TÉCNICO LISBOA

BlockSim: Blockchain Simulator

Carlos Sérgio Figueira Faria

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Professor Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. Alysson Neves Bessani

November 2018

Acknowledgments

I would like to thank my supervisor Professor Miguel Correia, who was always supportive, guided and positive thinking all along this journey.

I would like to thank mother, grandmother, brother and girlfriend for always helping me keep the focus and strength to finish my goals and for always help me walk through my life obstacles.

I would like to also thank my closest friends and colleagues whose feedback and support was valuable.

Resumo

Os sistemas *blockchain* têm recebido um incomensurável interesse por parte da indústria e da academia. *Blockchain* é um registo distribuído onde os participantes, não confiando em outrem, concordam no estado global desse mesmo registo. Dada a rápida expansão desta nova área torna-se importante, e desafiante, entender as possibilidades desta tecnologia. De forma a abordar esta questão, esta tese apresenta um simulador de eventos discretos, suficientemente flexível para avaliar diferentes implementações de blockchain. Estas *blockchains* podem assim rapidamente ser modeladas e simuladas através da extensão dos modelos existentes. O simulador foi utilizado para simular redes *Bitcoin* e *Ethereum*, e os resultados foram comparados com medições obtidas numa rede real. Os modelos de simulação do *Bitcoin* e *Ethereum* oferecem a possibilidade de alterar as condições de funcionamento e responder a diferentes questões ou realizar uma avaliação do sistema. Este processo pode ser aplicado a qualquer sistema *blockchain*.

Palavras-chave: blockchain, simulação, bitcoin, ethereum, desempenho

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 thesis 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.

Keywords: blockchain, simulation, bitcoin, ethereum, performance

Contents

List of Tables

List of Figures

Chapter 1

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].

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.

1.1 Topic Overview

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 [4] 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 [5]. 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 [6, 7] and Tendermint [8]. These permissioned blockchains are characterised do deploy deterministic mechanisms, such as Byzantine Fault-Tolerant (BFT) consensus protocols [9, 10, 11, 12].

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 reproduce the behaviour of a system in a large number of machines [13, 14]. 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. Simulators like The ONE [15], PeerSim [16], and CloudSim [17] are important tools in the development of protocols and systems for opportunistic networks, peer-to-peer networks, and cloud computing, respectively. 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 thesis proposes a blockchain simulator, *BlockSim*. The objective of the thesis 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.

1.2 Objectives

This thesis addresses problems related to design and implementation decisions of specific blockchains, and challenges regarding scalability. The main objective of this thesis is to provide a simulator capable of evaluating different blockchains in different environment conditions, enabling, thus, a richer understanding of this technology. The concrete objectives for the BlockSim are:

- To provide a simulator capable to run user defined simulation models.
- To provide a simulator capable to run thousands of nodes on a single host.
- To provide the possibility of users change the simulated environment conditions.
- Simulation should provide an accurate representation of a real blockchain system.
- Simulation should be performed in reasonable time.
- The simulator has to provide a report with the simulated results when concluded.

1.3 Thesis Outline

This dissertation is organised as follows: Chapter 2 explores current blockchain implementations and analyses existing simulation systems and mechanisms. Chapter 3 describes the architecture and present the implementation decisions of BlockSim with the existing simulation models. Chapter 4 validates BlockSim with respective simulation models and concludes with real use cases. Finally, Chapter 5 presents the conclusions of this dissertation.

Chapter 2

Background

This section provides an understanding of how a blockchain works. Several existing implementations will be introduced. Moreover, core concepts regarding simulation will be addressed, namely, how they can be applied, their advantages and current simulators.

This chapter is organised as follows. Section 2.1 explores Bitcoin, defining the core concepts of a blockchain. Section 2.2 presents the Ethereum architecture and upcoming improvements. Section 2.4 explores concepts about simulation and a final discussion will draw a comparison to simulators closely related between the adopted solution.

2.1 Bitcoin

Bitcoin [3, 18] is the first permissionless blockchain, allowing any participant to join and leave the network without permission.

The Bitcoin network is peer-to-peer (P2P), where all the participants in the network are equally privileged and equally powerful. The participants share the burden of providing network services, without the need for a centralised service, and they are both suppliers and consumers of resources. These participants, or nodes, communicate with each other primarily via the Internet, using the Bitcoin protocol.

