CS 425 MP2 CP2 Report

I. Submission Information

Group Members: Jianfeng Xia(jxia11), Cheng Hu(chenghu3)
VM Cluster Number: 10
GitLab Repository: https://gitlab.engr.illinois.edu/chenghu3/cs425_mp2

Instructions:

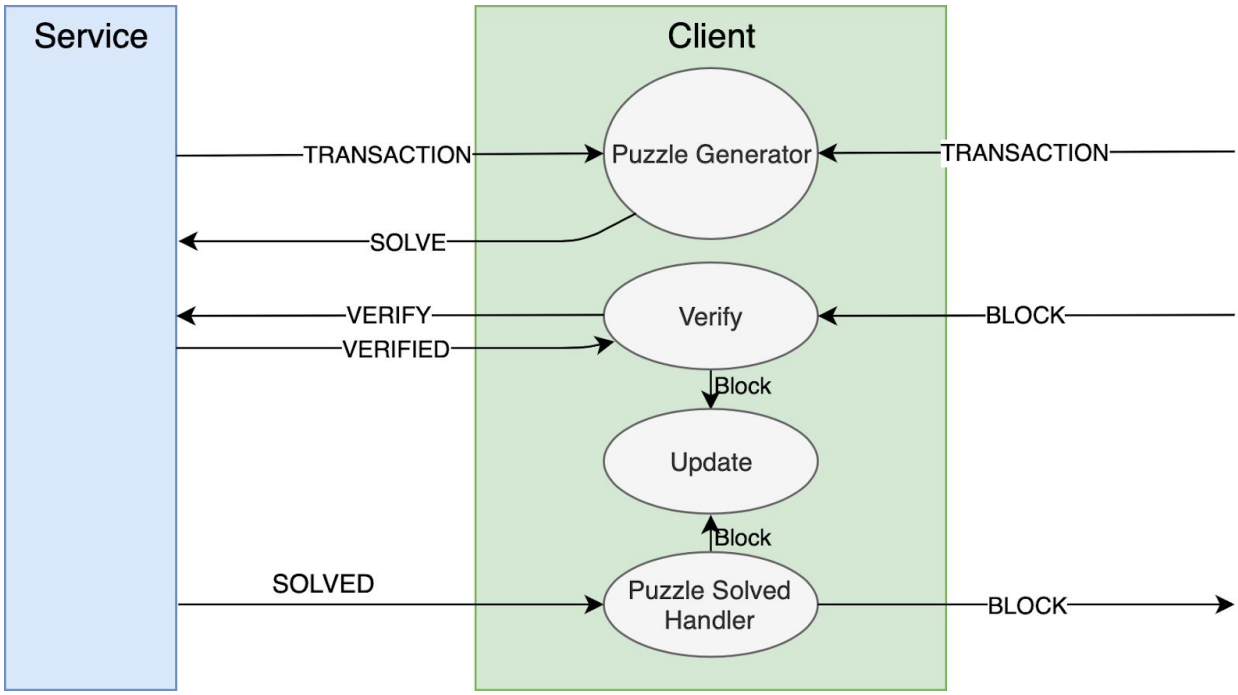
- The service should be run at vm number 7, port 8888
- To build: `go build client.go`
- To run: `./client [PORT]`
- Alternatively, use our Python3 script `experiment.py` to start multiple clients (starting a node every 0.5s) and write stdout to log files.
  Usage of Python script: `python36 experiment.py [NUMBER OF CLIENTS]`
- We have also included scripts to analyze performance and generate plots:
  ```
  cd logs
  python3 bandwidth.py
  python3 propagation.py
  ```
  We have also provided the Jupyter Notebook files:
  ```
  logs/bandwidth.ipynb
  logs/propagation.ipynb
  ```
  Note:
  1. Our plot scripts use matplotlib and numpy. We suggest running the scripts on machines with these libraries.
  2. Because the log files are large (we use logs in the case of 100 nodes, 20 mesg/s), please be patient when running the scripts.

