CS425 MP2 Report
Chris Coronado
Lishen He

Repo: https://gitlab.engr.illinois.edu/lhe10/mp2
Revision: c9b301c1ed6f892abb4a2045e8c003eab4b14b79
Build Instruction:

- Go to build folder: `cd build`
- Generate node executable: `make`

Run instruction:

- Create the introduction service, specifically in VM 1
  - `python3 ../src/python/mp2_service.py 9999 1.0`
- Create 1 node per VM
  - ./node <NAME> <VM#> <PORT> <SERVICE_PORT>
  - Example: `./node node2 2 2222 9999`
- Create multiple nodes per VM
  - ./launcher.sh <VM#> <NUM_NODES> <SERVICE_PORT>
  - Example: `./launcher.sh 2 20 9999`

Logged transactions path:

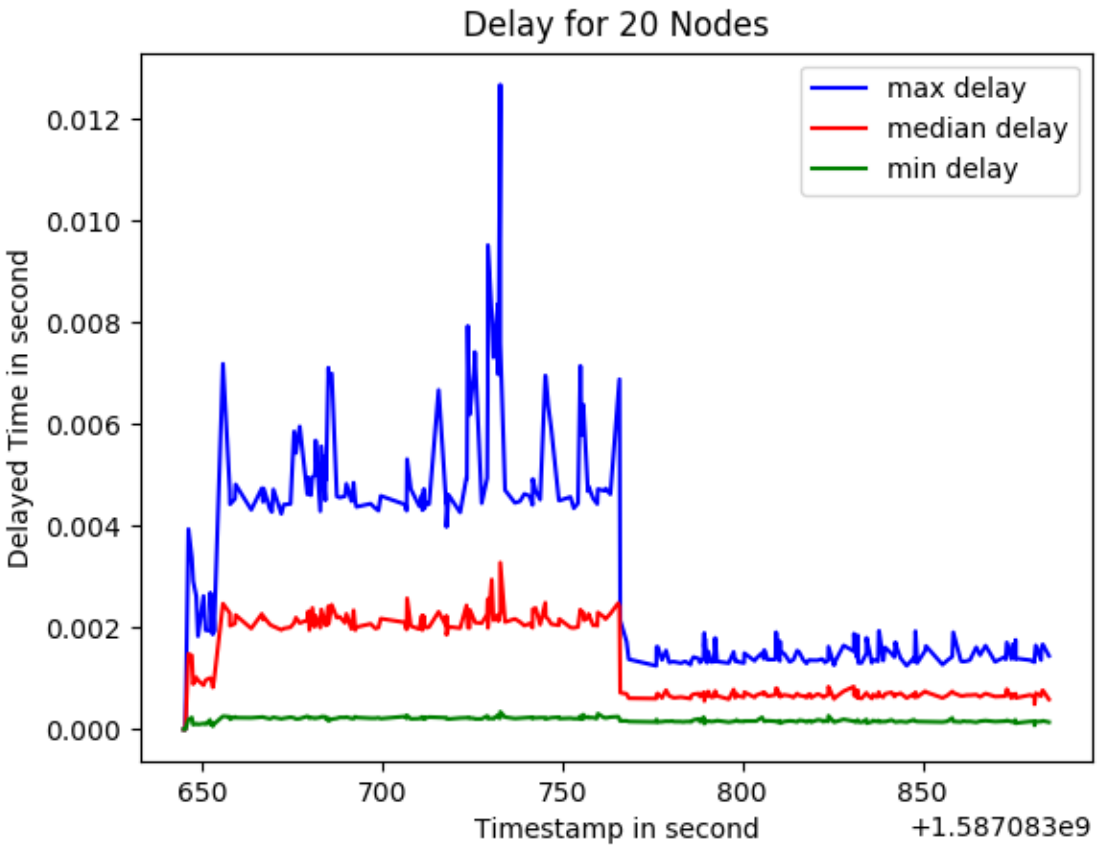
- ./build/transOutput_20nodes
- ./build/transOutput_100nodes

