BlockSim: Blockchain Simulator

Carlos Faria
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Blockbird Ventures
Lisboa, Portugal
carlosfigueira@tecnico.ulisboa.pt

Miguel Correia
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Blockbird Ventures
Lisboa, Portugal
miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—A blockchain is a distributed ledger in which participants that do not fully trust each other agree on the ledger's content by running a consensus algorithm. This technology is raising a lot of interest both in academia and industry, but the lack of tools to evaluate design and implementation decisions may hamper fast progress. To address this issue, this paper presents a discrete-event simulator that is flexible enough to evaluate different blockchain implementations. These blockchains can be rapidly modeled and simulated by extending existing models. Running Bitcoin and Ethereum simulations allowed us to change conditions and answer different questions about their performance. For example, we concluded that doubling the number of transactions per block has a low impact on the block propagation delay (10ms) and that encrypting communication has a high impact in that delay (more than 25%).**

***Index Terms*—Blockchain, simulation, Bitcoin, Ethereum**

I. Introduction

Blockchain is a promising new technology, generating widespread interest, and receiving considerable attention in the research community [1], [2]. This interest started with the success of Bitcoin [3], [4], but took speed with the promise of smart contracts [5] and a vast number of applications [6].

A blockchain, or distributed ledger, consists of an append-only data structure that stores an ordered list of transactions (operations, data items), 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, i.e., may fail arbitrarily [7]. At the same time, they need to reach a consensus on the order of transactions, which has to tolerate Byzantine faults. New transactions can be added to the blockchain but it is not possible to modify those already listed, ensuring integrity and non-repudiation of transactions.

The original blockchain was the core of the Bitcoin cryptocurrency system, where nodes store unspent transaction outputs (UTXOs), which are the state of the system. UTXOs can be transferred from a Bitcoin address to another Bitcoin address. Ethereum [5] is a blockchain with more complex states, enabling Turing complete code to be executed within a transaction. This code implements smart contracts [5], [8]. 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. Such blockchains do consensus using a variant of proof-of-work (PoW) [3], [5].

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 evaluations are often based on *emulation*, which imitates the behaviour of a system in a large set of machines [9], [10]. This approach, however, incurs a large overhead and lacks scalability for real world deployments. Besides, power consumption of a large-scale system may have to be taken into account. An 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. 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 that we designate *BlockSim*.[1] The objective is to present the design and implementation of a simulator where blockchains can be implemented in a simple way and have their performance evaluated in different conditions. BlockSim provides a framework and a set of base simulations models common to several blockchains (blocks, transactions, ledger, network). Users can extend these 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 versatile solution without the burden of implementing a simulator from scratch, and can be extended to simulate other blockchains.

We used BlockSim to evaluate the tradeoffs involved in four modifications to Ethereum. These simulations allowed us to conclude that: (1) when doubling the number of transactions in a block, it takes on average 10 ms more to propagate each block; (2) if messages were encrypted (they are not), there would be an increase of at least 25.8% in the block propagation time; (3) disseminating the block header and body together instead of sending them separately, would decrease the block propagation time by 27.9%; and (4) propagating each transaction only once, instead of propagating it a second time in the block, would decrease the block propagation time by 29.2%.

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

II. Background and Related Work

A simulation attempts to reproduce the performance of a system and its progress over time. To do so, simulations run a *model*. A model encompasses a set of assumptions about the operation of the system and can be classified as follows [11], [12]: a *stochastic* model receives input values that follow statistic distributions, leading to probabilistic outputs, whereas a *deterministic* model does not use random values; a *static* model represents a system at a particular point in time and a *dynamic* model over a certain time frame; a *discrete-event* model describes the system as a sequence of events, so it is possible to jump in time from one event to the next, whereas a *continuous* model tracks the system states over time.

