes tx1
      * 3 sees: [2, 5] (tx1: 1, tx2: 2) goes tx2
      * 4 sees: [2, 5] (tx1: 1, tx2: 2) goes tx2
      * 5 sees: [2] (tx1: 1, tx2: 2) stays tx2
    - No process declares consensus
  + ProposeSeq = 1:
    - [1, 2] tx1, [3,4, 5] tx2
      * 1 sees: [2], still has [3, 4] (tx1: 4, tx2: 0) consensus tx1
      * 2 sees: [1], still has [3, 4] (tx1: 4, tx2: 0) consensus tx1
      * 3 sees: [4, 5], still has [2] (tx1: 0, tx2: 4) consensus tx2
      * 4 sees: [3, 5], still has [2] (tx1: 0, tx2: 4) consensus tx2
      * 5 sees: [3, 4], still has [2] (tx1: 0, tx2: 4) consensus tx2
  + With 5 nodes switching proposal and two different partitions.
  + ProposeSeq = 0:
    - [1, 2, 3] tx1, [4, 5] tx2
      * 1 sees: [4, 5] (tx1: 1, tx2: 2) goes tx2
      * 2 sees: [4, 5] (tx1: 1, tx2: 2) goes tx2
      * 3 sees: [4, 5] (tx1: 1, tx2: 2) goes tx2
      * 4 sees: [1, 2, 3] (tx1: 3, tx2: 1) goes tx1
      * 5 sees: [1, 2, 3] (tx1: 3, tx2: 1) goes tx1
    - No process declares consensus
  + ProposeSeq = 1:
    - [1, 2, 3] tx2, [4, 5] tx1
      * 1 sees: [2, 3], still has [4, 5] (tx1: 0, tx2: 5) consensus tx2
      * 2 sees: [1, 3], still has [4, 5] (tx1: 0, tx2: 5) consensus tx2
      * 3 sees: [1, 2], still has [4, 5] (tx1: 0, tx2: 5) consensus tx2
      * 4 sees: [5], still has [1, 2, 3] (tx1: 5, tx2: 0) consensus tx1
      * 5 sees: [4], still has [1, 2, 3] (tx1: 5, tx2: 0) consensus tx1
  + No dropped messages except for proposeSeq1 which need to be replaced by proposeSeq0
  + ProposeSeq = 0:
    - [1, 3, 4] tx1, [2, 5] tx2
      * 1 sees: [2, 5] (tx1: 1, tx2: 2) tx2
      * 2 sees: [1, 3, 4, 5] (tx1: 3, tx2: 2) tx1
      * 3 sees: [1, 2, 4, 5] (tx1: 1, tx2: 2) tx1
      * 4 sees: [1, 2, 3, 5] (tx1: 1, tx2: 2) tx1
      * 5 sees: [1] (tx1: 1, tx2: 1) stays tx2
    - No process declares consensus
  + ProposeSeq = 1:
    - [1, 2] tx1, [3, 4, 5] tx2
      * 1 sees: [3\_0, 4\_0, 5], still has [2, 4] (tx1: 2, tx2: 2) consensus tx1
      * 2 sees: [1, 3, 4, 5], still has [4] (tx1: 4, tx2: 1) consensus tx1
      * 3 sees: [1, 2, 4, 5], still has [2] (tx1: 1, tx2: 4) consensus tx2
      * 4 sees: [1, 2, 3, 5], still has [2] (tx1: 1, tx2: 4) consensus tx2
      * 5 sees: [1, 2\_0, 3\_0, 4\_0], still has [2] (tx1: 2, tx2: 1) consensus tx2