Blockchain Simulation Project Report

Ding Tao Liu *
Department of Computer Science
University of Toronto
Student #: 1001287001, UTORid: liuding1

Hao Ze Wu *
Department of Computer Science
University of Toronto
Student #: 1001325390, UTORid: wuhao30

Jiahuang Lin *
Department of Computer Science
University of Toronto
Student #: 1002745238, UTORid: linjiah6

1 Introduction

Blockchains are a core technology of cryptocurrency. Two of the features of blockchains are the possibility to manage ledger information without a centralized system and the difficulty in tampering previously obtained data. Due to these features, blockchains are used in many cryptocurrencies, and their applications are extensively investigated. The popularity of blockchains is natural, as they have the potential to provide desirable features by replacing centralized communication architectures. However, with blockchain systems being decentralized, keeping track of the changes that happen within a network become hard as there is no centralized point of contact. For example, in a case where consensus may not be reached quickly, one set of nodes may have one idea of a longest chain, another set of nodes on the other side of the network may have another idea of a longest chain. In such a case, we would like to observe the local views of each set of nodes, and see how it affects consensus. Other issues that can arise such as double attacking, and forking are hard to visualize in a decentralized environment.

Thus, we have decided to create a simulation in order to confirm various theoretical assumptions about Bitcoin [6]. This simulation is created in Python, and simulates a network of interconnected nodes. We implement a gossip algorithm [3], and an event scheduler. Finally, we use our simulation to observe how bitcoin [6] works and see how it compares to the theory presented in the white paper.

2 System Assumptions

We simulate a blockchain system with n_{total} nodes that are connected in a gossip network [3]. Each node has an average of $n_{neighbour}$ random neighbours. The gossip network [3] is initialized once at the beginning of each simulation, and each node has fixed neighbours throughout the run. Our model assumes that there is a fixed network latency L_{fixed} between all pairs of nodes, but each node has a potentially unique upload $L_{i,upload}$ rate.

Our model assumes that all transactions are valid and available to all nodes. This simplifies the simulation by omitting the need to simulate block validation as well as transaction generation and relay. We assume that block generation intervals are independent and follow an exponential distribution. When an honest miner discovers that the block it is working on is stale, it starts over on the new best chain.

*Link to our implementations on Github

3 Implementation

3.1 Simulator

Our simulator is a single process Python script that takes in a list of parameters, executes the run, and then outputs a log of events, node statistics, and graphs of the final blockchain state.

There are three main classes in our implementation:

- *Simulator:* Main simulation loop that initializes nodes and network and logs results
- *Node:* Handles block generation and propagation
- *Network:* Configures network topology and latency

Each simulation is run for a set amount of time T_total. Each Node has its own event queue of block creation and receive events. Initially, the Simulator queues up a block creation event for each node. In each iteration, the Simulator queries each node for the time until its next event. The Simulator then advances time by the minimum time interval across all nodes. Because each node reports a time interval instead of a time stamp, this method ensures that no events are skipped. Each node updates its timestamp by the minimum interval, and processes any occurring events.

Here is a use case of this simulator. Let's say we want to simulate for 3 hours, with 10 nodes, and a block generation time of 10 minutes:

- At the beginning, our nodes will be initialized with a single create block event, sampled randomly from an exponential distribution. From there, we calculate the first event that will happen. Say this happens on $Node\ A$ 10 seconds from now.
- We forward our time to 10 seconds in the future, and create the block i for $Node\ A$. This block i will then be propagated to $Node\ A$'s neighbours, adding a process block event to all the chosen neighbours in their event queues.
- The simulator will then query for the next event to happen. Say it is a process block event on $Node\ B$. In this case, the simulator will advance time to that point and $Node\ B$ will process the block and add it to it's known blocks.
- Then, the simulator queries for the next event to happen. This timestamp increment along with an event processing will keep happening until the time passes 3 hours.
- Once there, the simulation stops and will draw graphs for all the nodes, showing what each node believes to be the chain, and then show a bird's eye view of the *true* longest chain through the master node.

Tuning the parameters given to the simulation will result in different test times or different configurations of the graph. This allows us to test and understand what is happening on the blockchain.

3.2 Nodes

Every node has an event queue which allows the simulator to choose the next event to process. Each node also is a *attacker node* or a *honest node*. Every node also has its own directed acyclic graph, which represents the blocks it knows. This is crucial to showing how one node may understand one chain as the longest chain while another node has a different chain.