Simulators like The ONE [13], PeerSim [14], CloudSim [15] and BFTSim [16] are useful tools in the development of protocols and systems for opportunistic networks, peer-to-peer networks, cloud computing and Byzantine fault-tolerant replication, respectively. There are also simulators created to perform evaluation on the impact of network-layer parameters on the security of Bitcoin PoW, such as Bitcoin Simulator [17], Shadow-Bitcoin [18] and VIBES [19].

All these simulators follow a discrete-event simulation model. All create a model for certain resources, such as: network latency, bandwidth, ad-hoc networks and CPU. Each resource model is described by 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.*

III. Modelling Random Phenomena

A phenomenon is said to be random if there is uncertainty associated to its occurrence, but also an observable pattern. When creating our models we aim to mimic this observable pattern. 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 derive a probability distribution that our models will use. For instance, in order to calculate the throughput when sending and receiving messages over TCP between different geographic locations, our procedure was: (1) setup two instances on the desired geographic locations with the *iPerf3* bandwidth measurement tool; (2) measure the throughput between the two instances using *iPerf3*, at each hour, for 24 hours; (3) at the end of 24 hours, we collect the *iPerf3* logs from the two instances; (4) use the Kolmogorov–Smirnov test [20] to know which distribution best fits the samples collected in Step 3; (5) the distribution name and parameters are then consumed by the simulator, to extrapolate the values of throughput 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 this information is then used by our models.

IV. BlockSim Architecture

BlockSim is a simulation framework that aims to provide assistance in the design, implementation, and evaluation of existing or new blockchains. The simulator follows a stochastic simulation model, being able to represent random phenomena by sampling from a probability distribution. Our models are considered dynamic, as they can represent the system over a certain interval. A discrete-event simulation model is suitable to model a blockchain system, since it changes states at discrete points in time, independently of the real time they take to happen. Therefore, the simulator can keep track of thousands of nodes and events that only change states.

BlockSim has the architecture of Figure 1. The figure shows the main components, connectors and interfaces of the implementation. The next sections present the components.

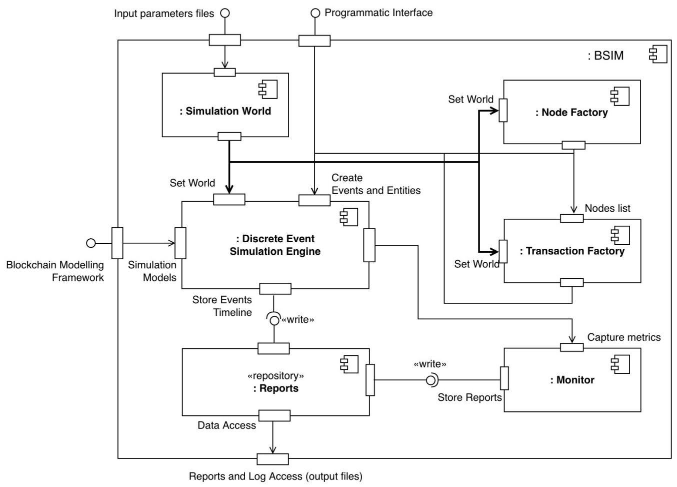

Figure 1: BlockSim architecture.

A. Discrete Event Simulation Engine

The core of a discrete event simulator is a *Discrete Event Simulation Engine* (DESE). The basic functionality of a DESE is similar in most discrete event simulators, so we did not design DESE from scratch, but leveraged an existing framework called *SimPy* [21]. 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 using the Python *yield* keyword.

The DESE component 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. The BlockSim user can also use all the functionality from SimPy when creating new models. However, SimPy is a framework to build arbitrary models or simulators, whereas BlockSim provides a more tailored framework to simulate blockchains by providing additional components (next sections).

B. Simulation World

The *simulation world* component (Figure 1) is responsible for handling the input/configuration parameters of the simulations. These parameters are necessary for the simulation models, which are defined using the *Blockchain Modelling Framework* (*cf.* Section IV-F). Many of these parameters characterize random phenomena (*cf.* Section III).

