BlockSim: Blockchain Simulator

Carlos Sérgio Figueira Faria
Instituto Superior Técnico, Universidade de Lisboa
carlosfigueira@tecnico.ulisboa.pt

***Abstract*—Blockchain systems have received an outburst of interest both in academia and industry. Blockchains are distributed ledgers where a group of network participants who do not fully trust each other, agree and reach consensus around the global state of the ledger. The rapid expansion of this technology makes it extremely challenging and rewarding to understand its frontiers and potential. However, the lack of tools to evaluate design and implementation decisions might be hampering a faster progress. To address such issue, this paper presents a discrete-event simulator that is flexible enough to evaluate different blockchain implementations. These blockchains can thus be rapidly modeled and simulated by extending existing simulation models. The simulator has been used to simulate both the Bitcoin and the Ethereum networks and to compare the results with measurements taken from the real networks. Running a Bitcoin and Ethereum simulation model allows for the possibility of changing environment conditions and answer different questions as well as performing a comprehensive evaluation of the whole system. The process can be adapted to any blockchain system.**

I. Introduction

Blockchain is a promising new technology, generating widespread interest, and receiving considerable attention in the research community [1], [2]. This success has predominantly been attributed to the success of Bitcoin [3], [4].

Blockchain is being applied in critical sectors of society, such as: financial, health care, energy and logistics, among others. However, it lacks on proper tools to evaluate or simulate certain events or conditions.

A blockchain, or distributed ledger, consists in an append-only data structure that stores an ordered list of transactions, replicated in several nodes connected by the Internet. Blockchains typically assume that these nodes, which do not fully trust each other, may behave in a Byzantine manner. At the same time, they need to reach a consensus on the order of transactions, which has to tolerate Byzantine failures. New transactions can be added to the blockchain but it is not possible to modify those already listed, thereby ensuring integrity of transactions.

The original blockchain was the core of the Bitcoin *cryptocurrency* system, where nodes store and replicate *digital coins* as system state. These digital coins move from one address to another. The notion of blockchain has grown beyond cryptocurrency systems, and Ethereum [5] has emerged as a blockchain capable of defining more complex states, enabling Turing complete code to be executed within a transaction - also known as *smart contracts*. Bitcoin and Ethereum operate in a public environment, where nodes can join and leave the network without authorisation - so they are known as *permissionless* blockchains.

The autonomous and decentralised nature of records and smart contracts provides the potential to transform important industrial sectors [6]. The raising interest from the industry has led to the development of new blockchains, specifically designed to meet requirements for such private environments. To authorise a limited set of participants, so called *permissioned* blockchains have been recently proposed. Upon the most recognised are Hyperledger Fabric [7], [8] and Tendermint [9]. These permissioned blockchains are characterised do deploy deterministic mechanisms, such as Byzantine Fault-Tolerant (BFT) consensus protocols [10]–[13].

While there is a broad interest in developing blockchain systems for specific use cases, there is a lack of tools to perform their evaluation. Current evaluation methods use *emulation*, which imitates the behaviour of a system in a large set of machines [14], [15]. This approach, however, incurs in large overhead and lacks scalability to real world deployments. Besides, power consumption of a large-scale system must be taken into account.

Another valid alternative is *simulation*. Network and distributed system simulators are important tools to evaluate the performance of protocols and systems in a large set of conditions. These simulators provide an environment that simplifies the implementation and deployment of protocols. With simulation it is possible to study a large-scale system with thousands of nodes in a single machine and gather results in reasonable time.

The present paper proposes a blockchain simulator, *BlockSim*. The objective of the paper is to design, and implement, a simulator for blockchains where such systems can be implemented in a simple way and have their performance evaluated in different conditions. The followed approach provides a framework with pre-existing simulations models, commonly present across all blockchain implementations (blocks, transactions, ledger, network). Users can extend these simulation models to evaluate their own design and implementation decisions. The framework will then take the created models and execute them in the simulator, according to a set of events defined by users. This approach provides a very versatile solution without the burden of implementing a simulator from scratch, and can be extended to simulate any kind of blockchain implementation.

The main objective of this paper is to provide a simulator capable of evaluating different blockchains in different environment conditions, enabling, thus, a richer understanding of this technology. The contributions of this paper are the following: (1) provide a simulator capable to run user defined simulation models; (2) provide a simulator capable to run thousands of nodes on a single host; (3) provide the possibility of users change the simulated environment conditions; (4) simulation should provide an accurate representation of a real blockchain system; (5) simulation should be performed in reasonable time; (6) simulator has to provide a report with the simulated results when concluded.

II. Background and Related Work

One common approach to perform system evaluations is to use machines at universities or use a global research network such as PlanetLab [15]. However, these deployments do not accurately reflect the same network conditions of a public live network, and also suffer from scalability and management issues. Simulators, on the other hand, provide a simpler environment to implement, scale and run systems or protocols.

Simulation are used to study a wide variety of questions about the real system. It is also used to predict the effect of changes to existing systems, and assist in the design of new systems, by predicting the performance under a set of variables.

A simulation attempts to reproduce the behaviour of a system and its progress over time. To do so, simulations run a *model*. A model is a set of assumptions about the operation of the system and can be classified as follows [16], [17]: *stochastic* model receives random input values, leading to random outputs and *deterministic* model does not contain random values; *static* model represents a system at a particular point in time and *dynamic* model over a certain time frame; *discrete-event* model describes the system as a sequence of events that occur after a change of state in the system, so it is possible jump in time from one event to the next. On the other hand, *continuous* model tracks the system states over time.

Blockchain implementations vary widely in choice of runtimes, programming languages, operation systems, cryptographic libraries, messaging, and thread model, making it hard to diagnose problems and analyse bottlenecks. By building a simulation model of the performance of such implementations, it is possible to use simulation to directly compare specific implementations variants in a common framework.

