COMP 415 Research Project: Blockchain Simulator Final Report

Ege Erdogan
Koç University
eerdogan17@ku.edu.tr

Can Arda Aydin
Koç University
canaydin17@ku.edu.tr

***Abstract*—As blockchain-based systems see wider adoption, it becomes more critical to ensure they are reliable, secure, and efficient. Among possible methods of evaluating blockchain systems, running simulations stands out as a low-cost and scalable alternative. However, many of the existing simulations have various shortcomings that yield them insufficient for a wide-range of scenarios. In this work, we present a new blockchain simulator, attempting to address the shortcomings of the existing simulators. We explain our design and implementation in detail, and to validate our simulator, we compare the results we obtained from our simulator with the real-world Bitcoin blockchain.**

I. Introduction

First introduced in 2008 as part of the Bitcoin cryptocurrency [1], a blockchain essentially enables a dynamic set of distrusting parties to reach consensus on a history of transactions. Although the hype around blockchains seems to shadow their actual use cases, they have been proposed as building blocks of novel solutions in different problem domains such as supply chain management [2], electronic voting [3], and finance [1], [4]. The decentralized nature of blockchains that ideally does not require a trusted centralized entity makes them especially appealing.

As the saying goes, with great power comes great responsibility. If blockchain systems are to be used for purposes that are highly crucial for the well-being of a society, it is of critical importance to ensure that those systems are reliable, secure, and efficient.

Therefore, methods and tools that enable evaluation of blockchain systems' security and performance can prove to be very valuable. Monitoring, benchmarking, experimental analysis, and simulations are four main methods that allow empirical analysis of blockchain systems [5]. Among these methods, performing simulations stands out as an accessible and low-cost alternative. While all the other methods require direct access to the system under evaluation, simulations can be ran independent of the actual system. Furthermore, a well-designed simulation could examine a wide range of scenarios in a fraction of the time it would take for the real-world system.

However, most existing simulators have various shortcomings that make them unfit for the task. As the authors conclude in the only systematic review of blockchain simulators to date, there is no universal simulator that can be used to evaluate a wide range of scenarios [6]. Since most simulators are designed with a specific but limited purpose (e.g. to demonstrate an attack [7], or test a certain property of the system [8]), they make unrealistic assumptions to simplify layers that are unrelated for their purpose. Furthermore, most simulators present little, if any, customization opportunities. For this purpose, a well-designed simulator could not only be an achievement in and of itself, but also pave the way for new research concerning the security and performance of blockchain systems.

For these reasons, we had decided to build a blockchain simulator for our course research project, hoping that we could address some of the shortcomings of the existing simulators. Although it is difficult to complete such a project in the matter of a few weeks, we have made more progress than we had expected, and hope that we can further build upon our existing work to build a more comprehensive simulator. So far, we have

- designed and implemented a simulator core that captures the main elements of a blockchain system, namely a P2P network of nodes capable of exchanging messages with each other,
- used the core simulator to implement a Bitcoin simulator,
- tested the Bitcoin simulator in various settings, and compared the results produced by the simulator with real-world data.

In the rest of the report, we will first give the necessary background on blockchains and Bitcoin, and briefly examine selected works from the literature on blockchain simulators. Afterwards, we will describe our design and implementation in detail, and present the results of our comparisons with real-world data. We will conclude by discussing our potential future work.

We provide a tutorial in the appendix, guiding the user through configuring the input parameters, running a simulation, and analyzing the results. Finally, the source code of the simulator can be found at http://github.com/simbadt/simbadt and the documentation can be found at https://simbadt.github.io.

II. Background

In this section, we present the necessary background on blockchains, using the Bitcoin blockchain as our primary example.

A blockchain is a distributed append-only ledger of *blocks* that contain *transactions* [9]. As opposed to *permissioned* blockchains (e.g. Hyperledger Fabric [10]) that function with a predetermined set of nodes, *permissionless*, or public,

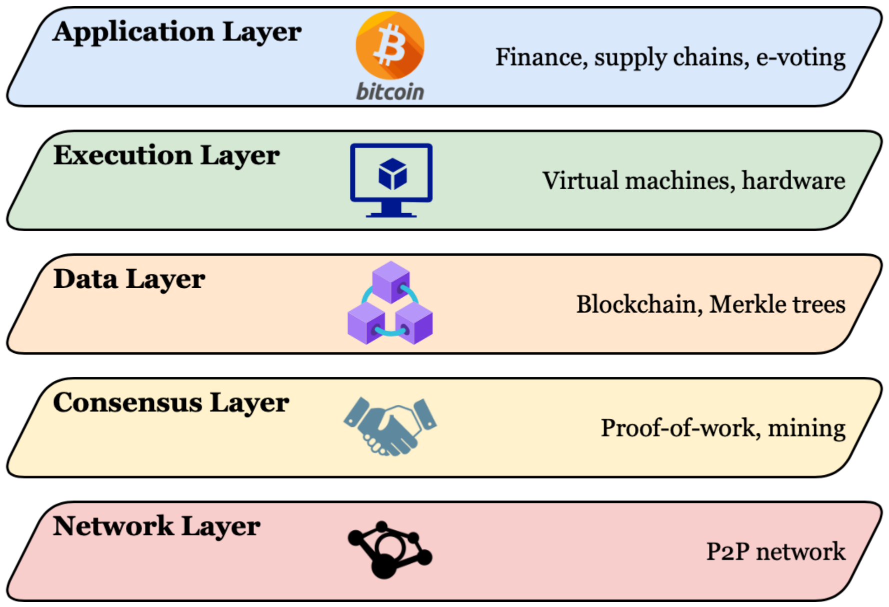


Fig. 1: Five-layered organization of a blockchain system, with application, execution, data, consensus, and network layers from top to bottom.

blockchains (e.g. Bitcoin [1], Ethereum [4]) allow the participants to leave and join the network without any authorization. The primary use case of blockchains today is as shared ledgers for cryptocurrencies. The blockchain acts as a ledger while the currency itself is used to incentivize the participants into following the protocol.

