MP2: Cryptocurrency Report

Paul Pok Hym P Ng (ppng2)
Bruno Seo (sbseo2)

April 18, 2020

1 Cluster number / Repository

Our cluster number is 15.
Below is the URL of our git repository.
URL: https://gitlab.engr.illinois.edu/ppng2/ece428/tree/master
(Revision number: efc12f398062a5e14688023b66da41070218a3a5) (Generated using git rev-parse HEAD)

2 Instructions for running code

Please follow the instructions written as below.

Server Node

1. Upload `mp2_service.py` to your main server node.
2. Run logger.py by using the following command. The first argument is the port the server runs on (default set to `9998`). The second argument represents the frequency at which client transactions are generated. Here it is `0.5` Hertz.

```
    $ python3 mp2_service.py 9998 0.5
```

4. Wait for client nodes to be connect
5. Once all clients have connected to the logger, start them. After a certain amount of time type THANOS into the service to kill half the nodes.

Client Nodes

1. Open `config.py` and type the IP address of server where `mp2_service.py` is uploaded.
2. For every node, run the following command.

```
    $ python3 main.py CONNECT <VM> <INSTANCE_NUM>
```

3. `<VM>` denotes the number of machine you are using. Its number can range from 1 to 100. `<INSTANCE_NUM>` is the number to differentiate multiple nodes running in the same machine. For example, if you want to run two nodes in VM1, please type the following commands.

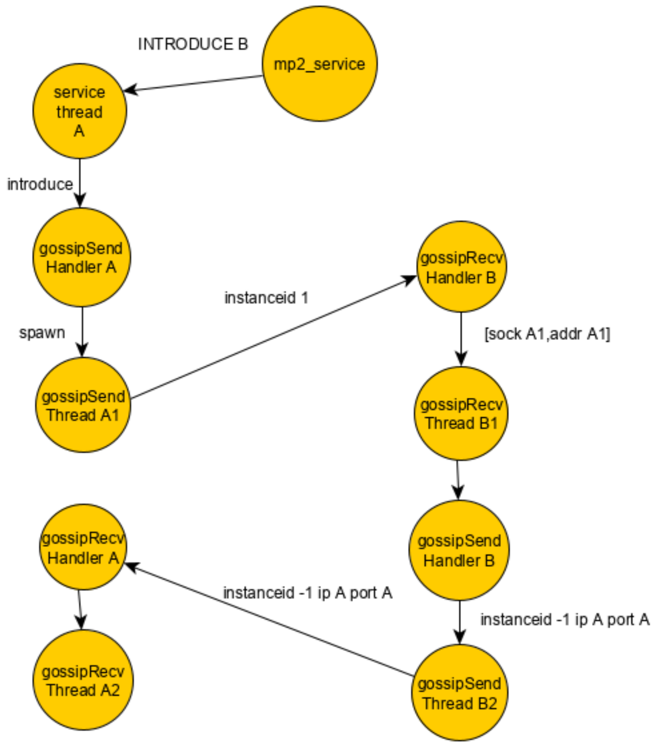

Figure 1: A flowchart depicting how the gossip connection is created

```
    $ python3 main.py CONNECT 1 1
    $ python3 main.py CONNECT 1 2
```

Use Ctrl+C to terminate the client. If you are restarting the client nodes and run into an error stating that "Address is already in use" please use the following command to kill all instance of Python and free up the ports. It may take a few seconds and a few invocations of the command for the port to be cleared.

```
    $ sudo pkill --signal SIGINT python3 && sudo pkill --signal SIGKILL python3
```

3 Part1

3.1 Gossip System Design

We designed gossip system such that two nodes communicate back and forth between two threads. This is done because the socket.recv() is a blocking call. Therefore, if we only used one thread on each node to communicate back and forth it would be possible to deadlock the system as both could end up on a `socket.recv()` call waiting for the other to send a message. While we could use a recv timeout we decided against it as it would still introduce delay. Let us discuss how this actually occurs. Pleaser refer to Figure 1

1. `mp2_service.py` sends an `INTRODUCE` message to `serviceThread` on node A.