These parameters are organised as a set of files: (a) *Configuration file*: name of blockchain being simulated, node locations, configurations that depend on the blockchain being simulated (e.g., probability of orphan blocks, message size, block size, and gas limit); (b) *Delays file*: probability distributions for time to validate transactions, time between blocks, etc.; (c) *Latency file*: probability distributions corresponding to the latency between possible locations for nodes; (d) *Throughput received and sent files*: probability distributions of reception throughput and another to sending throughput between possible locations for nodes.

The user needs to assign 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 passed to 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 transactions, which are again modelled as a random phenomena. These transactions are broadcasted by a random node on a list, when the simulation is running. The *node factory* creates nodes that are used during the simulation. The user can specify the location, number of nodes and identifier.

D. Programmatic Interface and Simulation Example

The programmatic interface is the main interface available to the user. Using *Python* language and SimPy [21], users can write their own models, use the existing ones to define their own blockchain system, or modify aspects of models already implemented. This interface is also responsible for starting the simulation. When started, the DESE will consume the events and entities before initialising the simulation, to know which models will be used.

Listing 1 shows how the user can define a simulation, starting by creating the simulation world. The simulation world

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

net = Network(world.env, "ETH Network")
miners = {
    "Ohio": {
        "how_many": 0,
        "mega_hashrate_range": "(20, 40)"
    }
}
non_miners = {
    "Tokyo": {
        "how_many": 3
    }
}
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)
trans_factory.broadcast(7, 6, 300, nodes_list)
world.start_simulation()
```

Listing 1: Example definition of a simulation.

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 the 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 6 transactions every 5 minutes, 7 times, in a total of 42 transactions broadcasted during the simulation. Finally, the function gives the order for DESE to start the simulation.

E. Monitor and Reports

The goal of the *monitor* is to capture metrics during the simulation. Examples are: 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 be able to simulate arbitrary blockchains, we have to consider different layers. We can identify the following major layers in a blockchain system: the *Node layer* specifies the responsibilities and how a node operates when being part of a given network; the *Consensus layer* specifies the algorithms and rules for a given consensus protocol; the *Ledger layer* defines how the ledger is structured and stored; the *Transaction and block layer* specify how information is represented and transmitted; the *Network layer* establishes how nodes communicate with each other; and the *Cryptographic layer* defines what cryptographic functions will be used and how. Models

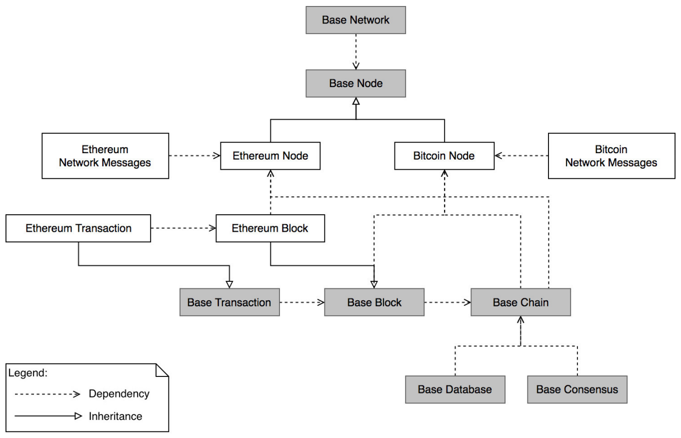

Figure 2: Class diagram of the modelling framework.

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 in grey are presented in the following sections. They can be extended to simulate specific blockchain implementations.

1) Chain Model: Reproduces 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 before the parent due to network delays. 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 instantiating 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: Provides the rules applied when validating blocks and transactions. In the abstract model, we opted for not performing 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 (for PoW). In this base model we opted for using 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 expression: $\mathit{difficulty} = P_d + (B_{TS} - P_{TS})$. This expression is inspired by the mechanisms used in Ethereum [5] and Bitcoin [3] for incrementing the difficulty of a block when it is created in less time that the target they consider (e.g., 10 min. in Bitcoin). 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) Network Model: The network model is responsible for knowing the state of each node during the simulation, establishing the connection channels between nodes, and applying a network latency on the messages being exchanged. The network latency delay applied depends on the geographic location of the destination and origin nodes. The simulation framework gives the user the freedom to choose what nodes to connect; it does not implement a specific peer-to-peer (P2P) discovery protocol [22]. It is possible to define an additional model to simulate a particular P2P discovery protocol.