A blockchain system can be thought of as consisting of five layers. From higher-level to lower-level, those are the *application*, *execution*, *data*, *consensus*, and *network* layers. Since the application and execution layers, corresponding to the problem the system is attempting to solve, and the platform its running on, are not relevant for our purposes, we will examing the data, consensus, and network layers in detail.

A. Data Layer

The data layer, also called the *storage* layer, corresponds to how various types of data are stored and managed on the blockchain. A critical property of the data stored on a blockchain is that it is immutable. This is ensured by storing a *hash pointer* in each block that contains the hash of its previous block. Thus, tampering with one block will require appropriately modifying hash pointers of its successors. Since that would essentially mean recomputing the proof-of-work of all the following blocks from scratch or finding a collusion for a hash function such as SHA-256, it is an infeasible task.

Decisions concerning the organization of data storage on the blockchain should also take into account performance. For example, in Bitcoin, transactions within a block are arranged in a Merkle tree [11]. This makes it easier to check whether a transaction is included in a block or not, eliminating the need for a full pass over the transaction list.

Finally, to manage the account balances, Bitcoin uses the *unspent transaction output* (UTXO) model. In this model, each transaction aside from the *coinbase* transaction described in the next section, spends a set of transactions as input, and outputs a new set of *unspent* transactions. Thus, the balance of an account is the total amount of unspent transaction outputs sent to that account.

B. Consensus Layer

The consensus protocol constitutes the core of a blockchain system. It is the protocol by which participating nodes agree on modifications to the blockchain [12]. Consensus protocols have been a major area of study within distributed systems for a long time, with results such as the practical byzantine fault tolerance protocol [13]. Two main properties that a consensus protocol should satisfy are *liveness* and *consistency*. Liveness ensures that requests from honest nodes are eventually processed, while consistency ensures that all honest nodes make the same decisions.

In an open and decentralized blockchain system without any trusted party and no established identities between the nodes, the problem of consensus is impossible [14]. Bitcoin's key breakthrough was its method of incentivizing the nodes towards honest behavior, assuming that the nodes are *rational* (i.e. they follow the course of action that maximizes their expected benefits). The underlying cryptocurrency acts as a source of value incentivizing the participants and reinforcing the robustness of the system, while the system's security and robustness in turn reinforces the currency's value. Thus, a valuable currency would not be possible without a secure consensus mechanism, and a secure consensus mechanism would not be possible without a valuable currency.

In Bitcoin, to propose a block to be added to the main blockchain, a node (called a *miner*) should find a value for the *nonce* field of a block such that the hash of the block falls below a target value. Since there is no better method to finding this nonce value than a brute force search, the probability that a miner will find an appropriate value first is equal to the ratio of its computing power to that of the entire network. The nonce value found acts as a *proof-of-work* (PoW).

Bitcoin miners are incentivized towards proposing valid blocks through *mining rewards.* A miner can add a special *coinbase* transaction to a block it finds, rewarding himself with a pre-determined amount of currency. Since the reward will be meaningless if its block is not on the main chain, or is an invalid block, it is better for an honest miner to mine on the main chain. The mining reward is halved every 210,000 blocks, and will eventually reach zero, limiting the total supply of bitcoins. Miners are also incentivized by *transaction fees*, corresponding to the difference between a transaction's inputs and outputs. Although transaction fees have been relatively small compared to the mining rewards so far, an interesting question is how the network will behave once the mining reward is so low that the miners are mainly incentivized by transaction fees [15].

Since the block generation protocol in Nakamoto consensus is probabilistic, multiple miners can discover valid blocks building on the same block at almost identical times. This would give rise to a *fork* on the blockchain, as different nodes could have different views regarding the next block, depending on the order in which they receive the candidate blocks. In such a case, miners will randomly pick one of the blocks and mine on it, and the other blocks will remain as *stale* blocks. Therefore, an important property of Nakamoto consensus is that *the longest chain at any time may not be a prefix of the longest chain at a later time.* For this reason, we can only talk about the *probability* that a transaction is valid, rather than counting one as absolutely valid. However, since the probability that a transaction's block going stale will decrease exponentially as more blocks are build on top of it, following a "k confirmations" rule that waits for k blocks (generally 6) to be built upon a certain block is advised to validate transactions [16].

C. Network Layer

Although blockchain systems with structured network topologies have been proposed [17], Bitcoin operates on top of an unstructured, random network [18]. When a node joins the network, it contacts one of the DNS servers run by volunteers, and obtains an initial list of bootstrap nodes to connect to. The joining node than receives the entire blockchain if needed from a bootstrap node. It can then discover new nodes to peer with by asking its existing peers, or listening for advertisements. There is no well-defined protocol for leaving the network, and thus nodes can leave arbitrarily. Disconnected nodes are detected by periodic checks from their neighbors, and are eventually removed from the address pool.

Although the default Bitcoin client has a default connection pool size of 8 per node, Bitcoin nodes that accept incoming connections have 32 open connections on average [18]. Furthermore, even though the network is random, some nodes have far greater degree than others [19]. These nodes can correspond to gateways for large mining pools, or centralized wallet services.

In Bitcoin, transactions and blocks are transmitted through a gossip-like broadcast protocol. Traditionally, the exchanges were performed through `INV` and `GETDATA` messages. To prevent unnecessary network load caused by the redundant sharing blocks, a node would first send an `INV` message to its peers, containing the block header. Peers that did not have the specified block would then respond with a `GETDATA` message and would receive the block. More recently, proposals such as compact block relay [20] extend this logic to individual transactions within a block, aiming to reduce the bandwidth consumed by the network.

III. Related Work

In this section, we review a set of selected existing simulators, taking into account the features they support and the configuration opportunities they provide. Table I displays the programming languages the selected simulators were implemented in, while Table II compares their features with our simulator. We refer the reader to [6] for a more comprehensive and systematic review of blockchain simulators.

Paper	Language	Code
[21]	Python	https://github.com/maher243/Blocksim
[22]	Java	https://github.com/dsg-titech/simblock
[23]	Python	https://github.com/carlosfaria94/blocksim
[8]	C++	https://github.com/arthurgervais/Bitcoin-Simulator
[24]	Python	https://github.com/RoseBay-Consulting/BlockSim
[25]	Scala	https://github.com/i13-msrg/vibes/

TABLE I: Programming languages the selected simulators are implemented in, and links to their source codes.