3.3 Gossip Network

Our base gossip network [3] is modeled with an Erdos-Renyi graph, where each node has an average of $n_{neighbour}$ number of neighbours. More precisely, for nodes i and j, there is an edge between i and j with probability $n_{neighbour}/n_{total}$. We used the *NetworkX* [4] package to create and manipulate the gossip network [3]. Each node in the network graph represents one miner, with an optional attribute propagation factor p. In some variations of the gossip protocol, each node gossips to a random subset of its neighbours. In our model, p is the percent of neighbours the node will gossip to. By augmenting the base gossip network [3], we can examine the effect of unique network topology on blockchain consensus. Some configurations that we have implemented include:

- *Isolated nodes:* This is achieved by adding additional nodes with a very small number of neighbours to an initialized network.
- *Hubs:* This is achieved by selecting a few nodes and adding 20% - 30%

3.4 Block Generation

We assume that the time T_j for node j to generate a block follow an exponential distribution. Each node j is initialized with a hashpower $0 < H_j < 1$. The sum of H_j for all nodes is 1. To achieve a desired average system block time of T_{avg}, the expected block time $T_{avg,j}$ of each node can be calculated as:

$$1/H_j * T_{avg}$$

The rate λ_j for the exponential distribution from which T_j is sampled from is $1/T_{avg,j}$.

At the beginning of a simulation, the event queue of each node is initialized with a generate block event. A generate block event has *two* components:

1. *Timestamp*. The timestamp for a generate block event is created by sampling a T_j from node j's λ_j, and then adding T_j to j's local timestamp.

2. *Block*. The Block has a unique id as well as a parent id that points to its parent on the block graph.

Thus, creating a generate block event is synonymous to a miner finalizing the fields and transactions of a block as it begins to mine the PoW [6].

Whenever a node dequeues a generate block event, it first checks if the parent of the event Block is still the last block of a longest chain. If not, it will discard the event. Otherwise, the node will add the block to its local blockchain state prepare to gossip the block to its neighbours. Finally, the node will queue up a new generate block event. If the node receives a block that extends the chain that the node is currently mining on, the block in progress will be abandoned.

Because we reset generate block events when their parent is no longer the last block on the best chain, the average time to generate a block among all blocks is the minimum of $_j$ for all nodes j. Since each T_j follows an exponential distribution, the min of all T_j also follows an exponential distribution with:

$$\lambda_{global} = sum(\lambda_j) \quad \forall j$$

which ends up being T_{avg}. Thus, by setting T_{avg}, we can set the average global block generation time.

3.5 Block Propagation [2]

Each node keeps track of its own local blockchain state as it generates and propagates blocks. Whenever a node discovers a new block (through generate block or receive block events), the block is added to the node's gossip queue.

A receive block event contains a timestamp and a Block object. Processing the receive event involves adding the block to the nodes local blockchain state. As discussed in the block generation section, this may alter the node's view of the longest chain, and force the node to abandon the in-process block and to start working on a new one. If due to gossip delay the node has not yet discovered the parent, the block will be a disjoint node in the graph until the parent is added.

Whenever an event is processed, the node will gossip all blocks in its gossip queue to a subset of its neighbours. To avoid circulating blocks in the network forever, events are only gossipped to neighbours that have not yet seen the block. Gossiping is done by enqueueing a receive block event to the event queue of the neighbour. This means that a node may receive multiple events for the same block at different times from its neighbours. To optimize the performance of our simulator, whenever a block is discovered from an event, the node will purge any remaining events with the same block from the event queue.

The timestamp of the receive event is calculated by adding the fixed network latency L_{fixed} and an L_{upload} latency to the receiving node's time. For example, let network latency L_{fixed} be 12 seconds and node p wishes to gossip a block to 10 different neighbours. P also has an upload bandwidth of

$L_{upload} = 5MBps$. Assuming each block is 1 byte, the first 5 neighbours will receive the event at time $T_{local} + 12s$, and the latter 5 neighbours will receive the event at $T_{local} + 12s + 1s$ (due to only being able to upload 5 blocks per second).

3.6 Consensus Mechanism

Our model uses a Proof-of-Work [6] system for block generation, and we implemented two consensus mechanisms: Nakamoto [6] and GHOST [7].

Each node maintains a graph of blocks that it has discovered. Whena generate block event is created, a block in the node's local state needs to be chosen as the parent of the block-in-progress. Under Nakamoto consensus, we choose the block at the end of the longest chain as the parent. If there are several longest chains, our tie breaker picks a parent randomly. Under the GHOST protocol, the local blockchain state graph is traversed starting from the genesis block. At each fork, the block with the heaviest subtree is chosen. If a block has multiple child nodes with the same subtree size, then a child is chosen at random for the traversal. The traversal ends when a leaf node is reached, which becomes the parent block.