2. `serviceThread` on node A detects this and adds the `INTRODUCE` message onto a globally accessible list named `introduceList`.
3. Node A's `gossipSendHandler` is in a while loop constantly checking if `introduceList` is empty.
4. Once it sees that it is not empty it spawns a new `gossipSend` thread A1.
5. Thread A1 now notifies node B's `gossipRecvHandler` that we wish to connect to it. It does so by sending its `INSTANCEID`.
6. B's `gossipRecv` handler now spawns a new `gossipRecv` thread and passes it the newly created socket and the introduction message.
7. The newly spawned thread B1 now creates its own `INTRODUCTION` message and puts it in B's `introduceList`.
8. Similarly to how A's `serviceThread` consequently created B1, A1 consequently created B2.
9. B2 then using the same steps will spawn A2.
10. With this we have a 2 way connection between nodes

Now that we have a 2 way connection, a node will not only send a message it received from a client to all its neighbors but also be able to forward transactions it overhears from its neighbors.

The explanation of how other nodes discover each other will be explained in Part 2 of the MP in the following section. The explanation for how failure detection will also be discussed there.

3.2 Log Analysis

All graphs that were generated via the `*.ipynb` files in the repository. The graphs have already been pre-generated there so there is no reason to re-run. However, if one desires to do so they would only need to change following variables and recalculate.

1. `NUM_OF_NODE` : This represents the number of nodes
2. `NUM_OF_INSTANCES`: The number of multiple instances running in the same machine

Then one only needs to re-run the code in each section.

Please keep in mind the following. In order to ensure that fresh data is always used, please delete all previous run's node text files to ensure no data gets leaked into the new run to avoid unnecessary duplicates.

3.3 Delay and Bandwidth Measurement

Once all N nodes are terminated, then our logger will generate N files in the logger directory labeled nodeN.txt.
Each contains logs for each node. The data we log include the following.

```
transactionId messageArrivedTime bytesTransfered
```

nodeTime is the time when the msg was generated on the node, *nodeData* is the data sent, *loggerTime* is the time which the logger received and parsed the msg, and finally *bytesTransfered* is the number of bytes that was sent by a particular node.

Bandwidth

In terms of bandwidth, we obtained it by sum of *bytesTransferred* on every second. This value was measured by tracking the ports used for TCP data transfer.

Propagation Delay

To measure the propagation delay, the proposed logger tracks the arrival time of every message. After the program is terminated, jupyter notebook gathers the arrival time from all of the nodes. Delay is calculated by the following equation.

```
| earliest timestamp of msg arrival - latest timestamp of msg arrival |
```

3.4 Graph Visualization

Figure 2 shows the result of bandwidth. When THANOS is executed after 60 seconds, the bandwidth has dropped drastically. That is because there is no need to transfer messages among the nodes where they stopped working.
In contrast to figure 2, figure 3 shows that propagation delay is increased. Intuitively, we can infer that multiple failure catastrophe occurred in transaction index #42. Overall propagation delay is increased because nodes messages started traversing the alive nodes in order to transfer messages. Min value is always 0 in this graph because we assumed that a node that has created his own message receives its message instantaneously.
Figure 4 also shows that massive killing catastrophe has occurred in transaction index #42. While there were 40 nodes alive at the beginning, the reachability dropped exactly into half in in transaction index #42.