A. Existing Simulators

1) Blocksim-Alharby [21]: Our first selected "Blocksim" was developed by Alharby and van Moorsel in Python. The last commit to the public repository was twelve days ago as of this writing, meaning that the simulator is regularly updated. The simulator focuses on the data and consensus layers of PoW blockchains, and thus makes simplifying assumptions regarding the network layer. For example, similar to many other simulators, they do not account for the geographic distribution of nodes and miners, or support an information propagation mechanism such as the `INV-GETDATA` protocol described above. For these reasons, it is difficult to use the simulator for analyses that are concerned with the network layer itself, or how various properties of the network layer affects other parts of the system. Another shortcoming of the simulator is that it assumes all nodes are honest. It is not feasible to program malicious node and miner behavior. For our simulator, being able to program in malicious miner behavior is of utmost importance. It is crucial that proposed attacks can be tested on isolated realistic environments so that insights can be gained on their real-world feasibility.

Features	Blocksim-Alharby [21]	Simblock [22]	Blocksim-Faria [23]	Gervais-Simulator [8]	Blocksim-Pandey [24]	VIBES [25]	**Our Work**
Block size distribution				✓			✓
Geographic network conditions		✓	✓	✓			✓
Information propagation mechanism		✓	✓	✓			✓
Compare real time and simulation time		✓		✓			✓
Consensus algorithms besides PoW							
Block interval distribution				✓			✓
Dynamic mining difficulty							
Transaction fees	✓					✓	✓*
Transaction modeling	✓		✓		✓	✓	✓*

TABLE II: Input and output parameters available for the selected simulators. Checkmarks correspond existing features. Empty cells either correspond to fixed values being assumed, or missing features ($*$: partially implemented).

2) Simblock [22]: Developed by Aoki et al. in Java and last updated four months ago, Simblock stands out as another somewhat regularly maintained simulator. Simblock aims to simulate the network and consensus layers, and provides limited support for consensus algorithms other than PoW as well, such as PoS. However, the main limitation of Simblock is that it does not simulate the data layer. There is no transaction generation or propagation mechanism and only fixed-size blocks are transmitted through the network. This limits the simulator's ability to simulate the network layer as well since the propagation of transactions is an important factor affecting the overall network load. Furthermore, not supporting transactions also implies that the simulator does not distinguish between miners and full nodes, since full nodes' role becomes meaningless without a transaction mechanism.

3) Blocksim-Faria [23]: The second selected "blocksim" was developed by Faria et al. in Python, and was last updated 12 months ago. It aims to simulate the network and consensus layers, and thus does not model transactions in detail. Despite containing a transaction generation mechanism, Blocksim does not take into account transaction fees. Although transactions fees are not necessary to simulate the network conditions, taking them into account can give rise to interesting potential use cases for the simulator. As further limitations, it assumes fixed block and transaction sizes, and does not support dynamic mining difficulty adjust- ments. It only supports PoW blockchains, specifically focusing on Bitcoin and Ethereum networks.

4) Gervais-Simulator [8]: Developed by Gervais et al. in C++, and last updated more than five years ago, this simulator is more detailed compared to many others, yet outdated. It simulates the network and consensus layers, disregarding transactions, but supports many configuration options for those layers. Geographic distribution of nodes and miners, and the number of neighbor connections of nodes and miners can be configured. This is also one of the few simulators that can compare simulation time with real time. Finally, an important reason we chose to include this simulator in our review was that it also demonstrates how simulators can be useful in generating insights about existing blockchain systems. The simulator was developed as part of a larger project analyzing the performance and security of PoW blockchains, and was used to analyze how features such as block interval and block size affected various properties of the systems.

5) Blocksim-Pandey [24]: The final one of the three selected "Blocksim"s was developed by Pandey et al. in Python, and was last updated around 2 years ago. The simulator is limited as it was designed to simulate only *private* Ethereum and Hyperledger blockchains. They employ a Gaussian transaction generation model, with transaction sizes sampled from another Gaussian distribution. They define a fixed mining time, and assume that all nodes are honest. This is a reasonable assumption since the simulator targets private blockchains, where participants are authenticated. Overall, Pandey et al.'s Blocksim is a useful simulator for private blockchains, as they also report that it has been used for decision-making in industrial settings. However, they do not fulfill our goals since our main focus is on public blockchains.

6) VIBES [25]: The VIBES simulator, implemented in Scala and last updated around 13 months ago, aims to simulate the data and consensus layers of blockchains. It is also distinguished from other simulators due to its support for adversarial miners. While most simulators, as described above, assume honest miner behavior, VIBES enables simulations of attacks such as selfish mining [7]. However, it has various shortcomings. First, its use of a centralized coordinator introduces a performance bottleneck for large-scale simulations. Furthermore, it only supports PoW-based consensus protocols, and it supports neither dynamic adjustments of mining difficulty, nor geographically distributed nodes/miners. Finally, VIBES assumes a fixed block size and a fixed block confirmation time. Even though it is a promising alternative compared to other simulators, these shortcomings hamper its potential. For example, due to the lack of a geographical node distribution, block propagation delays in VIBES consist of a fixed verification time, and a transmission delay sampled from a distribution. Simplifications like these make it difficult to analyze the network layer of a blockchain, since the geographical distribution of miners play an important role in which block gets appended to the blockchain.

B. Discussion

As the above analysis, and the systematic review at [6] shows, all existing simulators have various shortcomings that limit their potential use cases.