Simulators like The ONE [18], PeerSim [19], CloudSim [20] and BFTSim [21] are useful tools in the development of protocols and systems for opportunistic networks, peer-to-peer networks, cloud computing and byzantine fault tolerance, respectively. In addition to these, there are simulators created to perform evaluation on the impact of network-layer parameters on the security of Bitcoin PoW, such as Bitcoin Simulator [22], Shadow-Bitcoin [23] and VIBES [24]. All these simulators follow a discrete-event simulation (DES) model.

All these simulators create a model for certain resources, such as: network latency, bandwidth, ad-hoc networks and CPU. Each resource model can be associated to specific parameters. For instance, a network resource model can adopt two parameters: a link latency and link bandwidth, and a CPU resource can model the computation rate.

Bitcoin Simulator, Shadow-Bitcoin and VIBES try to simulate Bitcoin on a large-scale network running thousands of nodes, on a single host. *However, these simulators are restricted to a concrete blockchain and thereby they do not have the flexibility to extend or replace the model, to easily simulate other blockchain systems following different consensus models or protocols. This is the limitation that we aim to solve with this work.*

In summary, with simulators it is possible to design and evaluate all types of systems and protocols by modelling resources and running them in thousands of nodes, in a single host.

III. BlockSim

BlockSim[1] consists in a simulation framework that assists in the design, implementation, and evaluation of existing or new blockchains. BlockSim provides fast and useful insights as to how a certain blockchain system operates, thorough examination of certain assumptions on the simulation models without the overhead of deployment and implementation of a real network.

This simulator follows a stochastic simulation model, being able to represent random phenomena by introducing probability distributions for certain events. Our models are considered dynamic; they can represent the system over a certain interval. A DES model is suitable to model a blockchain system, since an event-based system collects and changes states at a discrete point in time. This way, the change of state variables only needs to be tracked at discrete points in time, as opposed to continuously over time. Therefore, the simulator can keep track of thousands of nodes and events that only change states.

IV. Modelling of Random Phenomena

A random phenomenon is a situation where we know a certain event could happen, but we do not know which particular outcome will happen. However, for these phenomena we can observe a regular distribution of outcomes in a large number of repetitions.

When creating our models we pretend always to mimic the behaviour of the entities. For instance, we know the average time between blocks during a certain interval on a public blockchain. With this information, we can predict the next outcome with a degree of confidence. We do it by extrapolating a probability distribution for a given phenomena observed in a real system.

In practical terms, we assembled a methodology to measure, collect, and extrapolate a probability distribution that our models will use. For instance, in order to calculate the throughput when sending and receiving TCP packets between different geographic locations, our procedure was: (1) instantiate two instances on Amazon Web Services (AWS) on the desired geographic locations with *iPerf3*; (2) measure the throughput received and sent between each instance using iPerf3, at each hour, for 24 hours; (3) at the end of 24 hours, we

[1]Available at: https://github.com/BlockbirdLabs/blocksim

collect the iPerf3 logs from the two instances; (4) use the Kolmogorov–Smirnov test to know which distribution and its input parameters that best fit the samples collected in Step 3; (5) the distribution name and its input parameters are then consumed by the simulator, to extrapolate the values of throughput between different geographic locations during the simulation.

We use the same procedure to extrapolate values for latency, by collecting *ping* traces between different geographic locations. Similarly, to obtain the time to validate a transaction or a block we use a real deployed blockchain node. All these information is then used by our models.

V. Architecture

BlockSim follow a single process architecture, represented in Figure 1, where we illustrate the main components, connectors and interfaces of the implementation. The following sections present the functionality of each one.

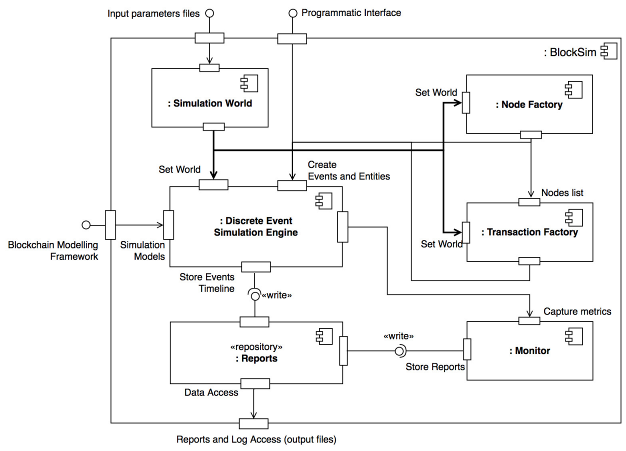


Figure 1: Architecture of BlockSim showing the main components, connectors and interfaces.

A. Discrete Event Simulation Engine

We use *SimPy* [25] as a framework to implement and run our *Discrete Event Simulation Engine* (DESE). SimPy is a process-based discrete-event simulation framework based on Python. *Processes* in SimPy are based on *Python generator functions* and can be used to simulate asynchronous networking or to implement multi-agent systems. Generators allow the programmer to specify a given function to be exited and then later re-entered at the point of last exit, enabling functions to alternate execution with each other. The exit and re-entry are performed by Python *yield* keyword.

In short, our DESE using SimPy supports several core functionalities, such as: scheduling of events; queuing and processing of events; communication between components; management of the simulation clock; and control the access of resources by the entities.

All these characteristics from SimPy helped us to accelerate the development of BlockSim. The BlockSim user can also use all the functionalities from SimPy when creating new models. SimPy is a framework to build arbitrary models or simulators. BlockSim, on the other hand, offers a more tailored framework to simulate any blockchain system with additional components that we will explore in the next sections.

B. Simulation World

The *Simulation World* component (see Figure 1) is responsible to manage the inputs of the simulator, which mainly are the probability distributions mentioned in Section IV. Additionally, configurations to the simulator are also considered. These inputs parameters are needed in the simulation models, which are defined using the *Blockchain Modelling Framework* (*cf.* Section V-F). These input parameters consist on the following files: (a) *Configuration file*: name of blockchain being simulated, possible locations for nodes, and for each blockchain different configurations are possible: probability of orphan blocks; message size; block size or gas limit; (b) *Delays file*: contains the probability distributions corresponding to the time to validate a transaction, block and time between blocks, for each blockchain; (c) *Latency file*: contains the probability distributions corresponding to the latency between possible locations for nodes; (d) *Throughput received and sent files*: a file containing the probability distributions of received throughput and another to sent throughput, between possible locations for nodes.