Logged file path and scripts

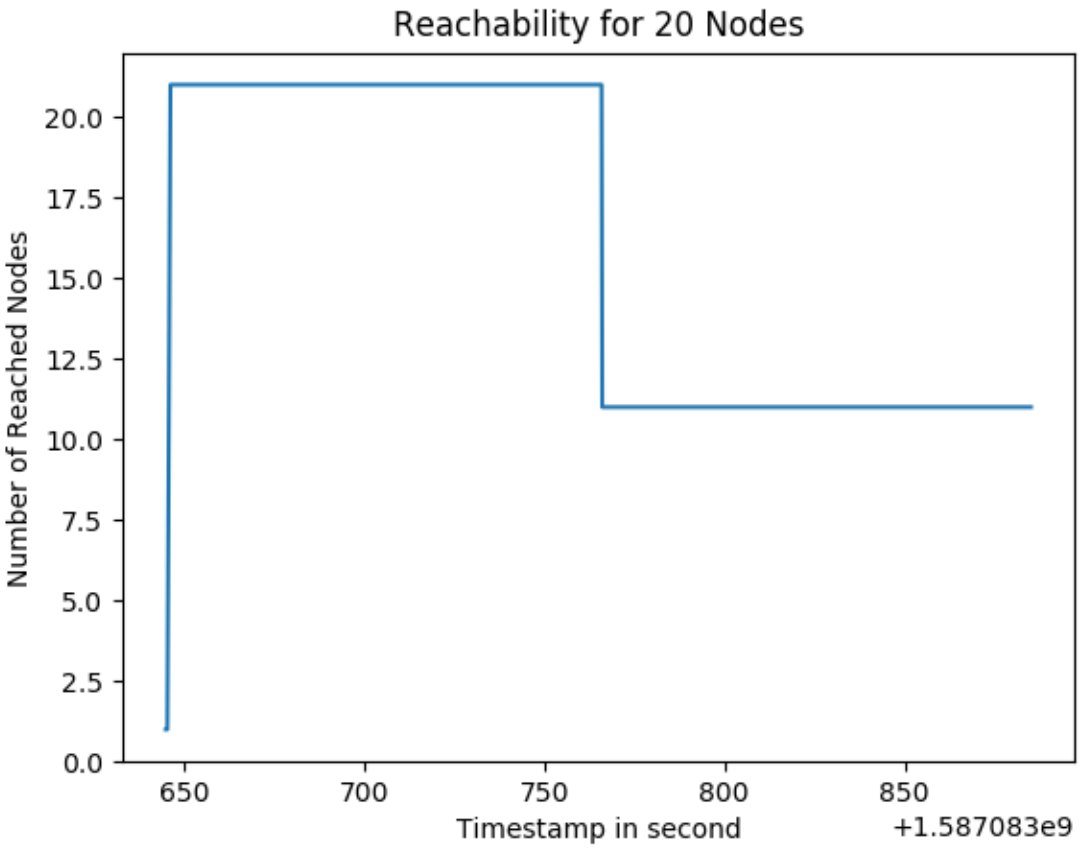
We will answer questions and provides graphs specifically to Part I and Part II first in our report. At the end of the report, we will provide the design of our MP2 as requested.