Bitcoin can be viewed as a *state transition system*, where there is a *state* which declares the ownership status of all existing bitcoins, and a *state transition function*, that takes a state and a transaction and outputs a new state [4].

Although nodes are equally privileged in the Bitcoin P2P network, they may take on four different roles: wallet, miner, full blockchain database and network routing. All nodes have the *network routing role*, having the responsibility to propagate, and validate, transactions and blocks, as well as discovering and maintaining connections to peers. These roles are required

for a node be able to participate in the network and in the consensus protocol. Some nodes with the *full blockchain database role*, also called *full nodes*, maintain a complete and up-to-date copy of the ledger and they can autonomously verify any transaction without external reference. Finally, there are nodes with the role of *mining*, that collect and aggregate all transactions on the network into a block. After checking the validity of transactions in a block, the mining node starts the consensus protocol, with the purpose of appending its block to the blockchain.

Bitcoin uses *Proof-of-Work* (PoW) to reach consensus among the participants in the network. This means that a mining node needs to solve a computationally hard puzzle, in order to append its block into blockchain and take a reward, which is an incentive to maintain the security of the Bitcoin network.

Bitcoin gathers concepts and technologies which are the basis of a cryptocurrency system. It also serves as a reference and motivation for all blockchain technologies. For this reason, we will introduce these core concepts to deeply understand how a blockchain works.

2.1.1 State Transition Function

A *state transition function* (STF) receives as input a *state* and a *transaction*, and outputs a new state. An example can be drawn to a banking system, where the state is the account balance, and a transaction a statement to move 10$ from Alice to Bob. The STF subtracts the 10$ in Alice's account and increases 10$ in Bob's account. However, if Alice's account balance has less than 10$, the STF returns an error.

In Bitcoin, the collection of all digital coins that have not yet been spent, is known as *Unspent Transaction Outputs* (UTXO). Each UTXO has an owner identified by his cryptographic public key, broadly known as *bitcoin address*.

A transaction contains one or more inputs. Each input references an existing UTXO and a cryptographic signature, obtained with its owners private key. A transaction also contains one or more outputs, each one containing a new UTXO.

To validate a transaction, the STF can not return an error. A STF can result in an error in the following situations:

1. For each input in a transaction:

   If the referenced UTXO is not part of the state.

   If the signature does not match the owner of the UTXO.

2. If the sum of the values of all inputs is less than the sum of the value of all outputs.

The first situation prevents senders from sending digital coins that do not exist and from spending digital coins that they do not own. The second validates the value of the transaction. Finally, the STF returns the final state with UTXO and inputs removed.

Normally the value of a UTXO is larger, or smaller, than the desired value of a transaction. Returning to the previous example, let us suppose that Alice wants to send 4.5 bitcoins to Bob. Alice needs to look for her UTXOs in the entire public ledger, which can be spent with the cryptographic keys controlled by her. Alice will not be able to send 4.5 bitcoins precisely. The smallest amount that she can send is 5 bitcoins (4 bitcoins UTXO plus 1 bitcoin UTXO). She then creates a transaction with two inputs and two outputs. The two inputs are the two UTXOs (4 bitcoins in a UTXO and 1 in another bitcoin UTXO). The first output is the exact amount that she wants to send (4.5 bitcoins) to Bob's associated address, while the second is the remaining 0.5 bitcoins, to be sent back to her. If Alice does not claim this change - by sending it to an address owned by her, the miner mining the block containing the transaction will be able to claim the change, as it will be considered a transaction fee. A transaction fee consists of the difference between inputs and outputs. Further explanations will be given about transactions fees.

The process of searching across all public ledger and choosing UTXOs to satisfy a transaction is performed by the wallet software.

2.1.2 Distributed Ledger

A *distributed ledger* is essentially a replicated append-only data structure that stores an ordered list of transactions. For instance, a simple ledger can record monetary transactions between banks, or exchange goods between known parties. In blockchains, the ledger is replicated by every node in the network, and transactions are typically assembled into blocks. Blockchain can be visualised as a vertical stack, as shown in Figure 2.1, with layers of blocks and the first block as the foundation of the stack.