matplot

- Revision to be graded:
  d4ee089f853326ec266f5b4e7cc9d778e6549004

II. Algorithms & System Design

Overview



Blocks & Puzzle Generation

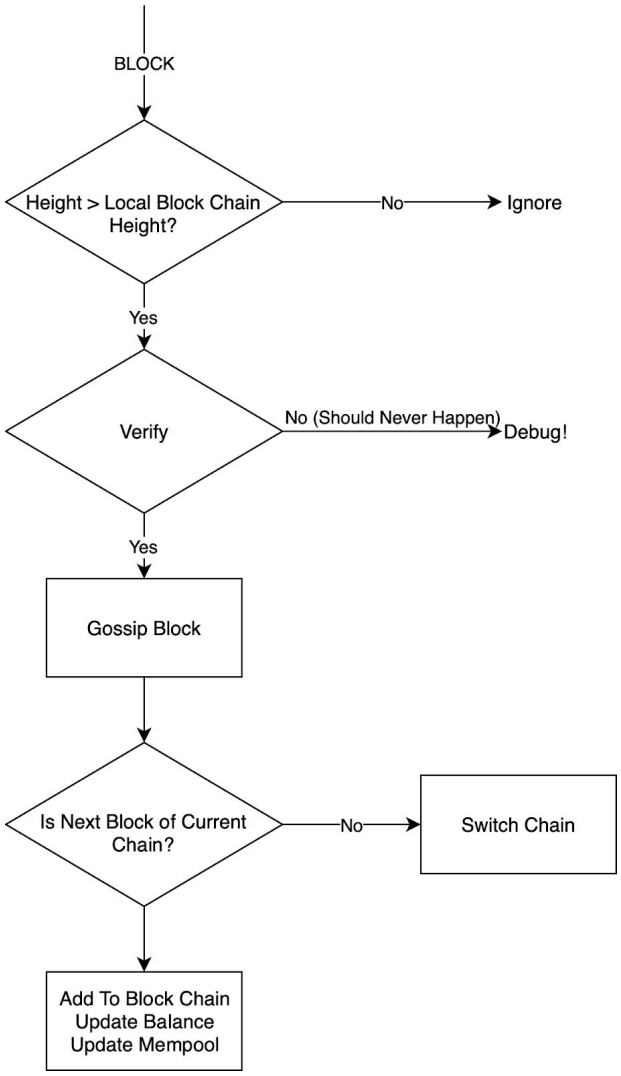
In addition to previous block hash, transaction list and puzzle solution, we added the block generator's IP+port and the height of the block to our block definition to uniquely identify blocks and avoid hash collisions. To generate puzzles, we define a `batchSize`. A node periodically checks if the amount of new transactions exceeds `batchSize`. If so, it generates a new puzzle by adding the new transactions to its tentative block and compute the SHA256 hash.
The transactions in each block are sorted in chronological order using the transaction timestamp.

Verify & Update

The diagram below explains how a received block is processed.

- When a node receives a block, it first checks if the block is a new block (has a higher height than its local blockchain) to avoid redundant operation.
- It then preemptively updates its local blockchain height to the new block's height (again, to avoid redundant operation).
- Then, it verifies the block with the service.
- After it receives `VERIFIED` from the service, it adds the block to the list of blocks to be sent using gossip.

- Then, the node checks if the new block is the next block of the current chain. If not, then it switches to the longer chain by asking the block generator node for any missing information (see Longest Chain Rule section).
- Otherwise, it is safe to update the node's local states. The node adds the new block to its local chain, updates its local balances and removes the committed transactions from its mempool.



Balance Updates & Consensus Rule

In our system, each node maintains a local map of account balances. After accepting a verified block, each node computes the updated balances locally. To enforce the consensus rule, It ignores any transaction that results in negative balance. Therefore, our protocol ensures that every node that has the same blockchain will have identical account balances.