4 Part2

4.1 System Design

4.1.1 Extended Gossip System Design

We mpw discuss how we extend our gossip protcol. While we keep the same method of introducing and connecting to nodes, we add another message `NODE IP INSTANCE`. Whenever

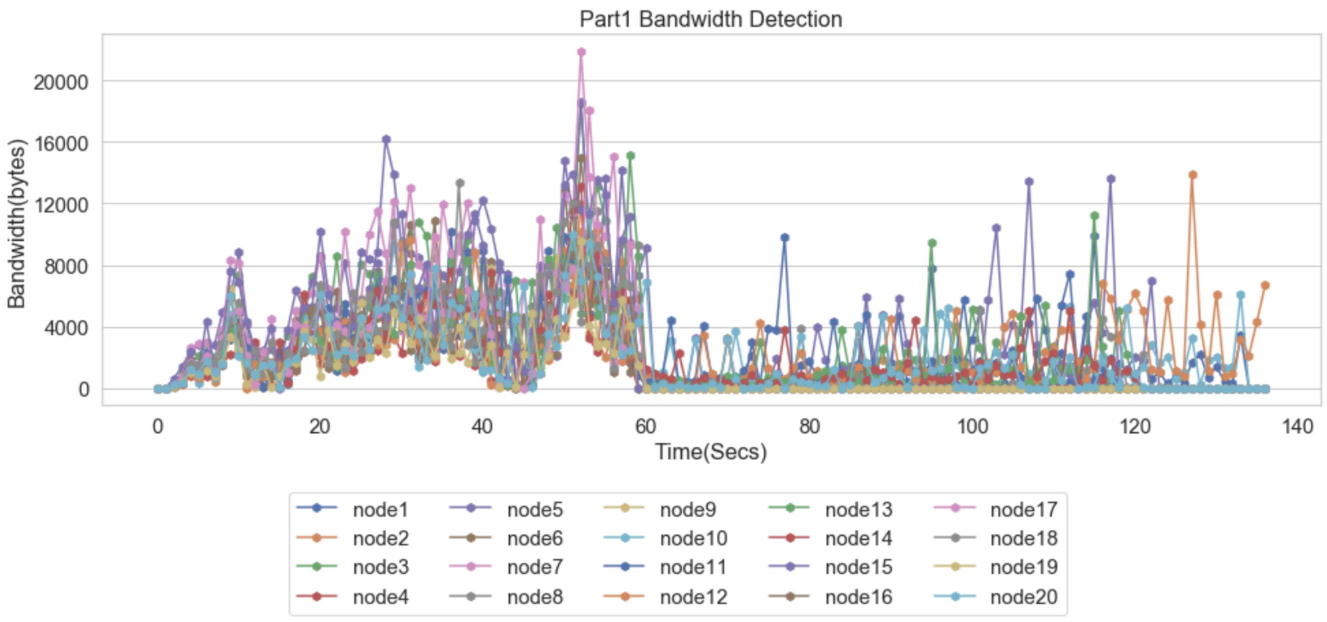

Figure 2: Part1 Bandwidth Detection of 20 nodes where THANOS snaps his finger after 60 seconds