3.7 Double Spend Attacker

The node implementation allows more than one appending mechanism. As opposed to the honest node mining strategy, a double spender attack works by choosing an "attack" node and forking from there. In our implementation, we can specify how long the node waits before attacking, which node to attack when starting to attack, and finally the hash power of this node. An example of double spend attack in our simulation would happen like this. Suppose a node has 50% hashpower and blocks are confirmed after 6 subsequent blocks are added. The attacker wants to double spend on block 5, and would wait until the main chain is 11 blocks long. The 5_{th} block is now 6-confirmed, and the attacker picks block 4 as the *attack block* and create his own fork from there.

In this case, the attacker node behaves like an honest node until the chain grows to 11 blocks. The attacker starts to mine a fork from block 4. Then, the race begins between the attacker and the honest nodes. If the attacker chain becomes longer than the honest chain, the attacker wins, and honest nodes will join the attacker chain as it is now the longest chain. This effectively reorganizes the entire blockchain, allowing a previously confirmed block to be invalidated as the attacker has control of the longest chain.

4 Results

For figure 1, we ran our simulator with the following parameter setting:

- $N = 150$
- $N_{neighbours} = 10$
- $L_{fixed} = 5s$
- $L_{upload} = 5$
- $T_{avg} = 10$

This is a normal run without attackers. We see that the number of blocks generated increases linearly with respect to the elapsed hours, which is expected.

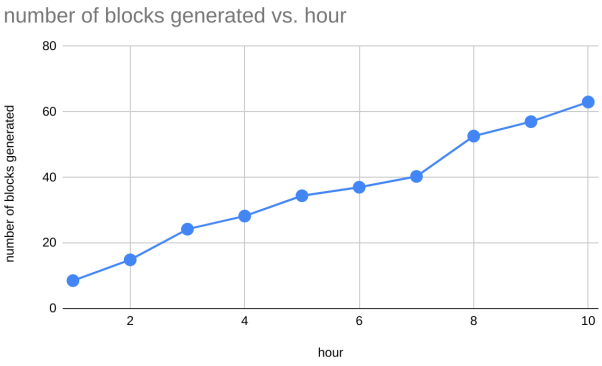

Figure 1: Number of blocks generated vs. time in hours

For figure 2, we use the same parameters but keep the time constant to 5 hours. This run is also no attackers.

As the latency across nodes increases, we see that the percentage of stale blocks also increases proportionally. Note that at delay around 200 the line plateaued a bit. This is due to the non-deternimistic characteristic of block generation distribution: the block generation is quite sparse during this time period. Overall we still see a linear growth pattern which is expected.

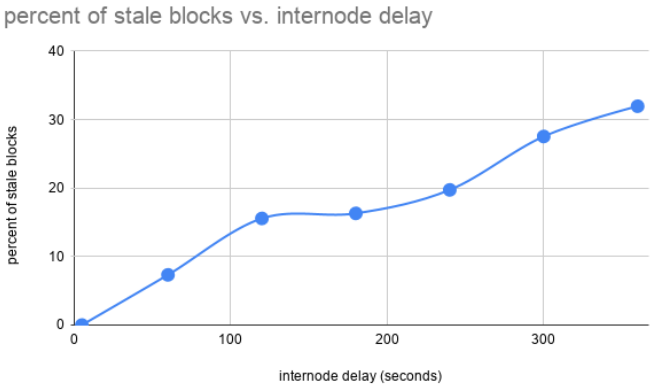

Figure 2: Percent of stale blocks vs. inter-node delay

As we mentioned before, our simulator also supports plotting the master chain. Figure 3 is again a 5 hours run without attackers. Each circle in the graph represents a block on master chain and the yellowed colored circles which assembly a path represents the current longest chain. We set to latency to be 200 which is relatively high with an intention of seeing forks.

It is known with comparing with *Longest Chain consensus* mechanism, *GHOST [7] consensus* mechanism is harder for attackers to double-spend. As attackers in *GHOST* has to make its sub-tree the heaviest among all others. To illustrate that, we plotted two simulation runs.