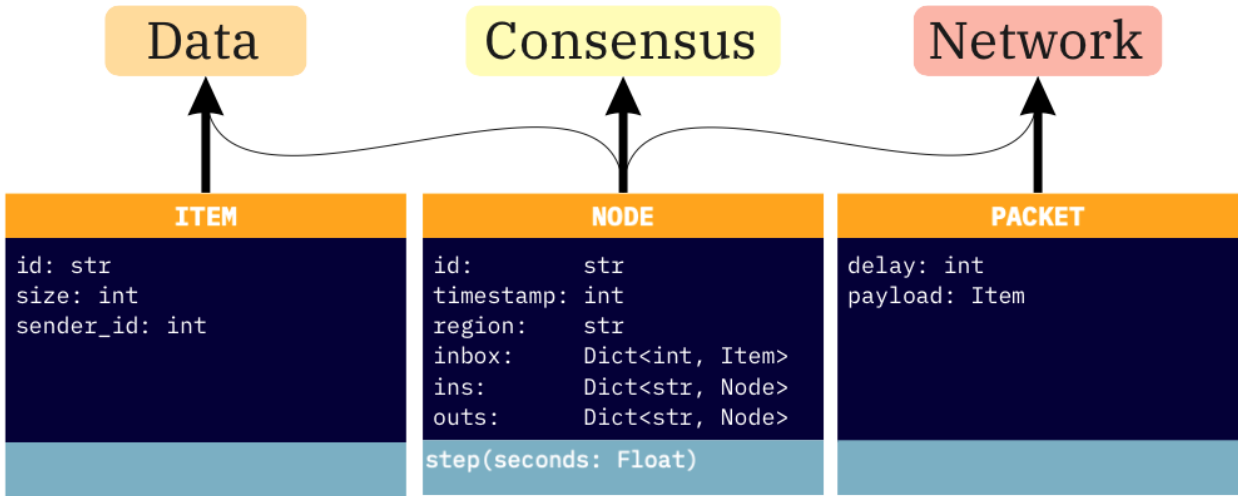


Fig. 2: Properties of the *Item*, *Node*, and *Packet* classes in our core simulator implementation and the system layers they are responsible of.

Furthermore, although most simulators allow the users to configure certain parameters, none of the simulators provides an easy-to-use API that allows the users to perform various customizations on top of the existing simulator. Such a capability could prove useful for trying out proposed attacks, different consensus protocols, or changes to the underlying network structure. To give a concrete example, it would be possible to implement an attack such as selfish mining [7] by modifying the *block discovery by own pool* and *block discovery by others* events of a prototypical Bitcoin miner.

IV. Design & Implementation

In this section, we describe in detail the implementation we have done so far. We start with the simulator core, then describe how we simulated Bitcoin and various altcoins by extending the core simulator, and conclude by summarizing the performance improvements we have performed since the in-class presentation as that was our main agenda since then.

A. Core Simulator

Since one of our goals was to make the simulator as extendable as possible, we decided to start by building a *simulator core* that captured the main elements of a blockchain system. It could then be possible to extend this simulator core to implement a specific blockchain system.

Our core simulator consists of three classes that tries to capture the main properties of the data, consensus, and network layers. Figure 2 displays the three main classes with their attributes and the layers they are responsible of. We can further detail the three classes as follows:

- The *Item* class corresponds to the data that is being exchanged between the nodes, and stored on the blockchain. It can be extended to represent objects such as blocks and transactions, or any protocol-specific messages.
- The *Node* class represents the individual participants in the network. Each node has a unique id (UUID [26]), a timestamp that keeps track of the simulation step count, and its geographic region. Furthermore, each node has two Python dictionaries, *ins* and *outs*, that keep track of the incoming and outgoing connections of the node respectively. Keys in the dictionaries correspond to the nodes' ids and the values are the nodes themselves. Finally, each node has a Python dictionary we call its *inbox*. Keys in the inbox correspond to certain times within the simulation, and the values are items. In each call of its `step()` function, a *Node* increments its timestamp by one, obtains the items corresponding to that timestamp from its inbox, and performs the necessary actions on those items.
- The *Packet* class is a simple wrapper class around the *Item* objects. It contains a *delay* field that represents how much time (in simulation steps) it should take for it to be obtained by its receiver.

To run the simulation, once the nodes and the connections are established, a centralized coordinator iteratively calls the `step()` methods of the nodes. The parameter given to the `step()` method represents how many real-world seconds one simulation step corresponds to.

B. Simulating Bitcoin[1]

After our core implementation was complete, we built a Bitcoin simulator by extending it, which could in turn be extended to add different behaviors. We extended the *Node* class to implement a *Miner* class, and the *Item* class to implement *Inv* and *GetData* messages, as well as *Block*s. We detail the three layers below.

[1]Although we describe the implementation as simulating *Bitcoin*, it can easily be configured to simulate similar altcoins such as Litecoin [27] and Dogecoin [28] that only require changes to the block interval values for our purposes.

1) Data Layer: We simulate the Bitcoin data layer by extending the *Item* class to represent *Inv* and *GetData* messages, and *Block*s. The *Inv* and *GetData* messages are used to exchange blocks between the nodes as described before.

A *Block* object inherits the fields from its parent *Item*. For Bitcoin, our most important additions are the *prev_id* field corresponding to the id of the block the current was mined on top of, and the *tx_count* and *size* fields corresponding to the number of transactions in the block, and its size. We analyzed the last 365 days' transaction data obtained from [29], and calculated the mean transaction size as 615.32 bytes with a standard deviation of 89.43, and the average number of transactions per block as 2104.72, with a standard deviation of 236.63. When a block is created in our simulation, its number of transactions and their sizes are sampled from normal distributions following these values.

2) Consensus Layer: The consensus layer is exclusively implemented in the *Miner* class, addressing block generation and fork resolution problems.

Since it would be infeasible to implement the actual PoW mechanism, we implemented a mechanism that probabilistically generates blocks depending on each miner's mining power. Given some value for the expected block interval in terms of the number of simulation steps, we compute the value

$$\text{Difficulty} = \frac{1}{\text{BlockInterval} \times \text{TotalMinePower}}.$$

Then at each step, a miner generates a block with probability

$$\mathbf{P}[\text{GenBlock}] = \text{MinePower} \times \text{Difficulty}.$$

Since miners generate blocks independently, forks can occur. To resolve forks and choose which block to mine on, a miner should pick the determine the longest[2] chain. To simplify this calculation, we added a *height* field to the blocks. The genesis block has height 0, and then each block has its parent's height plus one. Thus the block with the highest height value corresponds to the head of the longest chain, and should be mined on.