Longest Chain Rule

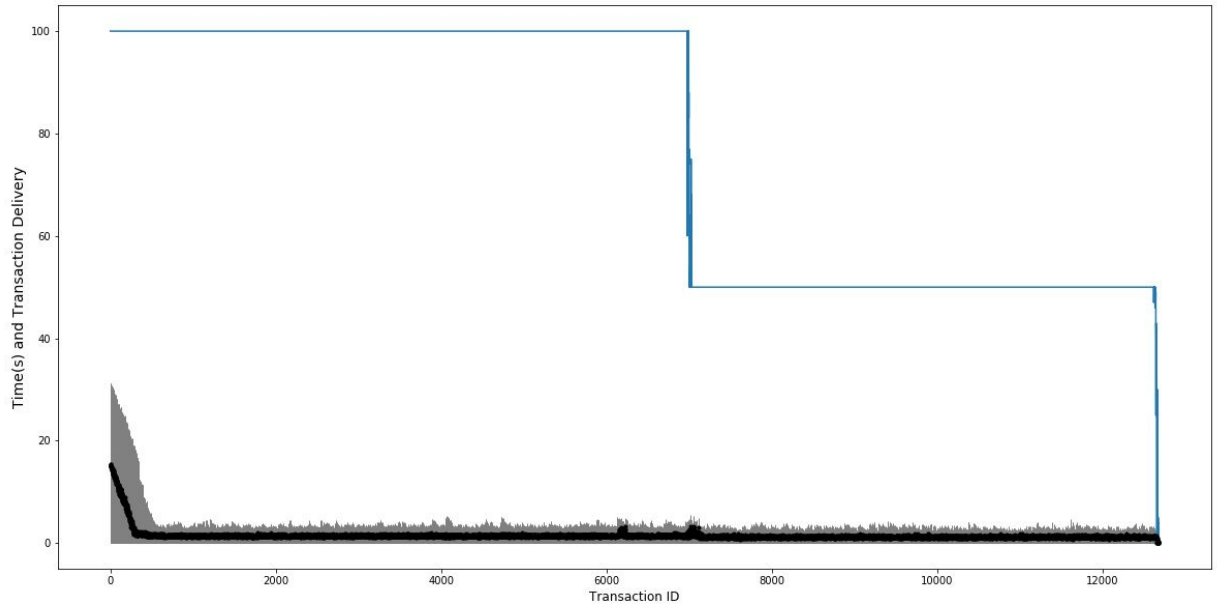
To handle chain split and enforce the longest chain rule, we implemented a switch chain mechanism. When a node receives a future block or a block on another chain:

- If it's a future block, the node first contacts the block generator for any missing blocks.
- If the new block(s) is on another chain: the node finds the first common ancestor of the two chains by sequentially asking for previous blocks until the hashes match. Then, the node only needs to update the blocks after the common ancestor to switch to the new chain.
- The node also asks the block generator for its mempool and account balance map and overwrites its own.