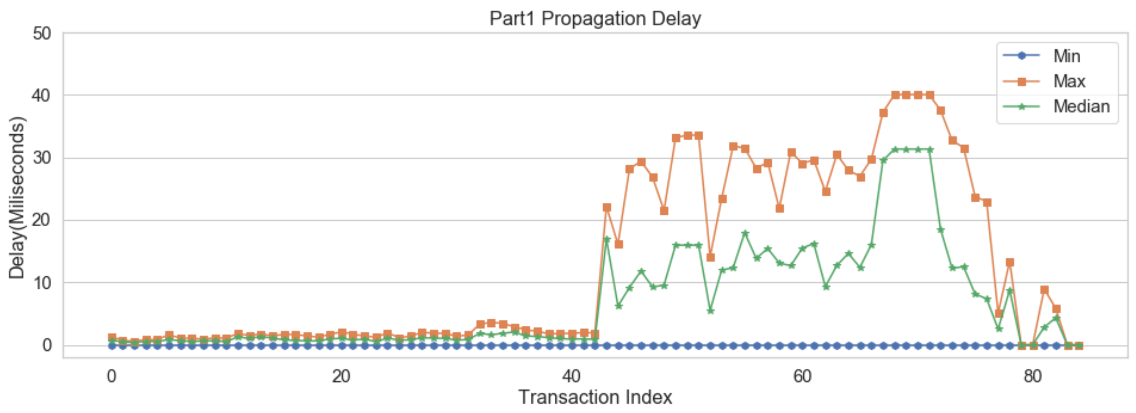

Figure 3: Part1 Propagation Delay

a node discovers another node, it logs its IP and its instance ID in a global structure named `IP`. This structure is then checked in our `gossipSend` threads intermitently. Everytime it detects that a new `(IP, instance)` pair, it will create a new `NODE` message and send it to its corresponding `gossipReceiver` thread. This message is then turned into an `INTRODUCE` message by the `gossipReceiver` thread and from that point the `gossipSendHandler` will take care of spawning new threads. This will allow us to maintain connectivity very well.

We argue that this is very robust because whenever a node (A) joins the system the service will introduce it to three other nodes (B, C, D). The three other nodes will then notify all its N number of neighbors which will then proceed to connect to the new node (A). In our case

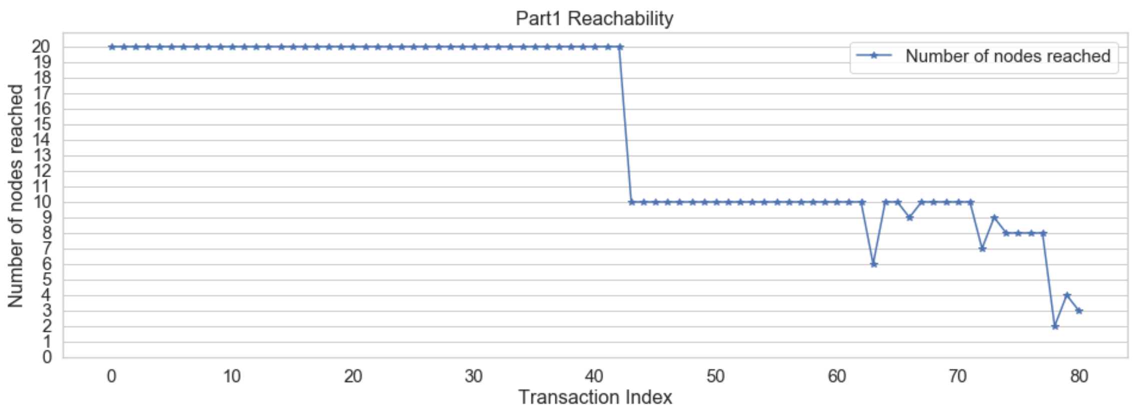

Figure 4: Part1 Reachability

because we do not bound the number of maximal connections, this will theoretically connect to all the nodes in the system thus leading to essentially a fully connected network. The downside to not limiting the number of connections with our scheme is that it will spawn an excessive amount of threads thus slowing down execution.

4.1.2 Building and Sending Blocks

In our system, we designate the number of transactions in a block to be a maximum of 2000. This value was a given. To determine what transactions to put in a block, we keep a globally accessible account dictionary. This account's keys are an integer value representing the account number and its value is the amount of money in each account. Account 0 is set to a constant 1000 dollars while all other accounts are initialized with 0.

Our block has the following format

```
PREVHASH h BLOCKHEIGHT bh TRANSACTION t id s d a TRANSACTION ... SOLUTION
```

`h` represents the hash of the previous block. `bh` represents the block height of the block. This is followed by a list of one or more transactions. Finally this is followed by the solution of the block provided by `mp2_service.py`

To determine if a transaction can be in a block we first check if the transaction lowers the account's balance to less than 0. If it does we do not add it to the current block but delay it to a later block. Otherwise we use the following logic to determine how many transactions go inside a block.

```
if len(receivedTrans) > 0:
    numTxInBlock = min(len(receivedTrans), MAX_TX_PER_BLOCK)
    genBlock(random.sample(receivedTrans, numTxInBlock))
```

This gives us a value ranging from 1 to 2000. However as discussed above, not all the transactions will be placed inside if they create a negative balance. These are returned to the received transaction queue to be used later.