The user needs to point these files to the Simulation World and also specify the simulation start time and duration. This component then returns a variable *world* that will be injected on different components, making available all the attributes which characterise the world of the simulation.

C. Transaction and Node Factory

The *transaction factory* is responsible for creating batches of random transactions. The created transactions will be broadcasted when simulation is running by a random node on a list.

The *node factory* creates nodes that are used during the simulation. The user can specify the location, number of miners and non-miners, and the range of hash rate for the miner nodes. The location of each node needs to be recognised by the simulator.

D. Programmatic Interface

The programmatic interface is the main interface available to the user. Using *Python* language and SimPy [25], the user can write their own models, use the existing ones to define their own blockchain system, or modify any aspect of models already implemented. This interface is also responsible to start the simulation. When started, the DESE will consume the events and entities before initialising the simulation, to know which models will be used.

```
now = int(time.time()) # Current time
duration = 7200 # 2 hours
world = SimulationWorld(
    duration,
    now,
    "input/config.json",
    "input/latency.json",
        "input/throughput-received.json",
        "input/throughput-sent.json",
        "input/delays.json")

    net = Network(world.env, "Network")
    miners = {
        "Ohio": {
            "how_many": 0,
            "mega_hashrate_range": "(20, 40)"
        }
    }
    non_miners = {
        "Tokyo": {
            "how_many": 3
        },
        "Ireland": {
            "how_many": 2
        }
    }
    factory = NodeFactory(world, net)
    nodes = factory.create_nodes(miners,
    non_miners)
    world.env.process(net.start_heartbeat())
    for node in nodes:
        node.connect(nodes)
    trans_factory = TransactionFactory(world)
    # Broadcast a batch of 6 transactions
    every 5 mins, 7 times
    trans_factory.broadcast(7, 6, 300,
    nodes_list)
    world.start_simulation()
```

Listing 1: A demonstration of how to define the simulation using different models.

Listing 1 demonstrates how the user can instantiate the simulation, starting by creating the Simulation World. The Simulation World is then instantiated with the simulation duration in seconds, timestamped when the simulation starts and finally the file path to each input parameter. After creating the network, the user uses Node Factory to create the nodes for the simulation. The user then starts the network heartbeat and connects all nodes with each other. Using the Transaction Factory, the user broadcasts a batch of six transactions every five minutes, seven times, in a total of forty two transactions broadcasted during the simulation. Finally, the function gives the order to DESE start the simulation.

E. Monitor and Reports

The goal of the *monitor* is to capture metrics during the simulation, such as: number of transactions each node broadcasts or receives; transactions added to the queue; blocks processed; time to propagate transactions and blocks. It should be easy for the user to update metrics wherever needed, and have them automatically collected and stored in the *Reports* component.

F. Blockchain Modelling Framework

To simulate any blockchain system, we need to split it into detached layers, creating an abstraction that does not follow a specific implementation. We can identify the following high-level layers in a blockchain system: *Node layer* specifies the responsibilities and how a node operates when being part of a given network; *Consensus layer* specifies the algorithms and rules for a given consensus protocol, hence responsible for gathering consent among all nodes in the network toward replicated data; *Ledger layer* defines how a ledger is structured and stored. The most common structure might be an ordered list of transactions or blocks and each node stores a copy of the ledger; *Transaction and block layer* specify how information is represented and transmitted; *Network layer* establishes how nodes communicate with each other; *Cryptographic layer* defines what cryptographic functions will be used and how.

Models such as: Node, Transaction, Block, Consensus, and Network are available as classes that can be extended by the user. These are then used by the DESE to create blockchain system entities, which interact within events defined in the models.

Figure 2 represents the classes available in the framework. The basic models, represented by gray colour, will be explored in the following sections, and they can be extended to simulate specific blockchain implementations.

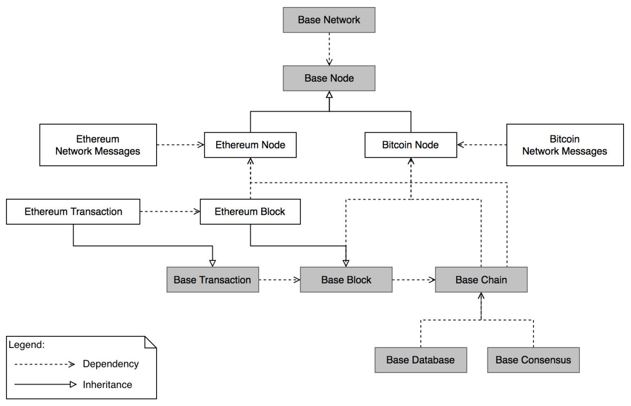


Figure 2: Class diagram of the modelling framework.

1) Chain Model: Responsible to mimic the behaviour of a chain. In this model, we implement an abstract functionality that works across different blockchains.

The most important functionality is adding a block to the chain. The chain model first checks if the block is being added to the head (previous hash of the block points to the head of the chain); if this is the case, it simply adds the block to the chain. Otherwise, the block is added to a parent queue that will be consulted every time a new block is being added, checking if the new block points to a block on the parent queue. This solves the problem of when a node receives the child block first, and then the parent, because of delays on the network.

When a block is not being added to the head, but the previous hash points to an old block, the model creates a *fork* on the chain by creating a secondary chain. Then, it checks if the block should be the new head by calculating the difficulty of the chain [3], [5]. If this is the case, it accepts the secondary chain as the main chain.

2) Consensus Model: It introduces the rules to be applied when validating blocks and transactions. In this model, we opt not to perform validations; on the other hand, the model adds a delay that simulates the validation process and we assume all blocks and transactions are valid.