Evaluation

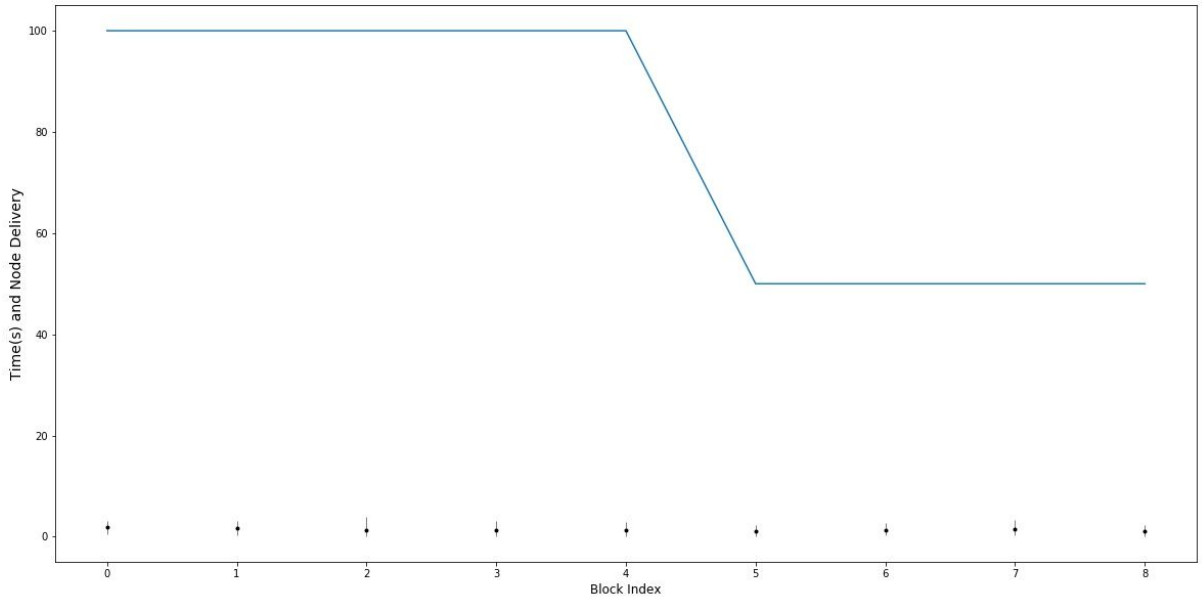
As in Checkpoint 1, our plots are generated using data from an experiment with 100 nodes initially and 20 transactions per second. After the system runs for a while(6 minutes), a `thanos` command was sent and half of the nodes died.
We logged and processed propagation delays of transactions and blocks using the same approach as in Checkpoint one. We have also verified that the blocks and balances are identical in all nodes. In the following experiment, **no chain split was observed.**

Transactions Propagation



Propagation delay and delivery count of each transaction

Block Propagation

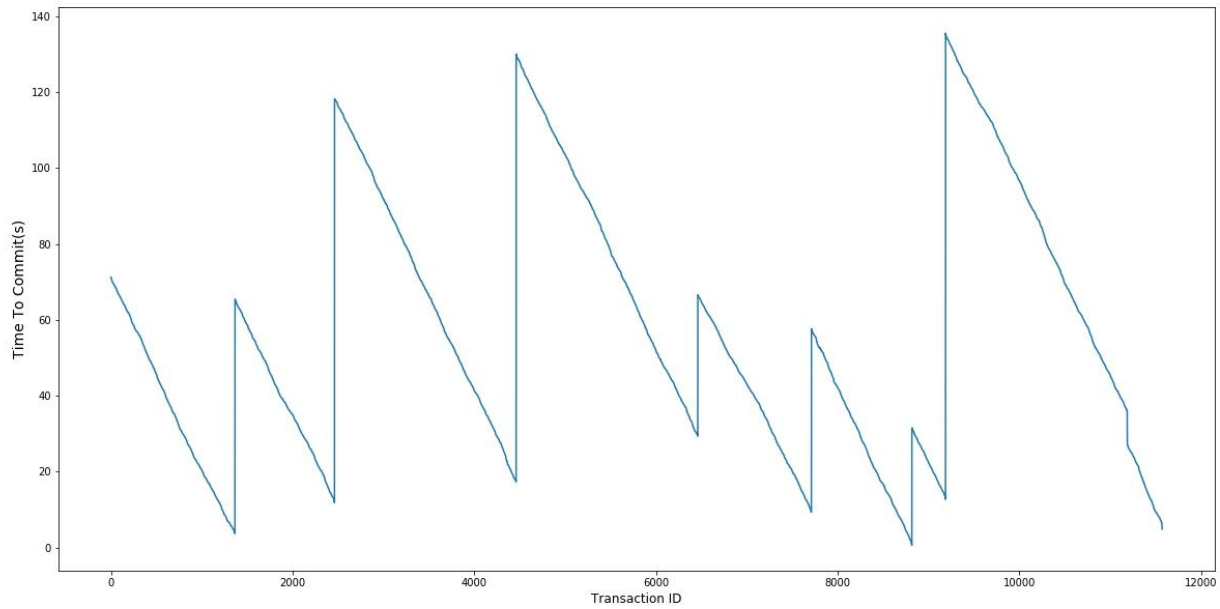


Propagation delay and delivery count of each block

The vertical grey lines are the ranges of propagation delays, and the black dots are medians of propagation delays. The blue line indicates the delivery count.
It is clear from the graph that we have achieved **100%** delivery and consistent, short propagation delays. The longer delays at the start are due to node joins happening after the transactions are created.