One thing of note is that we do not pass these messages in any form of chronological order (received or timestamp). We believe this is a more efficient method than passing the messages in chronological order. Due to the decentralized nature of the system the following transactions may be received by different nodes.

1. From: 0, To: 1, Amount: 100
2. From: 0. To: 2, Amount: 100
3. From: 1, To: 2, Amount: 100
4. From: 2, To: 3, Amount: 100

If the transactions above were to be received in the order they were sent in (as shown above) there would be no problem putting all 4 of these transactions into a block. Suppose (0,1,100) and (2,3,100) were passed to node A and (0,2,100) and (1,2,100) were passed to node B. Node A is unable to create block. Node A also gossips these transactions to node B. Node B now holds the following list of transactions received in order.

```
[(0,2,100), (1,2,100), (0,1,100), (2,3,100)]
```

If B were to cycle through this list from left to right, it would still not be able to create a valid block consisting of all 4 transactions. This would occur forever as the ordering of the transactions never changes. However if B were to randomize the ordering of transactions before input we could theoretically have the following input transaction list.

```
[(0,1,100), (1,2,100), (0,2,100), (2,3,100)]
```

This would be a valid list of 4 transactions if processed in that particular order. As such it is more probablable that we are able to create larger blocks by randomizing the order of instructions before passing them to the block generation procedure

One thing of note is that due to the randomization of the ordering of transaction and gossiped messages, it is highly possible that a transaction exists in a block with a transaction that occurred much later in global time. While it would not affect the final balance of the accounts, it is possible to delay a transaction for a significant amount of time.

Once a block has been crafted it will be sent to a global queue of blocks we need to `SOLVE`. The `serviceThread` will then periodically send these out and wait for a solution. Once solved, these blocks will then be gossiped out to all of a node's neighbors who in turn will either ignore it or pass it on.

4.1.3 Life-cycle of a Block: Travelling, Verification, Chain Splits, and Confirmation

Once created by a node, a block will be gossiped out to other nodes. The receiving node has its own current block height which will be compared with incoming block's height. If the block's height is larger than its own, it will attempt to verify it by adding it to the queue. Otherwise it will ignore it and continue mining on its own chain.

If the node's block height is less than (40) the incoming block's (41), it will stop mining. Furthermore, it will add the block to a verification queue which is globally accessible. The `serviceThread` will then periodically send these out to the server to verify. If the verification fails, the node will ignore it, and continue mining on its current block height. If verification succeeds, then it will update the globally accessible `solvedBlocksDictionary`. Moreover, it will also request for the previous block from the same neighbor that sent the larger one to it (request 40 after receiving 41). This is because we may have been on a chain split that was longer than one block. The node then will procedurally count down the block height and verify blocks until it finds a previous block which matches the hash of the currently verified block (eg. 39 was just verified 38 already exists in the solved blocks and matches 39's previous hash). However, if it does not find a solved block matching the previous hash of a block retrieved from its neighbor, the node will then rebuild the entire chain from scratch. Once the chain has been rebuilt, the node will then rebuild the balances with all the solved blocks so that it can use the most updated information to create the next block

As we briefly mentioned above, it is possible there are two chains with the same transaction in different blocks. This occurs because we gossip messages between blocks. Therefore is possible to have the situation as follows

- Node A is building block 35 with transaction (0,1,100)
- Node B has built a block 34 with transactions (0,1,100), (5,6,7)

Node A finishes creating block 35 with transaction (0,1,100) and node B receives it. After node B has verified A's block 35 it will then proceed to check its own blocks to see if it has any overlapping transactions. If it does, those blocks need to be invalidated and have **non-overlapping transactions**, in this case (5,6,7), returned for block queue. This means that the transactions have been essentially "uncommitted" and now available for another block during generation. This is another way that our system may delay any transaction by a significant amount of time.