The consensus model also defines the rules to calculate a difficulty of a new block. We opt to introduce a simple calculation of the difficulty, considering P_d as the block parent difficulty, B_{TS} as timestamp of new block, and P_{TS} as timestamp of parent block. The new block difficulty is calculated using the following equation: $\mathit{difficulty} = P_d + (B_{TS} - P_{TS})$

The equation simplifies and resembles the ideals of Ethereum [5] and Bitcoin [3] by incrementing the difficulty of a block when it is created in less time.

The difficulty represents the minimum amount of effort required to mine a new block on top of the current chain head. The consensus model can be extended and equation difficulty changed accordingly.

3) Block and Transaction Model: A block model defines the structure of a block, divided in its header and associated list of transactions. The transaction model defines the structure of transactions.

4) Network Model: The network model is responsible to know the state of each node during the simulation, establish the connection channels between nodes, and apply a network latency on the messages being exchanged. The network latency delay applied depends on the geographic location of destination and origin node. This delay is obtained by the probability distribution previously input as a parameter of Simulation World.

It is not defined any P2P discovery protocol; the user has the functionality to choose what nodes to connect. Therefore, he can define an additional model to simulate a particular P2P discovery protocol.

The mining process of a new block is in part held by the network model because it knows and can interact with any node. Hence, during all the simulation, the network entity selects one node to broadcast his candidate block. The interval between each selection, which we call the *network heartbeat*, corresponds to the time between blocks, depending on the blockchain system being simulated. Each node has a corresponding hash rate. The greater the hash rate, the greater the probability of the node being chosen.

The network model also simulates the occurrence of orphan blocks [4]. The network model simulates this behaviour by selecting two nodes to broadcast its candidate blocks. This event only occurs with a predefined probability [26], [27], set in configuration.

5) Node Model: The node model is responsible to provide the functionality to a node operating in a P2P network. When a simulation starts, a node connects to a list of nodes defined prior to the simulation run. When a connection occurs, the origin node starts listening for inbound communications from a destination node during the simulation. On the other hand, a node can send a message to a specific neighbour or broadcast a message to all neighbours. In the context of the simulator, an event is being scheduled to be processed by other entity, the destination node.

This node model is also responsible to apply a delay when receiving and sending messages. This delay depends on the message size. The size of each message is specified depending on the blockchain system being simulated and the throughput correspondent to where the node pretends to send or receive the message. These delays are obtained by probability distributions, as mention in Section IV. For the first connections between nodes, we apply a three-times latency delay corresponding to the TCP handshake. After that, the following communications only apply to one latency delay, which is referenced in the network model.

All these basic models can be extended to support different blockchain systems by creating high-level models, which we will explore in the next section.

VI. Modelling Bitcoin

Using the Blockchain Modelling Framework, we can easily model the Bitcoin blockchain, by reusing the base models already created, as shown in Figure 2.

In the Simulation World, we input the block size limit and also extrapolate the probability distribution for the number of transactions per block, considering the average number of transactions on the Bitcoin public network during the last two years [28]. Therefore, if the block size limit is 1 MB, as we know in Bitcoin [3], we take from the probability distribution the number of transactions; but, if the user chooses to simulate an environment with a 2 MB block, we multiply by two the number of transactions. With this, we can see the performance in different block size limits.

A. Bitcoin Network Messages Model

The Bitcoin network protocol [29] defines a group of messages that are exchanged between nodes. For each message, it is defined the name, payload, and size. We define the following messages in our model: *inv* allows a node to advertise its knowledge of one or more transactions or blocks; *getdata* used to retrieve the content of a specific block or transaction; *tx* sends a single transaction, in reply to getdata; *block* sends a specific body of a block in response to a getdata message that requests transaction information from a block hash; *headers* sends block headers to a node that previously requested certain headers with a getheaders message; *getheaders* requests a headers message that provides block headers starting from a particular point in the block chain.

The user can easily modify the message sizes in the configuration file. The model then reads configurations through the world variable (*cf.* Section V-B) to calculate the size of each message. The sizes are taken from the documentation [29].

B. Bitcoin Node model

The Bitcoin Node model inherits the base node model, as shown in Figure 2. With a predefined functionality to operate a node in a P2P network, inherited from the base node model, we can focus on building a specific model to Bitcoin protocol.

Bitcoin nodes in this simulation can be divided into two groups: a miner node or a non-miner node (or full-node). A non-miner node only needs to wait and validate new blocks that appear in the network or validate and broadcast new transactions. A miner node validates and collects, in a transaction queue, each new transaction. The creation of a candidate block is the process of collecting the pending transactions and fitting them in a block. The node only broadcast its candidate block to the network, when selected by Network base model; this process simulates the mining of a new block.

We do not perform any type of cryptographic operations or validations; we only apply a delay corresponding to the process of validation in a real system, which is previously measured (*cf.* Section IV).

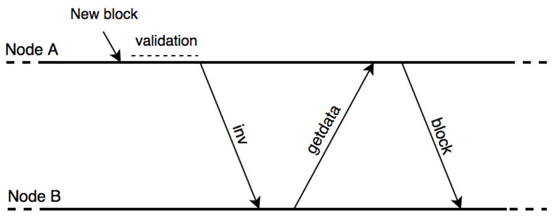


Figure 3: Messages exchange in Bitcoin protocol between nodes in order to obtain a new block.

The process of announcing a new block is illustrated in Figure 3, starting by Node A announcing a new block to his neighbours with an inv message. When Node B receives the inv message, it calls Node A by using getdata message to send the entire block it announced. Node A receives the getdata message and sends the entire block to Node B through block message. When Node B receives the block, it starts a validation process, and adds it to the Node B chain.

The same process works for new transaction(s) announced by a node on the network. When a miner node receives a new transaction, it is added to a transaction queue.

VII. Modelling Ethereum

Using the Blockchain Modelling Framework, we can also easily model the Ethereum blockchain by reusing the base models already created, as shown in Figure 2.