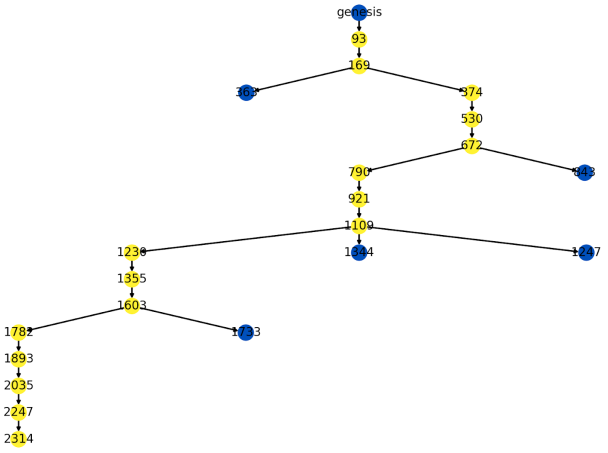

Figure 3: Master chain view with 5 hours simulation and 200 second latency

Figure 4 and figure 5 explores such difference. Figure 4 shows an attacker case for GHOST consensus with 55% hash power. We set the latency to be as high as 5 mins to buy more time for the attacker to perform the attack. On the graph, the red blocks are the blocks generated by the attacker that intends to double spend. In this case the attacker fails.

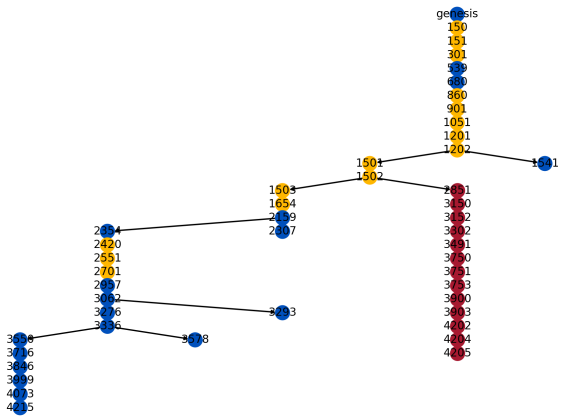

Figure 4: GHOST consensus, 55% attacker hash power with 5 mins latency

However, if we take a look at figure 5, which is a run of longest chain consensus where the attacker has only 30% of hash power but with a same high level of latency (5 mins), we see that the attacker succeeds with only 30% of hash power. Such result meets our expectation that GHOST is harder for double-spend attacker to attack.

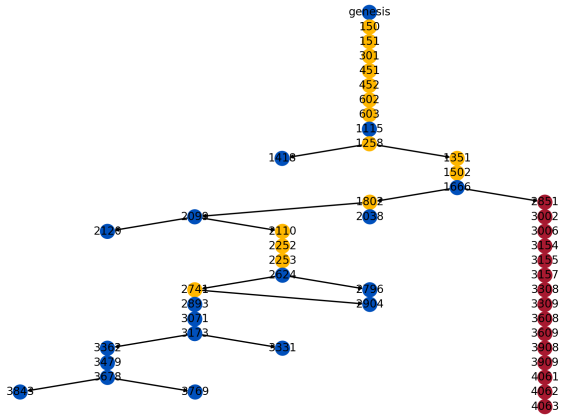

Figure 5: Longest chain consensus, 30% attacker hash power with 5 mins latency

5 Future Work & Challenges

We would like to implement other consensus mechanisms such as *Conflux* [5]. Furthermore, other consensus mechanisms like *Proof of Stake* [1] would be nice to simulate.

In terms of challenges, a lot of work was done to speed up our implementation. Our first implementation had simulation runtimes where one second of simulation time took longer than one second of real time. This led to optimizations such as our event scheduler picking particular nodes to process events rather than picking all of our nodes. However, more optimizations could be made perhaps in a parallelized environment to increase efficiency in executing operations.

Finally, we would like to implement better visualizer than just graphs printed at the end of the execution. A real time showcase of how blocks are added or how the longest chain grows in certain nodes would be really nice to have to increase visibility into the blockchain. We could also report better statistics like average propagation time or number of forks between nodes. This again would align with the goal of increasing visibility of the blockchain.

References

[1] Chris Chinchilla. 2015. *A Next-Generation Smart Contract and Decentralized Application Platform.* `https://github.com/ethereum/wiki/wiki/White-Paper`

[2] C. Decker and R. Wattenhofer. 2013. Information propagation in the Bitcoin network. In *IEEE P2P 2013 Proceedings*. 1–10. `https://doi.org/10.1109/P2P.2013.6688704`

[3] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '87)*. ACM, New York, NY, USA, 1–12. `https://doi.org/10.1145/41840.41841`

[4] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.

[5] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. 2018. Scaling Nakamoto Consensus to Thousands of Transactions per Second. arXiv:cs.DC/1805.03870

[6] Satoshi Nakamoto. 2009. Bitcoin: A peer-to-peer electronic cash system. `http://www.bitcoin.org/bitcoin.pdf`

[7] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*. Springer, 507–527.