Finally, each miner's mine power is determined based on its region. We divided the world into nine regions as shown in Table III, and obtained each region's share of global mining power from [30]. If there are multiple nodes within the same region, we dispersed the mining power equally among the nodes, so that the region as a whole maintains its global share.

3) Network Layer: Our network layer consists of a simple *Packet* class, and some helper methods to compute delay values. We assume the nodes can send messages to each other in a point-to-point manner, without any intermediate devices. When a node wants to send an *Item* to another node, it first wraps the item into a *Packet* object, and then invokes the network helpers to compute its delay value. The delay has two components: *propagation* and *transmission* delays. These values are again computed based on the sender and receiver nodes' regions. As shown in Table III, each region has fixed download and upload bandwidths (MB/s) which we obtained from [31]. Furthermore, each pair of regions has a fixed latency obtained from [32].

Region	**Down** (MB/s)	**Up** (MB/s)	**Nodes**	**Mine Power** (%)
China	31.6	23.6	189	65.08
US	55.3	19.2	1,892	7.24
Russia	23.1	20.2	251	6.90
Kazakhstan	12.9	8.0	5	6.17
Malaysia	25.1	19.4	19	4.33
Canada	59.0	14.6	316	0.82
Germany	63.4	24.5	1,769	0.56
Norway	37.5	17.6	32	0.48
Venezuela	7.7	3.5	2	0.42
TOTAL	-	-	4,475	92.0

TABLE III: Real world bandwidth, node count, and mine power distribution statistics for the Bitcoin network.

Considering regions A and B with latency l_{AB}, and bandwidths (d_A, u_A) and (d_B, u_B) respectively, the delay for a message with size S from region A to region B is computed as

$$\text{Delay} = l_{AB} + \frac{S}{\min\{u_A, d_B\}}.$$

The sender node obtains this delay value, adds its own timestamp to it, and uses the result as the key for inserting the *Item* into the receiver node's *inbox* dictionary. The receiver node then obtains the item from its inbox when its timestamp reaches the computed value.

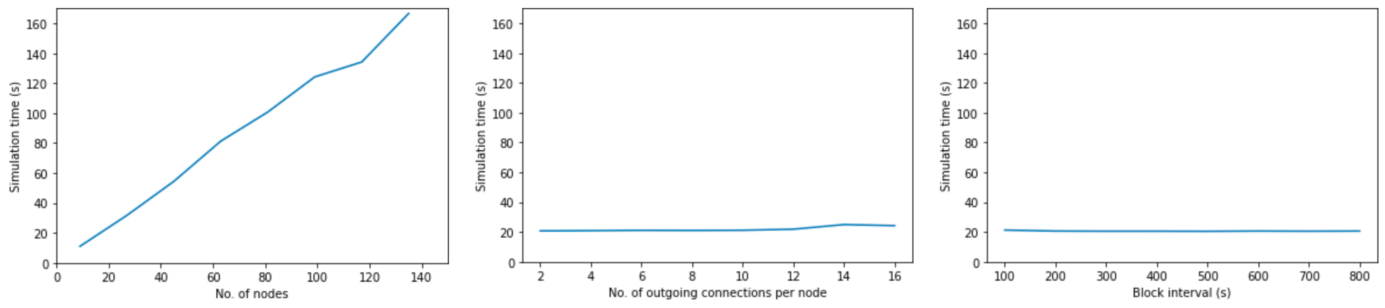
C. Performance Improvements

As we had explained in our previous report, our primary concern until the final demos was performance. To that end, we have made various performance improvements that resulted in a simulation running approximately 50 times faster. We detail the various improvements we have performed below.

The first and the most effective performance improvement we performed was related to the way we passed the data from the configuration file to the simulated entities. We use a YAML file to provide the main input parameters (total number of steps, node counts, mining powers etc.). This allows users to run different simulatiotns without changing any code. We then use the Hydra Python library [33] to parse the YAML file. When the program starts, Hydra loads the data in the YAML file into a structured Python object, from which we can access the parameters. One of the configuration parameters determines the number of real-time seconds one simulation step corresponds to, and we pass this parameter to the nodes' `step()` function in each call. The problem was that in each `step()` call, we were querying the related parameter of the object representing the configuration file. Performing this attribute lookup was taking a long time, and we were able to improve the simulation speed by around 20 times by first reading the value to a variable, and then passing that variable to the nodes.

The next major change was related to the way we handle the network layer. As described in our previous report, we initially

[2]In Bitcoin, the main chain is decided not based on length, but on the total amount of work done on the chain. Since we assume a fixed mining difficulty, the two concepts are equivalent for us.



(a) Number of nodes varies, with a fixed block interval (6,000 steps) and number of connections (2).

(b) Number of outgoing connections per node varies, with a fixed node count (18) and block interval (6,000).

(c) Block interval varies, with a fixed node count (18) and number of outgoing connections (2).

Fig. 3: Effects of the total node count, the number of outgoing connections per node, and block interval on simulation performance. It can be observed that the total running time increases linearly with the number of nodes, and is largely independent of the number of connections and the block interval. The values are obtained by averaging the running times of three simulation runs for 1,000,000 steps for each setup.

had a separate *Link* object for each connection between two nodes. Items were passed to this link object's queue and were transferred to the receiving node's queue in the appropriate time based on the item's delay value. In each simulation step, a node would first call the `step()` method of each of its outgoing links, and would then obtain the items it should act on that step from its incoming queue. Thus, with n nodes, k outgoing connections on average for each node, and incoming queues of size q, we would perform an operation taking time $O(nk + nq)$ at each simulation step. We then migrated to the setup described in the previous section. Now, sending a message from one node to another consists of a delay calculation, and inserting an item into a Python dictionary. To receive the items it should act on in step t, a node simply obtains the value for key t from its inbox dictionary. Both of these operations take $O(1)$ time, reducing the $O(nk+nq)$ time above to $O(n)$.