The mining process of a new block, i.e., the solution of a cryptopuzzle to obtain a PoW (in blockchains that use this consensus approach), is in part simulated 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 its 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, i.e., of blocks that have to be discarded due to a fork in the chain of blocks due to the probabilistic nature of PoW [4]. The network model simulates this behaviour by selecting two nodes to broadcast its candidate blocks. This event only occurs with a predefined probability [22], [23], set in the configuration.

4) Node Model: The node model is responsible for simulating the functionality of a node operating in a P2P network. When a simulation starts, a node connects to a list of nodes. 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. The node model is also responsible for applying 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. 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 higher level models, something that we will explore in the next two sections respectively for Bitcoin and Ethereum.

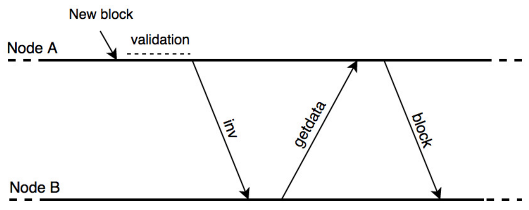

Figure 3: Messages exchanged in Bitcoin to get a new block.

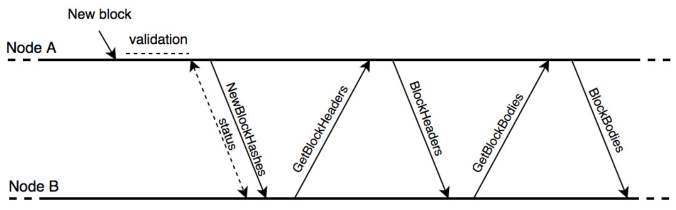

Figure 4: Messages exchanged in Ethereum to get a new block.

V. Modelling Bitcoin

Using the Blockchain Modelling Framework, we can easily model the Bitcoin blockchain by reusing the base models represented in Figure 2. We configure the simulation world with the block size limit and the probability distribution of the number of transactions per block, extrapolated from the last two years of the public Bitcoin network [24]. As the block size limit in Bitcoin is 1 MB [3], we take from the probability distribution the number of transactions; however, if the user chooses to simulate an environment with a 2 MB block, we multiply by 2 the average number of transactions. With this, we can assess the performance with different block size limits.

The Bitcoin network protocol [25] defines the messages that are exchanged between nodes. For each message, the name, payload, and size are defined. We define the following messages in our model: *inv* to advertise knowledge of new transactions or blocks; *getdata* to retrieve the content of a specific block or transaction; *tx* to send a single transaction; *block* to send a specific body of a block; *headers* to send block headers; *getheaders* to request block headers. The user can easily modify the message sizes in the configuration file. The model then reads configurations through the world variable (*cf.* Section IV-B) to calculate the size of each message. The sizes are taken from the Bitcoin documentation [25].

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 for the Bitcoin protocol. Bitcoin nodes can be divided into two major groups (reality is more complicated, with nodes implementing a sets of roles [4]): 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 each new transaction in a transaction queue. The creation of a candidate block is the process of collecting the pending transactions and fitting them in a block. The node only broadcasts its candidate block to the network, when selected by the Network base model; this process simulates the mining of a new block. The simulator does not execute cryptographic operations or validations; it only applies a delay corresponding to the process of validation in a real system, which is measured beforehand (*cf.* Section III).

The process of announcing a new block is shown in Figure 3, starting by Node A announcing a new block to its neighbours with an *inv* message. When Node B receives this *inv* message, it calls Node A using a *getdata* message to send the entire block it announced. Node A receives the *getdata* message and sends the entire block to Node B using a *block* message. When Node B receives this message, it validates the block and adds it to its chain (if valid). The same process works for new transaction(s) announced by a node on the network. When a miner node receives a new transaction in a *tx* message, it adds it to a transaction queue.