As this is decentralized, whoever is leading the chain is essentially the one who decides on the consensus of what is the next block. A block is suitably "confirmed" if it lives far enough down the chain that it should not be possible to override it without a significant amount of work. In our case, this would represent a significant amount of time for block generation and verification. In the mean time other nodes will continue increasing their own block height, thus making alterations to the chain considerably difficult. For Bitcoin, this safe "depth" is considered 6 blocks deep. However, because our block generation rate is so high (tens of seconds vs 10 minutes in Bitcoin), we require a much larger number here for this to be considered safe.

4.1.4 Handling Failures

We do not handle failures of connections explicitly (allowing them to end with a broken pipe). This is fine because a thread that sends messages into the void does not do any harm (for example a miner gossiping a new block it discovered). Moreover, because we are using TCP, we are guaranteed that a message will eventually arrive. This is because TCP is a reliable transmission scheme that provides re-transmission. Therefore, we can make the assumption that packets will not be lost due to a faulty transmission layer but due to a node failing or its application level software crashing.

However we do handle the case where a node A is trying to rebuild its chain from node B and node B fails. In this case we have timeout value of 10 seconds. If it times out in 10 seconds then it resumes mining on its own and will eventually send out an invalid block which will then promptly be overwritten by other live miners with a larger block height. We argue that this does not create any issues due to the two cases listed below.

1. Node A is only behind by one or two blocks. This means that other than the most recent blocks, it does not have any broken chain in the middle and therefore will not cause a node to have incomplete data.

2. Node A is severely behind while building its chain. In this case, node A will eventually produce a block which is significantly smaller in block height than other live nodes. This will trigger a rebuild once again. For example, while rebuilding block 30 from Node B, B crashes. However, node B and C were already on block 50. Therefore when node A times out and produces node 31 it will immediately be shut down with node C's 50th block thus not producing any permanent blocks.

4.1.5 Logs, Graphs, and Analysis

1. How long does each transaction take to appear in a block? Are there congestion delays?

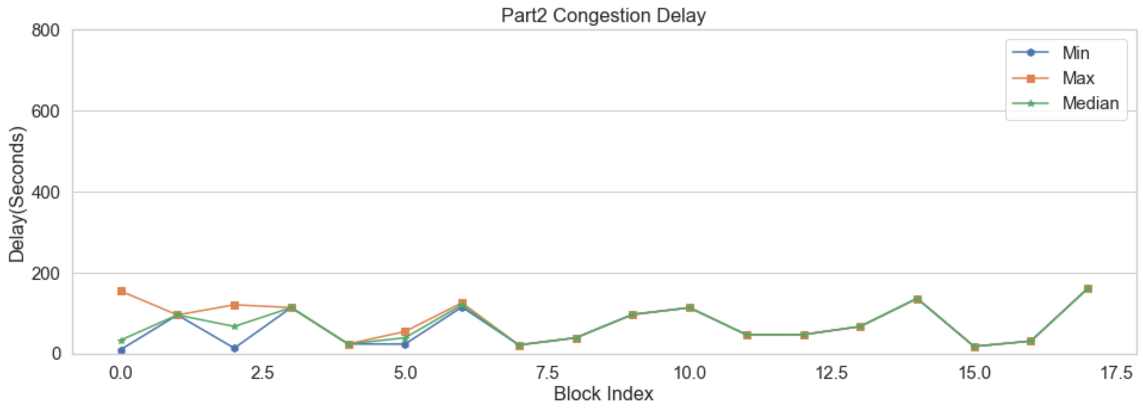

Figure 5: Part2 Congestion Delay

When transaction message is converted into blocks, it takes some time because proof-of-work has to be done. In part 1, there are 60 transaction indices. However, That number reduced into 17 because of the propagation delay.