Part I

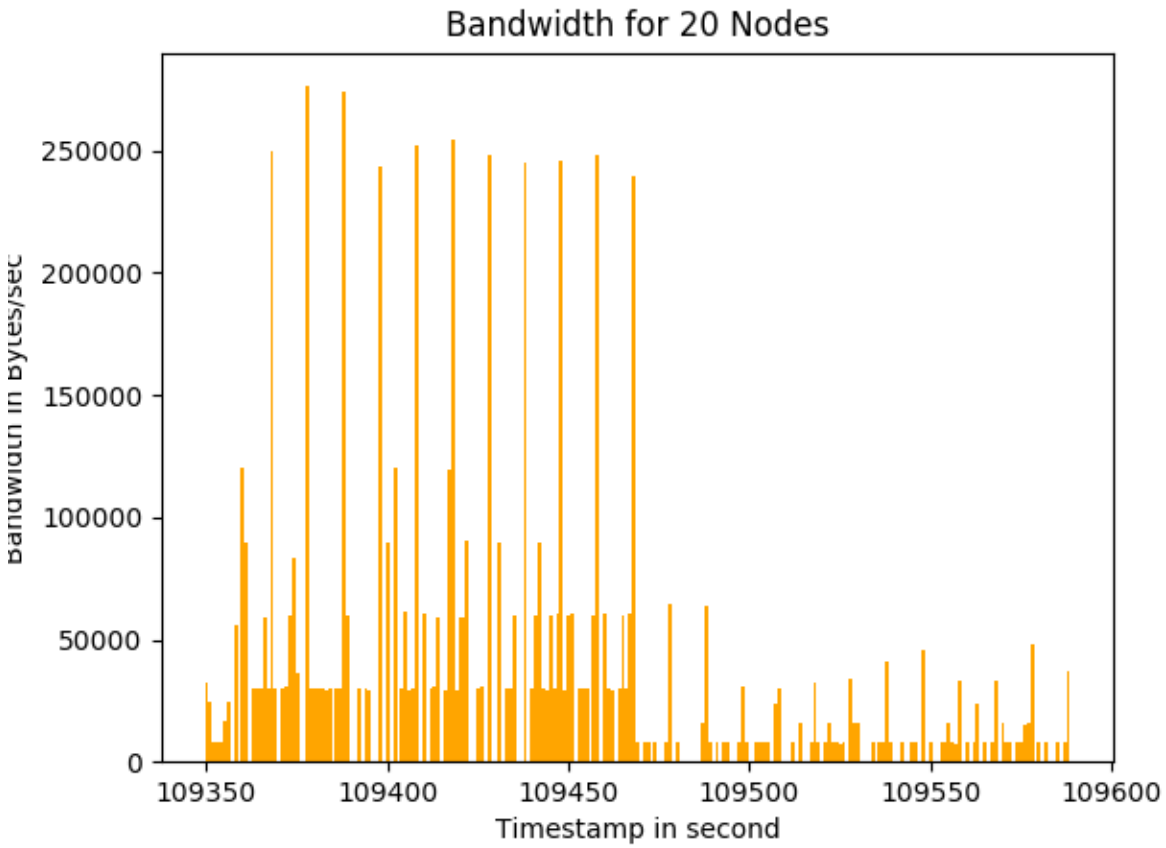
1. Evaluation Scenarios
   a. We start the service on VM1 with 1 transaction generated per second. On VM2, we emulate 20 nodes by using: `./launcher.sh 2 20 4000`. We kept all nodes running for about 120 seconds and then initiated “thanos” command to kill half of nodes. Then we keep running the remaining half nodes for 120 seconds to verify functionality.
   b. Next, we repeat the same experiment using additional 4 VMs to emulate 100 nodes in total with 20 messages per second.
2. Graphs
   a. Propagation Delay – 20 nodes