Time To Commit

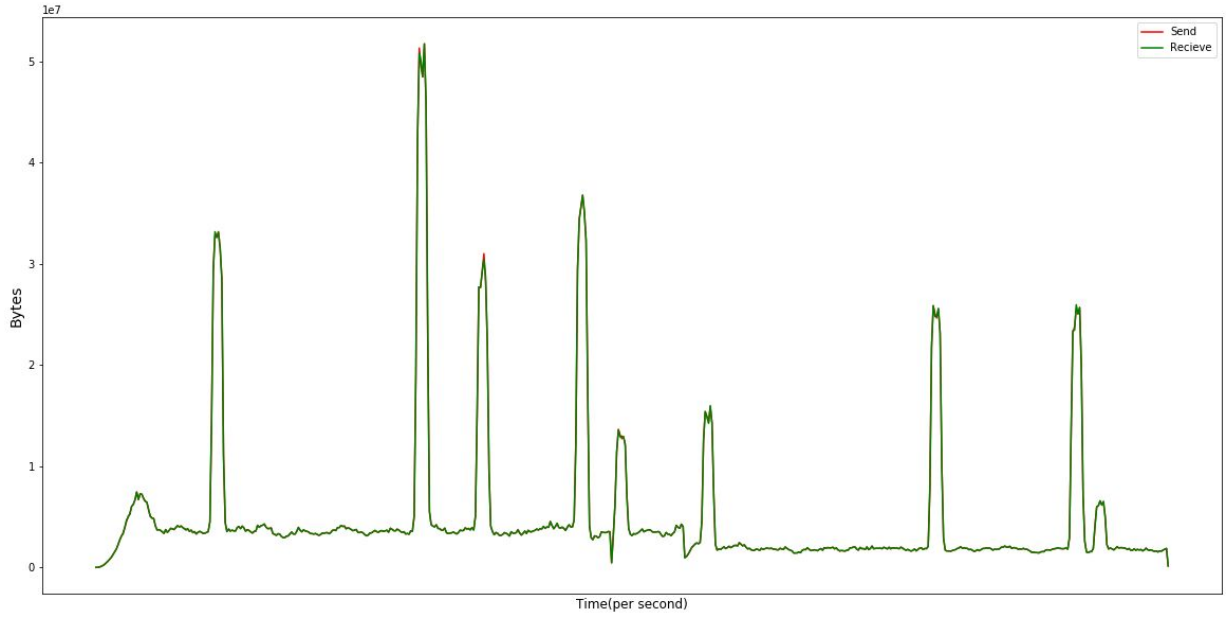
We have also logged the transactions in each block to analyzed the time to commit for each transaction (the time a transaction takes to appear in a block):



This distribution is expected because we sorted the transactions in chronological order.

Bandwidth

We logged the number of bytes for each message and calculated the bytes per second to measure bandwidth. The experiment results are as follows:



Aggregated bandwidth of all nodes(per second)

The spikes are due to blocks propagation. As we expected, the height of a spike is roughly proportional to the time interval between the spike and the previous spike because we are sending all transactions for the period along with the block.

Chain Splits

Chain splits are very rare in our implementation because the block propagation speed is much faster than the block generation speed. However, it did happen in one experiment where `thanos` happened at around the time a block is being gossiped. Intuitively, half of the nodes dying might slow down the propagation speed of gossip. Approximately ⅓ of the nodes were affected. The splits had length 1 (only one block was affected).
Later, the affected nodes successfully switched to the longer chain according to the longest chain rule.