In addition to these major changes, we also performed some minor changes such as removing redundant mathematical operations, or inlining some functions that are called many times to reduce the overhead of function calls. In the end, as mentioned above, we were able to reduce the running time by a factor of 50. As of this writing, we can simulate the total 4,475 nodes for 15,000,000 steps ($\approx$416 real-world hours with 0.1 second steps) in around 9 hours on a personal computer with a 3.2 GHz ARMv8-A processor. While this is still far from our desired level of performance, it is significantly more reasonable than the 500 hours it would take before.

After the various improvements described above, as shown in Table 3, our simulator's running time is linear with respect to the number of nodes, and independent of the number of connections between nodes or the block interval. To obtain the results displayed in Table 3, we ran three simulations for each setup with 1,000,000 steps each and averaged their running times. Note that as the analysis in the paragraph one above shows, the running time was linear with respect to the number of outgoing connections and block interval as well before these changes.

D. Summary of Supported Features

Table II compares the supported features of the existing simulators with our simulator. To reinforce the preceding explanations, we conclude this section with a summary of the features our simulator supports.

Block/transaction size distribution. Block sizes by multiplying two values (number of transactions in block, average transaction size) obtained from normal distributions with means and standard deviations calculated from the real world values.

Geographic network conditions. Each node belongs one of the nine regions. Each region has separate upload and download bandwidth values, and each pair of regions has a fixed latency value. Transmission times for messages between two regions are calculated using these values.

Information propagation mechanism. Blocks are propagated using `INV` and `GETDATA` messages, similar to the Bitcoin information propagation mechanism.

Comparing real time with simulation time. Each simulation step corresponds to a configurable amount of real-world seconds. This value mainly determines how many simulation steps message transmissions will take.

Block interval distribution. Due to the probabilistic nature of the mining operation, block intervals vary around a preconfigured expected value.

Different miner behavior. Due to the modular architecture of our simulator, it is easy to implement miners behaving differently, by extending the Bitcoin *Miner* class and overriding the *generateBlock* (for when the miner mines a new block), *consumeBlock* (for when the miner receives a block), and *choosePrevBlock* (to choose which block to mine on) methods.

Transaction modeling (work-in-progress). Although our simulator has the concept of a transaction, we do not yet implement transactions as individual objects being transmitted over the network.

Metric	Bitcoin		Simulation					
	2016	2020	#1	#2	#3	#4	#5	#6
r_s	0.0041	0.0006	0.0034	0.0069	0.0044	0.0058	0.0055	0.0063
Δ_B (s)	9.69	10.05	9.96	10.04	10.19	9.94	10.17	10.51
tps	2.87	3.48	3.53	3.52	3.52	3.55	3.47	3.36
d_{50} (s)	8.7	0.5	2.49	4.51	5.07	2.25	3.14	4.06
d_{90} (s)	17.3	3.3	4.35	6.29	8.82	2.77	3.62	4.6

TABLE IV: Comparison between the real-world Bitcoin metrics and the results we obtained from our simulator.

Parameter	#1	#2	#3	#4	#5	#6
Nodes per region	1	10	1	Full	Full	Full
Simulation steps	30M	30M	30M	15M	15M	15M
Connections per node	2	2	2	32	8	4
Seconds per step	0.1	0.1	1	0.1	0.1	0.1

TABLE V: Input parameters for the six experiments we have performed (M: million, Full: real-world node counts).

V. Validating the Simulator

In this section, to validate our simulator, we compare the results obtained from simulations with the real-world values observed in the Bitcoin network. We first define the metrics we will compare, describe our experimental setups, and then present the results.

A. Experimental Setup

To experiment with a setup similar to the real-world Bitcoin network, we divided the world into nine regions that when combined make up more than 90% of the total mining power. Table III displays the regions with their bandwidths [31], node counts [34], and mining power [30]. The metrics we will compare are:

- **Average block interval** (Δ_B): Mean time between two blocks in the main chain.
- **Stale block rate** (r_s): Share of blocks that are mined but not included in the main chain.
- **Block propagation delays** (d_p): The time it takes for blocks to reach a certain share ($p\%$) of the nodes.
- **Transaction throughput** (tps): Number of transactions appended to the main chain per second.

We performed six different experiments to isolate various parameters and see how they affect the overall results. Table V describes the six setups in detail. We repeated the experiments with a small number of nodes (#1, #2, and #3) ten times and averaged their results. For the experiments with a high number of nodes (#4, #5, and #6), the number of repeats was three due to time constraints.

B. Results

Table IV compares the results we obtain from our simulator with those of the real Bitcoin network. We obtained stale block rates[3] and block propagation delay values for the Bitcoin network from [6], average block interval values from [35], and transaction throughput values from [29].

[3]In our previous report, we had mistakenly reported a stale block rate for the real-world Bitcoin network of 0.41% as 0.41 instead of 0.0041. This is the reason of the discrepancy between the two reports.

Stale block rate. Overall, the stale block rates we obtain from our simulator are fairly similar with the 2016 Bitcoin network. However, since we have not yet implemented various new features such as relay networks [36] and compact block relay [20] which reduce block propagation delays and thus stale block rates, our simulator outputs higher stale block rates compared to the 2020 Bitcoin network.

Furthermore, considering experiments #1 and #2, it can be observed that increasing the size of the network but keeping the number of connections per node results in a higher stale block rate. In a less connected network, blocks will need to go through a higher number of intermediate nodes, obtaining higher delay values. Higher delay values in turn increase the stale block rate, since the probability of miners finding blocks before receiving the recent ones increases.

Block intervals. Even though it was evident with the mining probability calculations described in Section IV-B2, the experimental results confirm that we obtain an average block interval of around 10 minutes.

Block propagation delays. Discussion of block propagation delays follows a similar theme with that of the stale block rates. Even though our results are realistic when compared to both of the 2016 and 2020 Bitcoin networks, the 2020 network has a significantly lower 50% delay value. This might also be due to the small number of highly-connected nodes in the Bitcoin network [19] that enable blocks to reach some nodes in a small number of hops. Since our network is randomly generated, it is more homogeneous and this effect is not present.