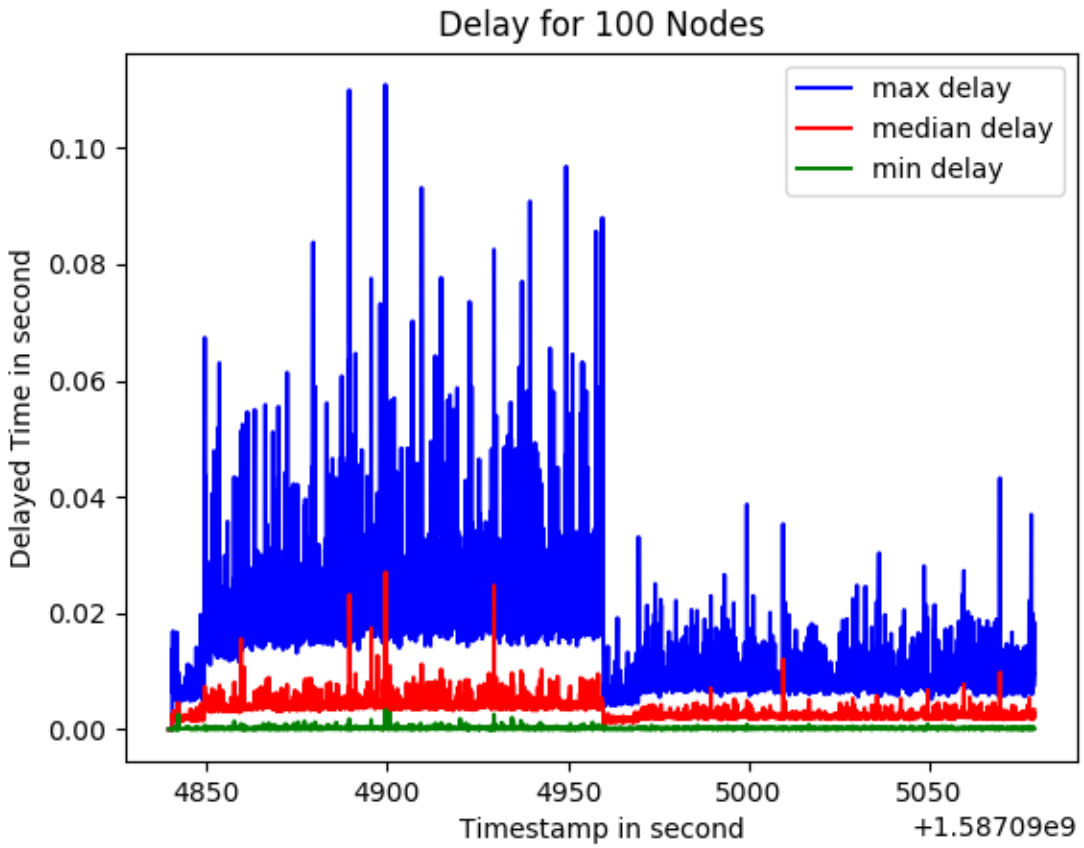
b. Reachability – 20 nodes



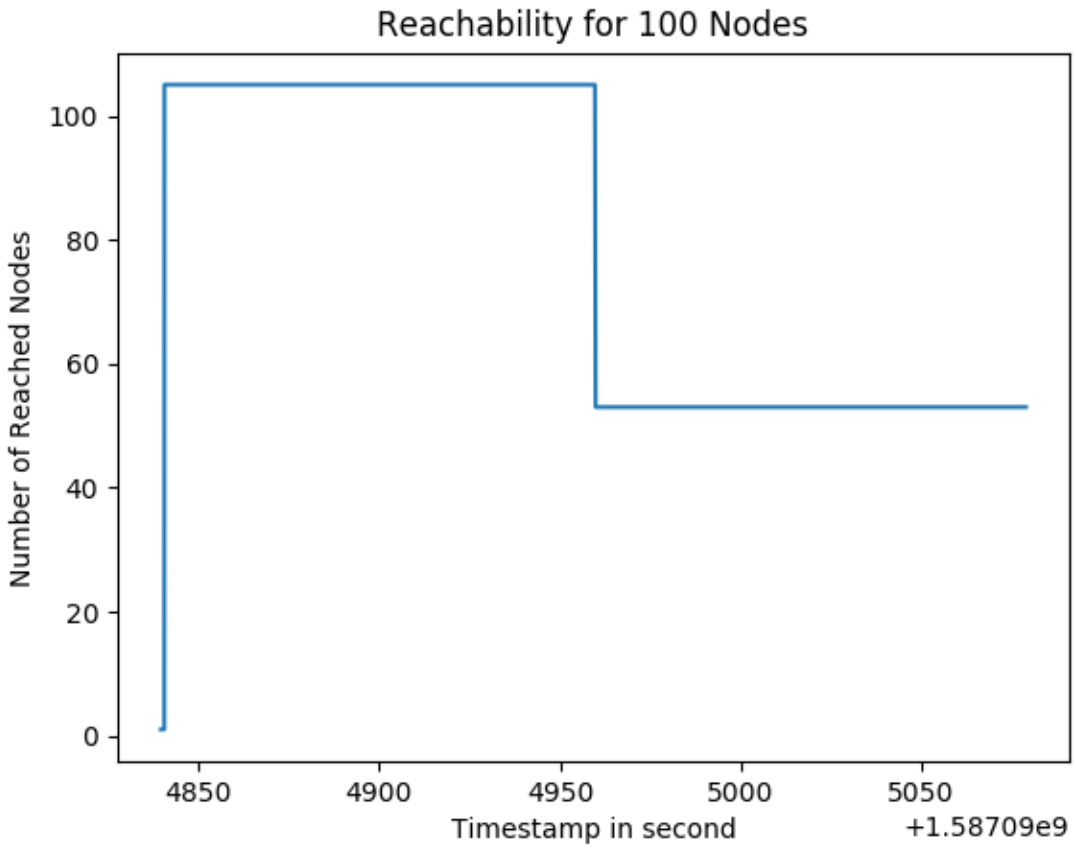
c. Bandwidth Usage – 20 nodes



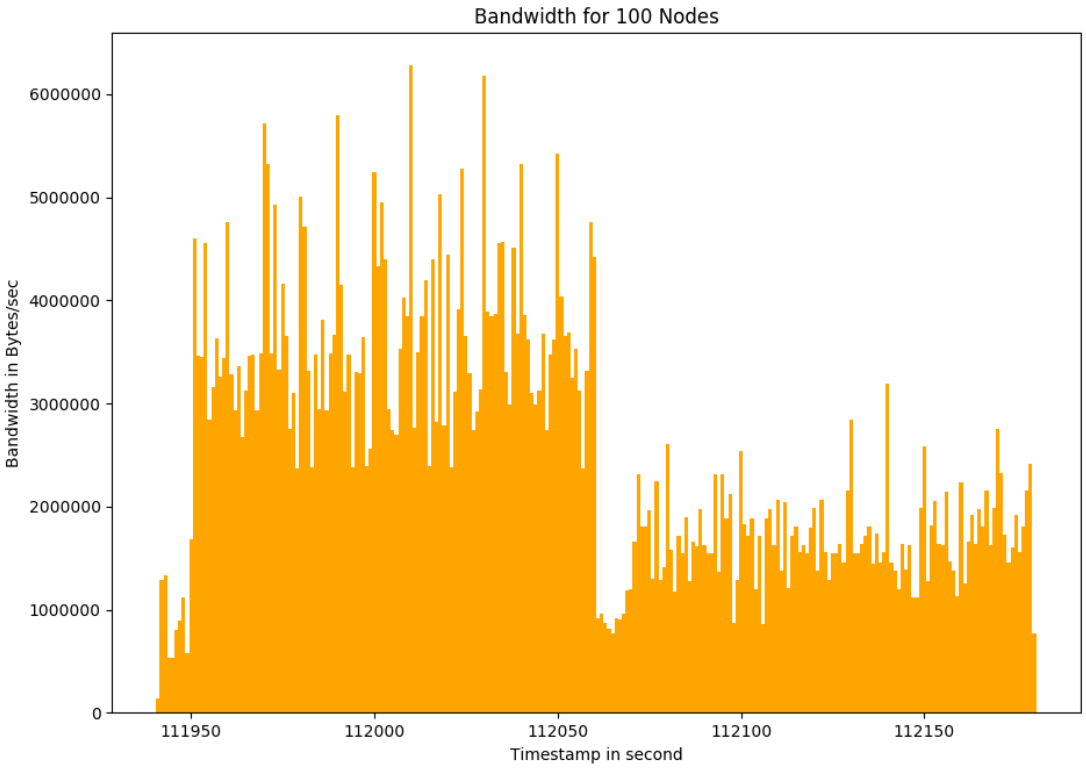
d. Propagation Delay – 100 nodes



e. Reachability – 100 nodes



f. Bandwidth Usage – 100 nodes



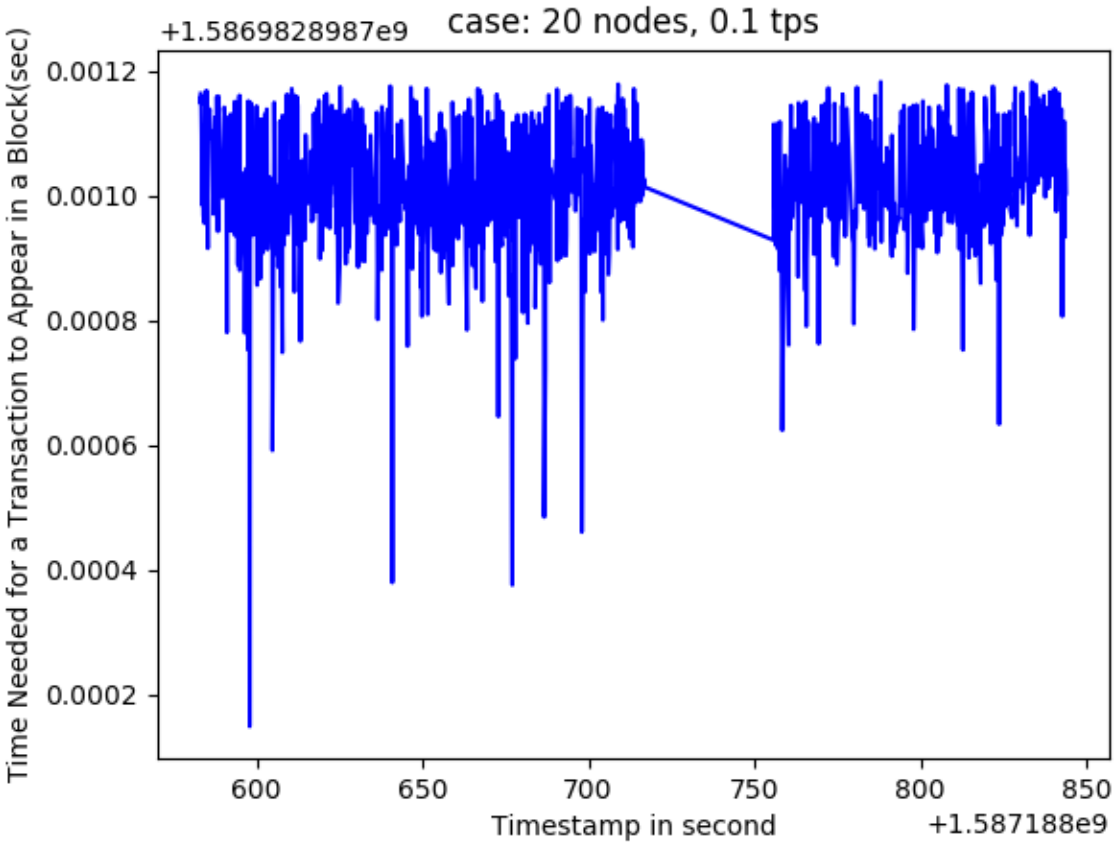
3. Discussion on the occurrence of “thanos”

   The “thanos” command is issued in the middle of the simulation (~2min). We can see from the reachability plot, the number of nodes reached by a message decreased to exactly half of the node. This verifies our implementation. The max and median delay dropped significantly in both scenarios. The bandwidth dropped significantly in both cases. In the 20 nodes case, bandwidth data is sparse: some 1 second interval does not use any bandwidth so the it is zero. This is caused by the 1msg/sec generating rate and stability and efficiency of our system: the messages transmit fast enough to expose these seconds without any incoming transaction from the service VM.