Additionally, in the Simulation World component, we input the block gas limit and the start gas for every transaction. The start gas represents the maximum amount of gas the originator of the transaction is willing to pay, also known as gas limit. For instance, if we configure our environment to have a block gas limit of 10,000, and a transaction gas limit of 1,000, in our simulation we will fit 10 transactions per block. With this we can see the performance in different block gas limits.

A. Ethereum Network Messages Model

The Ethereum Network protocol (PV62) [30] defines a group of messages that are exchanged between nodes. For each message, we set the name, payload, and size. We define the following messages in our model: *Status* informs a node of its current state: protocol version, network identifier, total difficulty, block hash in the head of the chain, and hash of genesis block. This message should be sent after the initial handshake and prior to any Ethereum-related messages; *NewBlockHashes* advertises one or more new blocks that have appeared on the network; *Transactions* sends one or more transactions; *BlockBodies* sends block bodies, that were previously requested to a node; *GetBlockBodies* are used to retrieve specific block bodies using block hashes; *BlockHeaders* send block headers to a node which was previously requested; *GetBlockHeaders* request a BlockHeaders message that provides block headers starting from a particular point in the block chain.

The user can easily modify the messages sizes in configuration file. The model then reads configurations through the world variable to calculate the size of each message. The sizes are taken from the documentation [30].

B. Ethereum Node model

The Ethereum Node model inherits the base node model, as shown in Figure 2. With a predefined functionality to operate a node in a P2P network, inherited from the base node model, we can focus on building a specific model to the Ethereum protocol.

Ethereum nodes in this simulation can also be divided into two groups: a miner node or a non-miner node. Providing the same functionality as the Bitcoin node model in Section VI-B.

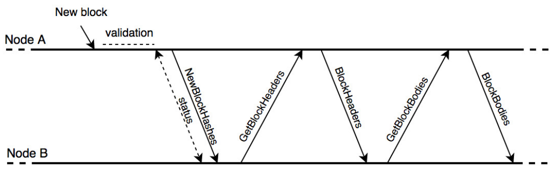


Figure 4: Messages exchange in Ethereum protocol between nodes in order to obtain a new block.

The process of announcing a new block is illustrated in Figure 4 starting with Node A announcing a new block to its neighbours with a NewBlockHashes message. When Node B receives the NewBlockHashes message, it calls Node A by using GetBlockHeaders message to send the block header of the block it announced. Node A sends the block header to Node B using BlockHeaders message. Node B then calls Node A to obtain the transactions and uncle blocks, using the GetBlockBodies message. Finally, Node A responds by sending the block body with the message BlockBodies. When Node B receives the block body, it starts a validation process, and adds it to his chain.

The process of announcing a new transaction has less overhead than announcing a new block. Node A receives a new transaction, validates the transaction, and then uses the Transactions message to broadcast the full transaction to its neighbours. When a miner node receives a new transaction, it adds the transaction to a transaction queue.

The *Ethereum Transaction model* extends the base transaction model by only adding new attributes, such as the gas price and start gas. The product of this two attributes is used to calculate the transaction fee.

The *Ethereum Block model* extends the base block model by only adding new attributes, such as the gas limit and gas used.

VIII. Evaluation

The goal of BlockSim is to provide an accurate representation of a real blockchain system. Therefore, Section VIII-A performs a *verification* and *validation* of BlockSim running our Ethereum models by replicating the same environment in a *real Ethereum network* and comparing the results. Section VIII-B explores and evaluates real use cases for BlockSim.

All the BlockSim executions were conducted on a computer with 2 GHz Intel Core i7 processor and 8 GB RAM.

A. Block and Transaction Propagation

We use BlockSim to perform a simulation study of the Ethereum reference implementation. The steps are the following: (1) Clearly identify the question that is to be answered. In our study we will answer the following question: "how long it takes to propagate a block and a transaction from one node to another?"; (2) Conceptualise the simple building models needed to answer the question. For our study, we need to represent the following models: block, transaction, network, messages, node and consensus; (3) Determine the input parameters for the models. For our study, we need the following input parameters: block and transaction gas limit; message size; distribution time delay to validate a block or transaction; distribution of latency and throughput between each node geographic locations; (4) Collect data from existing deployments for each input parameter. In Section IV we explained how data for the input parameters are collected; (5) Code the conceptual models composed in Step 2. We used the BlockSim Modelling Framework to create the specific Ethereum models for our study; (6) Perform verification of the models to understand if it is performing properly. If not, repeat the Step 5; (7) Validate if the conceptual model is an accurate representation of the Ethereum system, by comparing the simulated results with the measurements taken from a private Ethereum network.

In order to validate if the Ethereum models are an accurate representation of a real Ethereum system, as mentioned in Step 7, we collect data from the simulator and also from a private Ethereum network and compare the results.

We start by using the BlockSim Monitor, in respect to our simulation study question (*cf.* Step 1), to calculate the block and transaction propagation time between two nodes in the simulation, by calculating the difference when a block or transaction is sent and received. We start our simulation with the input parameters presented in Table I, for an Ethereum network with one miner node and one non-miner node.

We then changed the Ethereum client reference implementation[2], to be able to record in the log file the UNIX time when a block or transaction is sent to his peers and when it is received.

[2]Available at: https://github.com/carlosfaria94/go-ethereum

Table I: Input parameters for the probability distributions

	Distribution	Location	Scale	Additional parameters
Block validation delay	Log-normal	0.229 s	0.002 s	-
Transaction validation delay	Log-normal	0.004 s	0.00005 s	-
Time between blocks	Normal	15.79 s	3.00 s	-
Latency between Ohio and Ireland	Normal	73.70 ms	0.09 ms	-
Throughput between Ohio and Ireland	Beta	39.13 Mbps	59.02 Mbps	α = 0.463 β = 0.461
Latency between Ireland and Tokyo	Normal	105.42 ms	0.23 ms	-
Throughput between Ireland and Tokyo	Beta	-410160.67 Mbps	410197.05 Mbps	α = 272250.32 β = 3.69

We then deployed a private Ethereum network using the changed Ethereum client in Amazon Web Services (AWS). The private network has two instances, each one with 2 virtual CPUs, 4 GB RAM with 8 GB SSD. The goal is to replicate the same environment during the simulation with two nodes, in which one node is a miner. At the end of the execution we collect the logs from the two nodes and calculate the propagation time for a block and transaction, by calculating the difference when a block or transaction is sent and received. Following this process we validate the Ethereum models and also verify if BlockSim is working properly, by comparing the results from the simulation with a real network. At the end of the simulation study, we have collected from the simulation and from the real Ethereum network the propagation time for a block and transaction. Therefore, we can evaluate if BlockSim and Ethereum models are valid by comparing the times.