It is also visible that more well-connected simulations result in smaller block propagation delays. For example, as the number of connections decreases between experiments #3, #4, and #5, the block propagation delays increase as well.

Transaction throughput. Since we model the number of transactions in blocks by observing the Bitcoin network, we also obtain similar transaction throughput values from our simulator. Although these values do not provide much insight in their current form, we include them here as some form of a sanity check since the transaction throughput is one of the key performance metrics of a blockchain system.

Effect of simulation resolution. Comparing experiments #1 and #3 in which the only differing parameter is the number of seconds per simulation step (0.1 and 1 respectively), it can be observed that block propagation delays, and thus stale block rate, increases with a lower resolution. This is because with a simulation of one-second steps, each message transmission delay will be rounded up to the nearest integer. This increase can be expected to be more significant in a larger network in which messages make a higher number of hops on average, each hop adding an artificial delay between 0 and 1 seconds.

Effect of network size. Contrary to our expectations, we have observed that running the simulation with the exact number of nodes (experiments #3, #4, #5) did not seem to have a significant effect on the simulation accuracy as opposed to simplified setups with a fixed number of nodes per region (experiments #1, #2, #3). One possible explanation is that since we do not simulate the exact real-world network topology (i.e. a small number of highly-connected nodes), networks of both sizes follow a similar structure. Another reason might be that we do not yet differentiate full nodes from miners. While in the real-world Bitcoin network there is a small number of miner nodes (excluding those that are mining as part of a pool without joining the P2P network) compared to full nodes, all nodes are miners in our simulation. This means that both in small and large simulations, blocks can enter the network from any node, resulting in a more homogeneous distribution.

VI. Future Work

As of this writing, our primary goals for the future are implementing a realistic transaction model and some of the recent developments that improve network performance.

Implementing a realistic transaction model would help simulate more realistic network conditions, as transactions propagating over the network can affect the overall network performance. However, since transaction generation does not follow a well-defined distribution like block generation, it would first require us to find a way to model nodes generating transactions more realistically, following upon the existing works such as [37] and [38]. One way of simulating transactions without sacrificing performance could involve classifying the nodes into a few categories with respect to their transaction generating behavior, and modeling them in our simulator.

As we discussed while explaining our results, we should also implement various recent developments such as relay networks [36] and compact block relay [20] to bring our simulation up to date. These developments would help us simulate more realistic network conditions and yield our simulation more accurate.

Considering the customization opportunities, we plan to add an optional feature that allows users to define custom network topologies without modifying the actual source code, through another configuration file with a domain-specific language.

Running the simulator over an actual network over an RPC mechanism could be another potential feature that can be implemented with minimal modifications. The central coordinator would call nodes' `step()` methods over RPC, and nodes would likewise pair with their peers over RPC.

Overall, even though we are pleased with the progress we have shown so far, we plan to implement these changes among others and potential performance improvements that might become necessary to bring our work to our desired level.

VII. Conclusion

Although blockchain simulators can be useful tools for analyzing the performance and security of blockchain systems, most existing simulations suffer from various shortcomings. In this report, we presented our own blockchain simulator is designed with the goal of addressing some of those shortcomings without sacrificing too much performance.

Our initial results show that our simulator can simulate the Bitcoin network reasonably. However, its lack of a comprehensive transaction modeling capability and the recent network layer developments open up avenues for future work.

In the end, we hope that our work can become a meaningful contribution to the area of blockchain simulators and prove useful for future research.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf."

[2] G. Perboli, S. Musso, and M. Rosano, "Blockchain in logistics and supply chain: A lean approach for designing real-world use cases," *IEEE Access*, vol. 6, pp. 62018–62028, 2018.

[3] N. Kshetri and J. Voas, "Blockchain-enabled e-voting," *IEEE Software*, vol. 35, no. 4, pp. 95–99, 2018.

[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger,"

[5] C. Fan, S. Ghaemi, H. Khazaei, and P. Musilek, "Performance Evaluation of Blockchain Systems: A Systematic Survey," vol. 8, p. 24, 2020.

[6] R. Paulavicius, S. Grigaitis, and E. Filatovas, "A Systematic Review and Empirical Analysis of Blockchain Simulators," *IEEE Access*, vol. 9, pp. 38010–38028, 2021.

[7] I. Eyal and E. G. Sirer, "Majority is not Enough: Bitcoin Mining is Vulnerable," *arXiv:1311.0243 [cs]*, Nov. 2013. arXiv: 1311.0243.

[8] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the Security and Performance of Proof of Work Blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, (Vienna Austria), pp. 3–16, ACM, Oct. 2016.

[9] A. Narayanan, *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton: Princeton University Press, 2016.

[10] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," EuroSys '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[11] R. C. Merkle, "Protocols for public key cryptosystems," in *1980 IEEE Symposium on Security and Privacy*, pp. 122–122, 1980.

[12] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Consensus in the Age of Blockchains," *arXiv:1711.03936 [cs]*, Nov. 2017. arXiv: 1711.03936.

[13] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, pp. 173–186, 1999.

[14] M. Okun, "Agreement Among Unacquainted Byzantine Generals," in *Distributed Computing* (D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and P. Fraigniaud, eds.), vol. 3724, pp. 499–500, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. Series Title: Lecture Notes in Computer Science.

[15] J. A. Kroll, I. C. Davey, and E. W. Felten, "The Economics of Bitcoin Mining, or Bitcoin in the Presence of Adversaries," p. 21, 2013.

[16] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies," in *2015 IEEE Symposium on Security and Privacy*, (San Jose, CA), pp. 104–121, IEEE, May 2015.

[17] Y. Hassanzadeh-Nazarabadi, A. Küpçü, and Özkasap, "Lightchain: Scalable dht-based blockchain," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2582–2593, 2021.