Part II

1. Experiments performed
   a. We start the service on VM1 with 1 transaction generated per second and 0.1 for the block rate. On VM2, we emulate 20 nodes by using: ./launcher.sh 2 20 4000.
   b. Next, we repeat the same experiment using additional 4 VMs to emulate 100 nodes in total with 20 messages per second. Now the transaction rate is set to 20tps.
2. How long does each transaction take to appear in a block? Are there congestion delays?

a. Small 20 nodes with 20 messages/second, default block rate
We realized that we can hardly fill the block if only 1 message is generated per second. Neither can it timeout with a reasonable value. So we bump up message generating rate to 20 messages/sec and run for a about 4 minutes. [Note that label for y is wrong: should be per 1000 sec].



b. We did not have a consistent graph for a larger scale test for 100 nodes. Thus, we would not include any graphs here.

3. How often do chain splits occur? How long is the longest split you observed?
In all our tests, we did not observe any chain splits. This is partly due to the parameter for the experiments we set.
   a. N/A
   b. N/A

Design for MP2

1. <u>Node Connectivity</u>
   a. How your nodes keep connectivity
   Each node maintains a global variable of connectivity:
   ```
   neighbors map[string]Node = make(map[string]Node)   // Dict of connections, accessed by name
   n_mutex   sync.Mutex
   ```

This variable is appended during introduction and polling, which will be discussed in the next. And this will be reduced upon detection of failed nodes, also explained below.
Each node maintains a map of connections, with each connection representing a neighboring node. This map is updated periodically whenever a node discovers any new connections, or whenever a connection is considered dead.

b. How discover each other beyond introduced ones
When first joining the network, nodes connect to up to 3 other nodes thanks to the introduction service's INTRODUCTION messages. In order to discover additional nodes, a "pollForIntros" function runs on its own thread and periodically poll neighbors' known neighbors' information. When a neighbor receives a poll request, it will randomly select a subset of neighbors from its list. From these, the neighbor creates a series of INTRODUCTION messages which allow the requesting node to connect to more of the network. To prevent much resources being taken from maintaining unnecessary number of neighbors, especially in the case of a large network, we set a hard limit of maximum number of neighbors to be 20. Polling results will be discarded if it leads to more than 20 neighbors. The polling period can be in principle purely randomized and we set it to be 2 second which has proven to work fine for this MP. In the bandwidth plots above, we can see spikes that comes out to be separated by about 2 seconds. This is as a result of the polling action.

c. How to detect failed nodes
Within a node, each connection has a dedicated thread which will listen for any new messages and dispatch them to the corresponding threads to be processed. When a connection is closed, the go reader will return some sort of error, typically an EOF. Upon detection of failure, the map of connections is modified, and the dead connection is removed.

```
msg,err := r.ReadString('\n')
if err != nil {
    fmt.Printf("(%s)#:Failure to read %s : %s\n",self_name, n.name, err)
    // Connection closed, remove from neighbors
    n_mutex.Lock()
    delete(neighbors, n.name)
    n_mutex.Unlock()
    return
}
```

d. How our code is robust
The code is robust to failures because it is essentially constantly trying to repair the network in a semi-random fashion. All new nodes are randomly connected to up to 20 neighbors, so, in the event of a massive failure, it is unlikely that every single neighbor would be taken offline. Because of this, the network is able to eventually recover from a large failure because the remaining nodes will eventually re-discover and reconnect.