Block propagation time. It starts when the origin node sends his block to a peer, and stop when the block is processed, validated and added to the peer chain. All the created blocks in the simulation and in the real Ethereum network are empty, not containing any transaction.

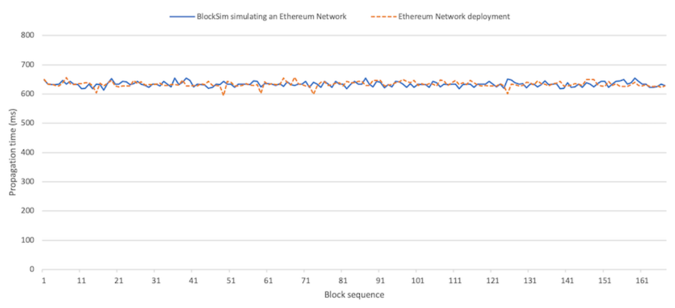
Figure 5a shows the time in milliseconds to propagate a block created by a miner in Ohio and a node receiving the block in Ireland. We achieved an average of 634 ms in the real Ethereum network with a standard deviation of 9.2 ms. The BlockSim simulating the Ethereum models achieved the exact same average of 634 ms, with a similar standard deviation of 8.28 ms.

Figure 5b shows the time to propagate a block created by a miner in Ireland and a node receiving the block in Tokyo. In this result we achieved an average of 836 ms in the real Ethereum network, the exact same result as in BlockSim. However, the real network standard deviation was 6.51 ms, slightly higher than BlockSim (6.17 ms).

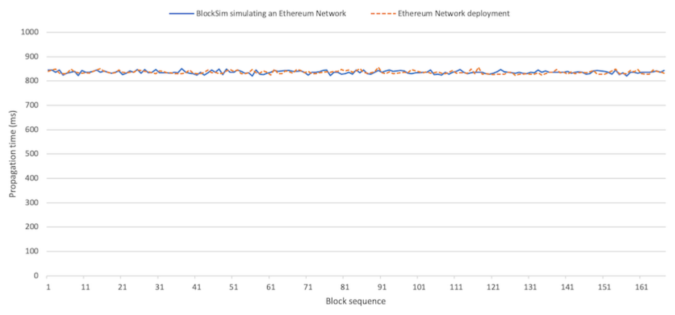
The results in Table II shows that BlockSim runs our Ethereum models presenting slightly low values for standard deviation compared to a real network. These results were expected because our network model does not consider packet loss, routing and other variations that influence packets deliver in a wide area network (WAN).

Transaction Propagation. It starts when the origin node sends a new transaction to a peer, and stop when the transaction is processed and validated by the peer.

Figure 6a shows the time in milliseconds to propagate a transaction created by a node in Ohio and a node receiving the transaction in Ireland. We achieved an average of 93 ms in the real network with a standard deviation of 1.22 ms. The



(a) Between Ohio and Ireland



(b) Between Ireland and Tokyo.

Figure 5: Block propagation time.

Table II: Final results for block propagation between a real Ethereum network and BlockSim simulating an Ethereum network.

	Average		Standard deviation	
	Real network	BlockSim network	Real network	BlockSim network
Ohio and Ireland	634 ms	634 ms	9.2 ms	8.28 ms
Ireland and Tokyo	836 ms	836 ms	6.51 ms	6.17 ms

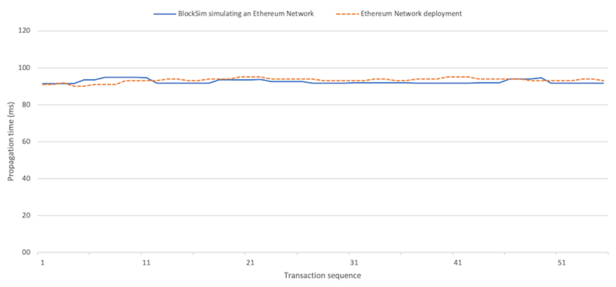
BlockSim simulating the Ethereum model achieved the exact same average of 93 ms, with a similar standard deviation of 1.12 ms.

Figure 6b shows the time to propagate a transaction created by a node in Ireland and a node receiving the transaction in Tokyo. In this result we achieved an average of 98 ms in the real Ethereum network, the exact same result as in BlockSim. However, the real network standard deviation was 1.01 ms, slightly higher than BlockSim (0.81 ms).

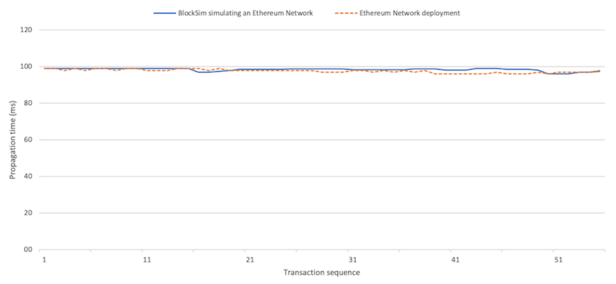
The final results in Table III and Table II answer to our study question. We can observe identical results for average propagation time and a slightly different standard deviation. *Thus, we can conclude that BlockSim runs our Ethereum models presenting an accurate representation of the Ethereum system with regards to block and transaction propagation.*

Table III: Final results for transaction propagation between a real Ethereum network and BlockSim simulating an Ethereum network.