VI. Modelling Ethereum

The process of modelling Ethereum using the Blockchain Modelling Framework is similar to Bitcoin's (Figure 2). The simulation world in this case is configured also with the block gas limit and the start gas for every transaction. The start gas (or gas limit) is the maximum amount of gas the originator of the transaction is willing to pay. 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. Gas is consumed by the execution of Ethereum virtual machine (EVM) instructions. The value of gas consumed per instruction varies from 1 to 32,000 [5].

The Ethereum Network protocol (PV62) [26] messages are defined in our model as following: *Status* informs a node of its current state, sent after the initial handshake and prior to any other message; *NewBlockHashes* advertises one or more new blocks that have appeared on the network; *Transactions* sends one or more transactions; *BlockBodies* sends block bodies; *GetBlockBodies* are used to retrieve specific block bodies using block hashes; *BlockHeaders* sends block headers to a node; *GetBlockHeaders* requests block headers. The sizes are taken from the documentation [26].

The Ethereum Node model inherits the base node model. Thus, we can focus on building a specific model to the Ethereum protocol. Ethereum nodes can also be divided into two classes – miner node and non-miner node –, providing the same functionality as the Bitcoin node model.

The process of announcing a new block is shown in Figure 4, starting with Node A announcing a new block to its neighbours (*NewBlockHashes* message). When Node B receives such a message, it calls Node A sending a *GetBlockHeaders* message with the block header of the block it announced. Node A sends the block header to Node B using a *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 in a message *BlockBodies*. When Node B receives the block body, it starts a validation process and adds it to its chain. The process of announcing a new transaction has less overhead than announcing a new block. Node A receives a new transaction, validates it, then uses the *Transactions* message to broadcast the full transaction to its neighbours. When a miner node receives a new transaction, it adds it to a transaction queue.

The *Ethereum Transaction model* extends the base transaction model only by 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 only by adding the same attributes.

Table I: Input parameters for the probability distributions

	Distribution	Location	Scale	Other 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 Ohio-Ireland	Normal	73.70 ms	0.09 ms	-
Throughput Ohio-Ireland	Beta	39.13 Mbps	59.02 Mbps	α = 0.463 β = 0.461
Latency Ireland-Tokyo	Normal	105.42 ms	0.23 ms	-
Throughput Ireland-Tokyo	Beta	51.33 Mbps	89.06 Mbps	α = 0.512 β = 0.914

Table II: Block/transaction propagation times (ms) in a real Ethereum network deployment and simulated by BlockSim.

		Average		Standard deviation	
		Real	BlockSim	Real	BlockSim
Block	Ohio and Ireland	634	634	9.2	8.28
	Ireland and Tokyo	836	836	6.51	6.17
Transaction	Ohio and Ireland	93	93	1.22	1.12
	Ireland and Tokyo	98	98	1.01	0.81

VII. Evaluation

The goal of BlockSim is to provide an accurate representation of the performance of a real blockchain system. Therefore, Section VII-A does a validation of BlockSim and Section VII-B evaluates four use cases with BlockSim. All the BlockSim executions were conducted on a PC with a 2 GHz Intel Core i7 processor and 8 GB RAM.

A. BlockSim Validation

In order to validate that BlockSim provides realistic results, we compare simulations of Ethereum models with values obtained in a private Ethereum network. We follow the usual steps for a simulation study: (1) Clearly identify the question to be answered, in our case "how long it takes to propagate a block and a transaction from one node to another?"; (2) Conceptualise the underlying models needed to answer the question, in this case block, transaction, network, messages, node, and consensus models; (3) Determine the input parameters for the models, here block and transaction gas limit, message size, distribution of delay to validate a block and transaction, distribution of latency and throughput between each node geographic locations; (4) Collect data from existing deployments for each input parameter (explained in Section III); (5) Code the conceptual models of Step 2, in this case we used the BlockSim Modelling Framework to create the specific Ethereum models for our study; (6) Perform a verification of the models to understand if they are performing properly (if not repeat Step 5); (7) Check 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.