2. <u>Transaction propagation</u>

a. How transactions are propagated: Algorithm
   We use the Gossip algorithm to propagate the transaction. Each connection is bi-directional, so upon receiving a transaction, a node will broadcast the transaction to all known neighbors. It will also add the UID to a queue. Whenever a transaction is received, the node will determine if it has already propagated the transaction by referencing the queue. If the message is previously unknown, it will broadcast. Otherwise, the transaction is ignored. Upon receiving any new message, a node propagates this to all of its neighbors using the following code. As there are 20 neighbors held by each node and the worst test case is 100 nodes. This propagation should take no more than just a few rounds.
b. Parameters and how we arrive at them
   Just as a summary for the above.
   i. Poll from others
      2 second polling period – this came from testing. Once a suitable behavior was reached, we decided to leave it as is.
   ii. Maximum number of neighbors kept by a single node
      20 – This was another arbitrary choice. The system seemed to work well this this constant, so it was kept. It is always possible to increase this in order to increase robustness at the cost of bandwidth consumption.
   iii. Justification
      When a thanos event occurs, the probability of at least one node singled out by thanos is $1 - \left(1 - \frac{1}{2^{20}}\right)^{20} = 0.0000191$

3. <u>What the process is for validating transaction validity and collecting them into blocks?</u>
   Each node maintains a map of balances. Upon receiving a transaction, it is first broadcast to all neighbors, regardless of validity. From here, the transaction is evaluated. Each node maintains a map of all accounts, and this map is referenced when a new transaction arrives. If the transaction would result in a negative balance, it is rejected and discarded. Once the transaction is verified, it is added to a queue of valid transactions. From here, the block-building thread takes over. First it waits for one of two conditions (timeout, or the number of transactions is greater than 2000). After one of these events, the block-construction thread will take up to 2000 transactions into a buffer. From here the block is assembled, and, if a solution is found, the block will permanently remove the transactions from the queue and place them into a block struct. From here, the block is processed and propagated to the rest of the network.
4. <u>How new blocks are propagated and imported. Discuss particularly what happens during a chain fork</u>
   New blocks are also propagated using a gossip system. When a solution is found, the block-construction thread remove transactions from the mempool in order to be packaged into a block. From this point, the block struct is serialized using a gob encoder object, and the result is placed into a byte buffer. Before being transmitted, some metadata is prepended to the buffer, and then it is sent to all of the neighboring nodes. When a node receives a new block, it will first reconstruct the block struct from the byte

data. Then, if the block has not been seen before, it will be added to a queue (to identify that it has been seen) and repropagated. Finally, the validity of the block is determined by sending the solution hash and the combined hash (previous hash, transactions) off to the mp2 service. When verified, the block is either accepted or rejected, depending on its height.

Although we did not experience forks in our testing, the node would determine if a fork has occurred by comparing the imported block's previous hash with the previous hash generated by the top of the block-chain. If these hashes do not match, the node will send a CHAIN request to the original node. A CHAIN request prompts the original block to encode and send a copy of it's block-chain to the node that requested it. Upon receiving a reply to this message, the node would then replace the old block-chain with the reply and check the first n block for duplicate transactions, where n is the difference in height when the fork was detected.

5. Logging
    a. What information is logged and how you used this to generate the graphs
        i. Transactions – Part I
        ii. Bandwidth – Part I
        iii. Transactions in Blocks – Part II

            We record two sets of data. The first set of data is the earliest timestamp a transaction is seen by any node. The data are saved by a number of .txt files, each create by a different node, and saved in the folder of "New_Transactions". This is essentially the same as the what mentioned above. The other set of data is the earliest time a transaction appeared in a block. Since all blocks are created by some node, we only a look at blocks generated by each node and save their time stamps in a file. All files are saved in a folder called "Blocked_Transactions". A python script is used to read both sets of data into dictionaries with transaction content to be the key and timestamp to be the value. Then, a difference is taken which is the time taken for each transaction to appear in a block.
        iv. d
    b. Describe how scripts work

        The scripts name and their functionalities are described above. All the post-process scripts are in python saved in the folder src/python
6. Other notes
    a. How to emulate multiple nodes on one VM

        We use a shell script to run multiple instances of compiled program "node". This can emulate any number of nodes running on a single VM subject to the limit of computation resources only.