	Average		Standard deviation	
	Real network	BlockSim network	Real network	BlockSim network
Ohio and Ireland	93 ms	93 ms	1.22 ms	1.12 ms
Ireland and Tokyo	98 ms	98 ms	1.01 ms	0.81 ms



(a) Between Ohio and Ireland



(b) Between Ireland and Tokyo.

Figure 6: Transaction propagation time.

B. BlockSim Use Cases

In this section we will explore four real use cases for BlockSim. For each, BlockSim was configured to create 8,000 transactions with a total of 400 nodes: 100 non-miner nodes in Tokyo, 100 in Ireland and 100 in Ohio; 25 miner nodes in Ireland, 25 in Ohio and 50 in Tokyo. We used the same input parameters in Table I.

1) Different Block Gas Limits. A standard transaction in Ethereum has a 21000 gas limit [5] and we consider a standard transactions to have 200 Bytes. These values can be adjusted in the configuration file as input parameter.

The block gas limit represents the maximum amount of gas allowed in a block, it determines how many transactions can fit into a block. For instance, if block gas limit is set to 100, we can fit four transactions with a gas limit of 10, 20, 30 and 40, or only two transactions with 50 gas limit. On public Ethereum blockchain the block 6441886 [31] has a gas limit of eight million, if we consider a standard gas limit for a transaction, we can fit 380 transactions into the block 6441886. The miner can adjust the gas limit by $\frac{1}{1024}$ (0.097%) in either direction [5]. The block gas limit can be set in the configuration file as input parameter.

We set the transaction gas limit to 21000 during all executions. However, for each execution of the simulation we change the value of block gas limit. Table IV shows the block propagation time, when increasing the number of transactions per block.

This use case was simulated in 36 minutes and 21 seconds. Table IV reveal an expected grow of 20 kB block size between each execution, corresponding to an additional number of 100

Table IV: Results for average block propagation with different block gas limit between Tokyo and Ireland.

Transaction gas limit	Block gas limit	Transactions per block	Average block propagation	Block size
21000	2100000	100	847 ms	20.045 kB
	4200000	200	858 ms	40.045 kB
	6300000	300	869 ms	60.045 kB
	8400000	400	879 ms	80.045 kB

transactions. Additionally, an increasing raise in propagation time (10 ms) is observed.

2) Encrypted Network Messages. In this use case we will use BlockSim to observe the impact in performance when a node encrypts and decrypts all the network messages. To simulate this behaviour we have added to our basic node model a fixed delay when receiving and sending a network message. Table V presents the impact in block propagation for two different delays to encrypt and decrypt each message: 100 ms and 50 ms.

Table V: Results for average block propagation with different block encryption and decryption delay between Tokyo and Ireland.

Encrypted	Transactions per block	Encrypt and decrypt delay	Average block propagation time
No	100	-	847 ms
Yes		50 ms	1297 ms
Yes		100 ms	1747 ms

This use case was simulated in 36 minutes and 21 seconds. Table V shows a 25.8% increase in the block propagation time when encrypting and decrypting messages with a delay of 50 ms and a 51.6% increase with a delay of 100 ms.

3) Simplified New Block Delivery. The PV62 protocol [30] is the one used in our models. We adapted[3] and simplified our model to make the node request the full blocks (headers and bodies) when a new block is announced, as illustrated in Figure 7. We created two new network messages: *GetBlocks* that requests the full blocks by the hashes; *Blocks* that sends the requested full blocks. Additionally, the Ethereum node model was changed to respond the new network messages accordingly.

This simplified new block delivery is similar to the Bitcoin protocol (*cf.* Section VI-B). Also, the PV63 Ethereum protocol [30] follow a similar design, and is only used when the node is not synchronise with the rest of the network.

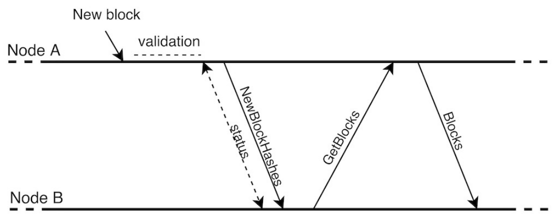


Figure 7: Adapted message exchange protocol to obtain a new block.

[3]Available at: https://github.com/BlockbirdLabs/blocksim/blob/ethereum-use-case/blocksim/models/ethereum

This use case was simulated in 23 minutes and 8 seconds. Table VI shows a 27.9% decrease in block propagation time with our simplified new block delivery. We have obtained a better performance due to the less overhead in the protocol.

The protocol PV62 is used, despite having low performance, because the node when first receives the block header it can perform validations before requesting the block body. Thus, protecting the node from requesting non valid blocks. This characteristic is also important for light client nodes, which in some circumstances does not need the full blocks, only the headers.

Table VI: Results for average block propagation with simplified new block delivery between Tokyo and Ireland.

Protocol	Transactions per block	Block size	Average block propagation time
Standard (PV62)	100	20.135 kB	847 ms
Simplified (Figure 7)			610 ms

4) One Transaction Propagation. Croman *et al.* [1] highlighted an inefficiency in the Bitcoin network layer, that can also be applied to Ethereum network layer. They observed the network layer protocol first propagate all transactions, and then propagate a full block when it is mined, that contains the previously propagated transactions. Thus, requiring each transaction to be transmitted twice. To avoid this process, there is the possibility to rely on a reconciliation protocol in which nodes only fetch transactions that they do not own in a newly mined block [32]–[35].

However, we can use BlockSim to observe the impact on block propagation without the need to implement a complex protocol. We do that by simply not delivering the block body (that contains the previously propagated transactions), as shown in Figure 8. The adapted message exchange protocol[4] simulates the best scenario of a reconciliation protocol when Node A owns all the transactions in the newly mined block.

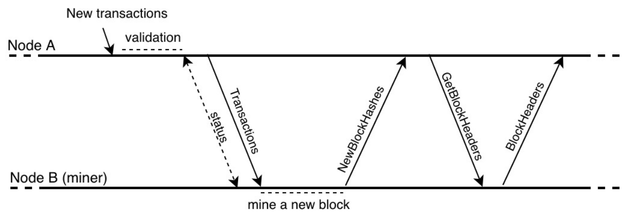


Figure 8: Adapted message exchange protocol that only deliver block header.