We started by capturing block interval on the Ethereum *mainnet*. Next, we changed the Ethereum client reference implementation, to get the instant when a block or transaction is sent to its peers and when it is received.

We then deployed a private Ethereum network using the modified Ethereum client in Amazon Web Services (AWS). The private network had two instances, each one with 2 virtual CPUs, 4 GB RAM with 8 GB SSD. The goal was 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 collected the time measurements from the two nodes and calculated the propagation time for a block and transaction (the difference when a block or transaction is sent and received).

We configured the simulation with the parameters in Table I, for an Ethereum network with one miner and one non-miner nodes. Following this process we validated the Ethereum models and also verified if BlockSim operates 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. At this stage it is possible to evaluate if BlockSim and the Ethereum models are valid by comparing the times.

For lack of space we present only a very brief summary of our results in Table II. We can observe identical results for average propagation time and slightly different standard deviations. Thus, we can conclude that BlockSim is able to obtain results according to those obtained in real systems.

B. BlockSim Use Cases

In this section we present four 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 parameters of Table I.

1) Different Block Gas Limits: A standard transaction in Ethereum has a 21,000 gas limit [5]. 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, in the public Ethereum the block 6441886 [27] has a block gas limit of 8 million, so with the standard transactions, we might fit 380 transactions into that block. Additionally, the miner can adjust the block gas limit by 1/1024 (0.097%) in either direction [5].

In our simulation, we set the transaction gas limit to 21,000 during all executions, but changed the block limit from 2.1

Table III: Block propagation time with different gas limits (between Tokyo and Ireland).

Trans. gas limit	Block gas limit	Trans. per block	Block propagation time	Block size
21000	2,100,000	100	847 ms	20.045 KB
	4,200,000	200	858 ms	40.045 KB
	6,300,000	300	869 ms	60.045 KB
	8,400,000	400	879 ms	80.045 KB

Table IV: Block propagation time with different block encryption and decryption delays (between Tokyo and Ireland).

Encrypted	Trans. per block	Encrypt / decrypt delay	Block propagation time
No	100	-	847 ms
Yes		50 ms	1297 ms
Yes		100 ms	1747 ms

million to 8.4 million. The simulation took 36 minutes and 21 seconds. Table III shows the block propagation time, when increasing the number of transactions per block. We can observe an expected growth of 20 KB block size between each execution, corresponding to an additional number of 100 transactions. Additionally, an increasing raise in propagation time of 10 ms was observed. This is a low increase.

2) Encrypted Network Messages: In this use case we used BlockSim to assess the impact in performance when nodes encrypt and decrypt all network messages, something that Ethereum and Bitcoin does not do. To simulate this behaviour we have added to our basic node model a fixed delay when receiving and sending a network message. Table IV presents the impact in block propagation for two different delays to encrypt and decrypt each message: 100 ms and 50 ms. Table IV 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 Block Delivery: Our Ethereum models use the PV62 communication protocol [26]. We adapted our model to make the node request full blocks (headers and bodies) when a new block is announced, as shown in Figure 5, instead of requesting headers and bodies individually. We created two new messages in our model: *GetBlocks* that requests the full blocks by the hashes; *Blocks* that sends the requested full blocks. This simplified new block delivery is similar to the Bitcoin protocol (*cf.* Section V). Also, the PV63 Ethereum protocol [26] follow a similar design, but is only used when the node is not synchronized with the rest of the network. The protocol PV62 is used in Ethereum because when the node 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 do not need the full blocks, only the headers.