[18] C. Decker and R. Wattenhofer, “Information propagation in the Bitcoin network,” in *IEEE P2P 2013 Proceedings*, (Trento, Italy), pp. 1–10, IEEE, Sept. 2013.
[19] A. Miller, J. Litton, A. Pachulski, N. Gupta, N. Spring, B. Bhattacharjee, and D. Levin, “Discovering Bitcoin’s Public Topology and Influential Nodes,” p. 17.
[20] M. Corallo, “Compact block relay.” https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki, Feb 2020.
[21] M. Alharby and A. van Moorsel, “BlockSim: An Extensible Simulation Tool for Blockchain Systems,” *arXiv:2004.13438 [cs]*, Oct. 2020. arXiv: 2004.13438.
[22] Y. Aoki, K. Otsuki, T. Kaneko, R. Banno, and K. Shudo, “SimBlock: A Blockchain Network Simulator,” p. 5, 2019.
[23] C. Faria and M. Correia, “BlockSim: Blockchain Simulator,” p. 8.
[24] S. Pandey, G. Ojha, B. Shrestha, and R. Kumar, “BlockSIM: A practical simulation tool for optimal network design, stability and planning.,” p. 5.
[25] L. Stoykov, K. Zhang, and H.-A. Jacobsen, “Demo: VIBES: Fast Blockchain Simulations for Large-scale Peer-to-Peer Networks,” p. 2.
[26] P. Leach, M. Mealling, and R. Salz, “A universally unique identifier (uuid) urn namespace,” 2005.
[27] C. Lee, “Litecoin - open source p2p digital currency.” https://litecoin.org, 2021.
[28] J. Palmer and S. Nakamoto, “Dogecoin.” https://dogecoin.com, 2021.
[29] “Blockchain explorerr.” https://www.blockchain.com/charts#currency, 2021.
[30] C. C. for Alternative Finance, “Bitcoin mining map.” https://cbeci.org/mining_map, Apr 2020.
[31] TestMy.net, “Top countries for bandwidth.” https://testmy.net/country, 2021.
[32] WonderNetwork, “Global ping statistics.” https://wondernetwork.com/pings, 2021.
[33] F. O. Source, “Hydra.” https://hydra.cc, 2021.
[34] A. Yeow, “Global bitcoin nodes distribution.” https://bitnodes.io, 2021.
[35] BitInfoCharts, “Bitcoin block time chart.” https://bitinfocharts.com/comparison/bitcoin-confirmationtime.html, 2021.
[36] K. Otsuki, Y. Aoki, R. Banno, and K. Shudo, “Effects of a simple relay network on the bitcoin network,” in *Proceedings of the Asian Internet Engineering Conference*, AINTEC ’19, (New York, NY, USA), p. 41–46, Association for Computing Machinery, 2019.
[37] D. Ron and A. Shamir, “Quantitative analysis of the full bitcoin transaction graph,” in *International Conference on Financial Cryptography and Data Security*, pp. 6–24, Springer, 2013.
[38] D. Kondor, M. Pósfai, I. Csabai, and G. Vattay, “Do the rich get richer? an empirical analysis of the bitcoin transaction network,” *PloS one*, vol. 9, no. 2, p. e86197, 2014.

Appendix A
Tutorial

This tutorial is designed to walk the user through the process of running a Bitcoin simulation and analyzing the results.

A. Prerequisities

To obtain the latest version of the simulator, clone the `main` branch of our GitHub repository (`https://github.com/simbadt/simbadt`), and `cd` into the main directory. The files and directories have the following meanings:

- `/bitcoin`: Source code of the Bitcoin simulator.
- `/docs`: Documentation HTML files.
- `/sim`: Source code of the simulator core.
- `main.py`: Main script to run simulations.
- `config.yaml`: Simulation configuration file.
- `requirements.txt`: Required Python libraries.

You should run the following command to install any missing Python libraries:

```
pip install -r requirements.txt
```

B. Running Simulations

The `config.yaml` can be modified to run Bitcoin simulations without delving into the code. The repository contains a sample configuration file with all the possible options:

- `sim_name`: Name of the simulation run. Results will be saved under a directory with this name.
- `results_directory`: Location the results will be saved at. Make sure you have write access to this directory.
- `log_level`: Detail of the logging output during the simulation. Possible values are `CRITICAL` (no logging), `WARNING` (+ only simulator messages), `SUCCESS` (+ block generations), `INFO` (+ block receipts), and `DEBUG` (+ all protocol messages). A less detailed option is recommended for better performance.
- `sim_reps`: How many times to repeat the same simulation.
- `sim_iters`: How many steps to perform in one simulation.
- `iter_seconds`: How many real-world seconds one simulation step corresponds to.
- `block_int_iters`: The expected block interval in *simulation steps*.
- `connections_per_node`: Number of *outgoing* connections per node.
- `nodes_in_each_region`: Number of nodes in each region. Giving this parameter -1 as value uses the real-world values for node counts. Giving any other value will assign that many nodes to each region.

Finally, the `nodes` option of the sample configuration file inputs the necessary real-world data (see Table III) into the simulation.

To run the simulation, run the `main.py` script as follows:

```
python main.py
```

C. Analyzing Results

Once a simulation is done running, each node will be dumped into the directory provided in the configuration file, under the folder with the provided simulation name. Each node will be in a different file, with the name corresponding to that node’s human-legible name. By default, names follow the convention `MINER_<region>_<count>`.

We provide various basic analysis methods in the file `bitcoin/analysis.py`, and a Jupyter notebook (`bitcoin/analysis_notebook.ipynb`) demonstrating how to deserialize the nodes and call the analysis methods. The documentation describes the various analysis helper methods in more detail.

D. Customizing Simulations

Although the configuration file provides a fast way to run basic simulations, its capabilities are limited. It is possible to experiment with different node types or network topologies.

To **implement a new node type**, you can extend the `bitcoin.Miner` class and override the necessary methods, modifying the `bitcoin/models.py` file. Afterwards, by modifying the `type` option for regions in the configuration file, you can add your nodes into the simulation. Refer to the documentation for details on how the base `Miner` class operated and which methods can be overriden.

To **experiment with a custom network topology**, you can modify the `main.py` file to give nodes names fitting your purpose, and then connect them to each other as desired. For example, to construct a network with a ring topology, it would suffice to connect each node to its predecessor and successor in the list they were created into.

We plan to provide various tools that make these customizations easier in the future.