2. How long does a block propagate throughout the entire network?

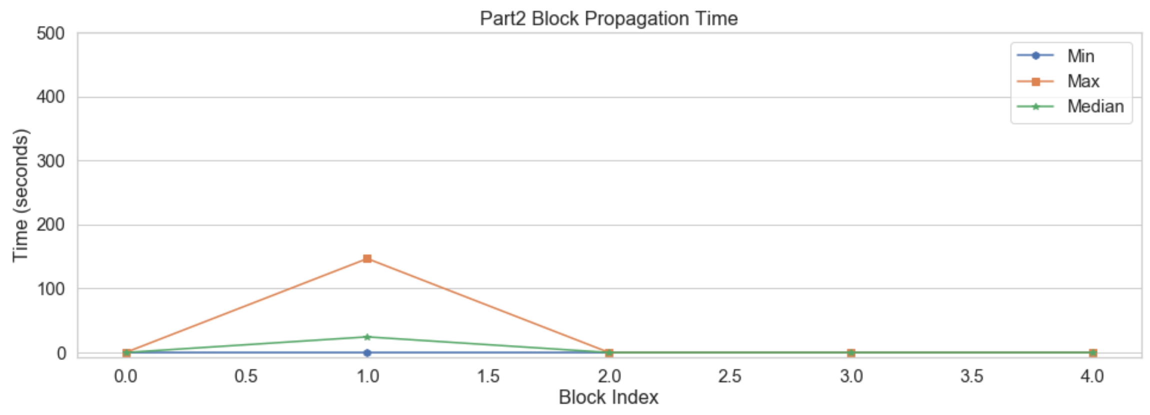

Figure 6: Part2 Block Propagation Delay

Based on the figure 6, it usually takes 40 seconds. Unlike transaction messages, blocks require much more time to propagation entire network.

3. How often do chain splits (i.e., two blocks mined at the same height) occur? How long is the longest split you observed? (i.e., smallest distance to least common ancestor of two nodes)

   In our experiment, chain split only occurred once. Chain split is measuring the height of the blocks. When the height of the blocks are equal at the same moment, chain split occurs.

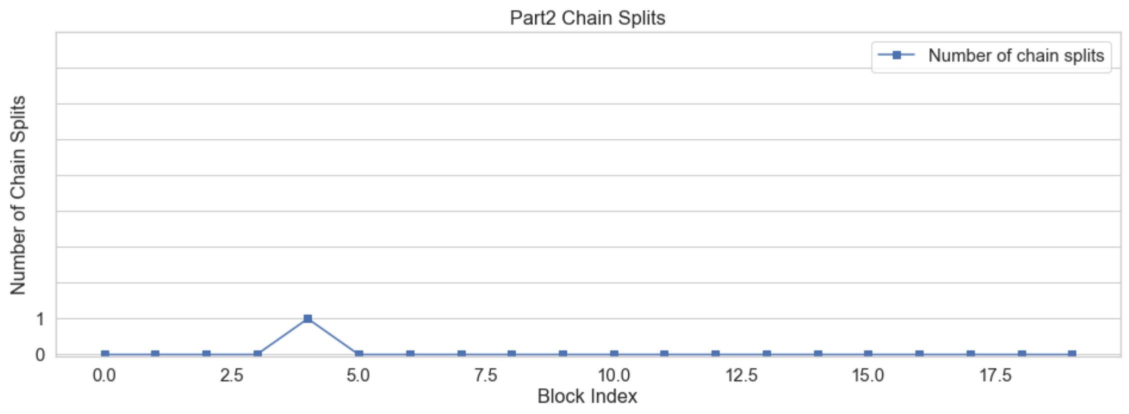

Figure 7: Part2 Chain Splits

4. What information is logged and how you used this to generate the graphs. Please make sure that the logs you used in your experiments are checked into the git repository. If you wrote any scripts to analyze the logs please include them in the repo and describe how they work.

   All of the logs from the experiments are saved in the `/log/` directory. Bandwidth, delay, message arrival time, block arrival time, message ids, and block ids have been logged.