This use case was simulated in 23 minutes and 8 seconds. Table V 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.

4) One Transaction Propagation: Croman *et al.* [1] identified an inefficiency in the Bitcoin network that also exists in Ethereum. They observed the network layer protocol first propagates all transactions, then propagates 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 the transactions from the newly minted block that they do not own [28]–[31].

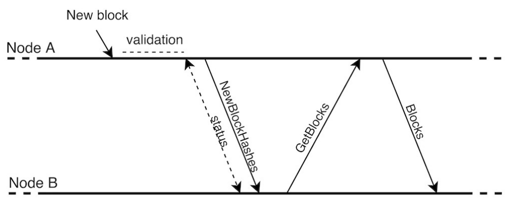

Figure 5: Simplified protocol to obtain a new block.

Table V: Block propagation time with simplified new block delivery (between Tokyo and Ireland).

Protocol	Trans. per block	Block size	Block propagation time
Standard (PV62)	100	20.135 KB	847 ms
Simplified (Figure 5)			610 ms

We used BlockSim to assess 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 6. The adapted message exchange protocol simulates the best scenario of a reconciliation protocol when Node A owns all the transactions in the newly mined block. This use case was simulated in 22 minutes and 10 seconds. Table VI 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 cases 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 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.

VIII. Conclusion

The paper presented BlockSim, to the best of our knowledge the first effort to provide a blockchain simulator that is not restricted to a concrete blockchain implementation, i.e., that 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,

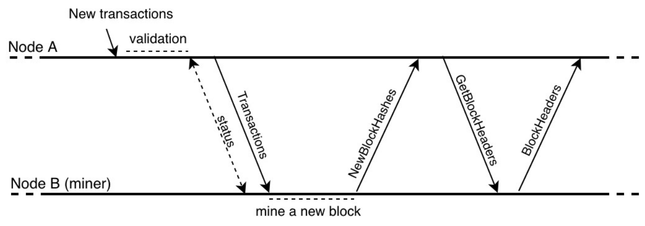

Figure 6: Protocol that only delivers the block header.

Table VI: Block propagation with one transaction propagation (between Tokyo and Ireland).

Protocol	Trans. per block	Block header size	Block propagation time
Standard (PV62)	100	0.09 KB	847 ms
Transaction Propagation (Fig. 6)			600 ms

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 Ethereum variants in a short period of time. An open challenge is to adapt BlockSim for permissioned blockchains that do not use PoW, but more convencional Byzantine fault-tolerant consensus algorithms [32].

Acknowledgements We warmly thank Alysson Bessani and anonymous reviewers for feedback on previous versions of this work. This work was supported by the European Commission under project QualiChain (SEP-210506957) and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2019 (INESC-ID).

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] M. E. Peck, "Blockchains: How they work and why they'll change the world," *IEEE Spectrum*, vol. 54, no. 10, pp. 26–35, 2017.

[7] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan-Mar 2004.

[8] N. Szabo, "The idea of smart contracts," 1997, http://szabo.best.vwh.net/smart_ contracts_idea.html.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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*, 2008, pp. 189–204.

[17] 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*. ACM, 2016, pp. 3–16.

[18] 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.

[19] 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*. ACM, 2017, pp. 19–20.

[20] H. W. Lilliefors, "On the Kolmogorov-Smirnov test for normality with mean and variance unknown," *Journal of the American Statistical Association*, vol. 62, no. 318, pp. 399–402, 1967.

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

[22] C. Decker and R. Wattenhofer, "Information propagation in the Bitcoin network," in *IEEE 13th International Conference on Peer-to-Peer Computing*. IEEE, 2013, pp. 1–10.

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

[24] "Average number of transactions per block: Blockchain," https://www.blockchain.com/charts/n-transactions-per-block, 2018, [Online; accessed 06-Sep-2018].

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

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

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

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

[29] 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*, vol. 33, no. 2, p. 5, 2015.

[30] 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.

[31] 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*, 2008.

[32] 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.