This use case was simulated in 22 minutes and 10 seconds. Table VII shows a 29.2% decrease in block propagation time when simulating the impact for one transaction propagation policy.

We can observe in the third and fourth use case a similar block propagation time, despite the differences of block sizes (20.135 kB full block and 0.09 kB block header). A full block

[4]Available at: https://github.com/BlockbirdLabs/blocksim/tree/eth-one-transaction-model/blocksim/models/ethereum

Table VII: Results for average block propagation with one transaction propagation between Tokyo and Ireland.

Protocol	Transactions per block	Block header size	Average block propagation time
Standard (PV62)	100	0.09 kB	847 ms
One Transaction Propagation (Figure 8)			600 ms

transmission only takes approximately 6 ms plus latency, on the other hand, a block header transmission has no impact (tends to zero ms). The major overhead on block propagation time is due to block validation delay (*cf.* Table I) with an average of 299 ms. This leads us to conclude that the size of the messages does not play a big role on the block propagation time.

IX. Conclusion

The paper presented BlockSim, the first effort to provide a blockchain simulator that is not restricted to a concrete blockchain implementation and can be used to model different blockchain systems. This flexibility became possible because we created abstract models that gather common parts in different blockchain systems and made them available, in order to be extended to a specific implementation. The user has a fine-grained control over the created models and events, such as the number of nodes, transactions, or connections among nodes, by using a programmatic interface. In this matter, input parameters can be easily modified to enable a good perception of the impact of certain phenomena. By building a discrete-event simulator, we made it possible to study a large-scale Ethereum and Bitcoin network in a short period of time.

References

[1] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer *et al.*, "On scaling decentralized blockchains," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 106–125.

[2] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 51–68.

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[4] A. M. Antonopoulos, *Mastering Bitcoin*, 2nd ed. O'Reilly, June 2017.

[5] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[6] A. Nordrum, "Wall street occupies the blockchain-financial firms plan to move trillions in assets to blockchains in 2018," *IEEE Spectrum*, vol. 54, no. 10, pp. 40–45, 2017.

[7] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger Fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 30.

[8] J. Sousa, A. Bessani, and M. Vukolić, "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform," *arXiv preprint arXiv:1709.06921*, 2017.

[9] E. Buchman, "Tendermint: Byzantine Fault Tolerance in the age of blockchains," Master's thesis, The University of Guelph, June 2016.

[10] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. USENIX Association, 1999, pp. 173–186.

[11] M. Correia, G. S. Veronese, N. F. Neves, and P. Veríssimo, "Byzantine consensus in asynchronous message-passing systems: a survey," *International Journal of Critical Computer-Based Systems*, vol. 2, no. 2, pp. 141–161, 2011.

[12] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient Byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[13] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary," in *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, 2009, pp. 135–144.

[14] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1085–1100.

[15] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[16] J. Banks, *Discrete-event System Simulation*. Prentice Hall, 2010.

[17] ——, "Introduction to simulation," in *Simulation Conference Proceedings, 1999 Winter*, vol. 1. IEEE, 1999, pp. 7–13.

[18] A. Keränen, J. Ott, and T. Kärkkäinen, "The ONE simulator for DTN protocol evaluation," in *Proceedings of the 2nd international conference on simulation tools and techniques*, 2009, p. 55.

[19] A. Montresor and M. Jelasity, "Peersim: A scalable P2P simulator," in *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*. IEEE, 2009, pp. 99–100.

[20] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[21] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "BFT Protocols Under Fire," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, vol. 8, 2008, pp. 189–204.

[22] 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*, ser. CCS '16. ACM, 2016, pp. 3–16.

[23] A. Miller and R. Jansen, "Shadow-bitcoin: Scalable simulation via direct execution of multi-threaded applications." *IACR Cryptology ePrint Archive*, vol. 2015, p. 469, 2015.

[24] L. Stoykov, K. Zhang, and H.-A. Jacobsen, "VIBES: Fast blockchain simulations for large-scale peer-to-peer networks: Demo," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Posters and Demos*, ser. Middleware '17. ACM, 2017, pp. 19–20.

[25] "SimPy 3.0.10 documentation," https://simpy.readthedocs.io/en/latest/, 2018, [Online; accessed 06-Sep-2018].

[26] K. Finlow-Bates, "Adding trust to cap: Blockchain as a strong eventual consistency recovery strategy," 2017.

[27] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*. IEEE, 2013, pp. 1–10.

[28] "Average Number Of Transactions Per Block - Blockchain," https://www.blockchain.com/charts/n-transactions-per-block, 2018, [Online; accessed 06-Sep-2018].

[29] "Protocol documentation - Bitcoin Wiki," https://en.bitcoin.it/wiki/Protocol_documentation, 2018, [Online; accessed 06-Sep-2018].

[30] "Ethereum Wire Protocol · ethereum/wiki Wiki," https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol, 2018, [Online; accessed 06-Sep-2018].

[31] "Ethereum Block 6441886 Info," https://etherscan.io/block/6441886, 2018, [Online; accessed 02-Oct-2018].

[32] "O(1) block propagation," https://gist.github.com/gavinandresen/e20c3b5a1d4b97f79ac2, 2018, [Online; accessed 10-Oct-2018].

[33] H. D. Johansen, R. V. Renesse, Y. Vigfusson, and D. Johansen, "Fireflies: A secure and scalable membership and gossip service," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 2, p. 5, 2015.

[34] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set reconciliation with nearly optimal communication complexity," *IEEE Transactions on Information Theory*, vol. 49, no. 9, pp. 2213–2218, 2003.

[35] R. Van Renesse, D. Dumitriu, V. Gough, and C. Thomas, "Efficient reconciliation and flow control for anti-entropy protocols," in *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. ACM, 2008, p. 6.