Each block is identified by a hash, generated using a cryptographic hash algorithm on the header of the block. These blocks are linked to the previous block in the chain (also known as parent block). To create that link, each block contains the hash of its parent inside its own header, in a field called *previous block hash*. This sequence of hashes, linking each block to its parent, creates a chain of blocks which can be traced back to the first block, the *genesis block*. Furthermore, each block contains a timestamp of when a block was created and a summary of all transactions. To this end, a *Merkle tree* is used, where leaves represents transactions. The process of pairing and hashing the results produces a *Merkle root* hash, included in the header

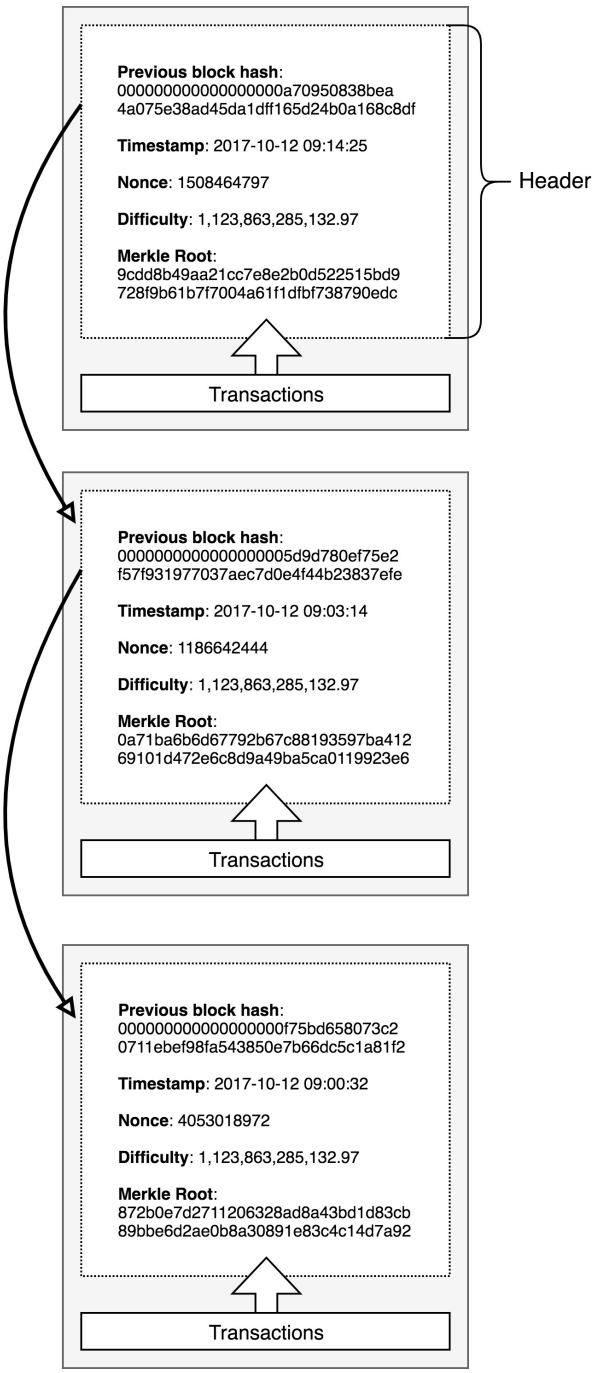


Figure 2.1: Representation of a chain of three blocks as stored in Bitcoin blockchain.

of the block.

In the case of Bitcoin, each block has a single parent, yet can temporarily have multiple children. This occurs when different blocks are discovered, almost simultaneously, by different miners. This transitionary situation is commonly known as *blockchain fork*. Eventually, only one children block becomes part of the blockchain and the fork is resolved.

The existence of a long chain of blocks makes the blockchain immutable, a key security feature on any blockchain system. This is accomplished by the previous block hash field, which remains inside of the block header, thereby affecting the current block hash. An update to the parent will lead to a change in it's hash, and a consequent change in the previous block hash pointer of the child. This change also applies to any further children - namely to changes in the pointer of the grandchild. This cascade effect exist to force the recalculation of all subsequent blocks. Such recalculation are strenuous, computation-wise.

2.1.3 Mining and Consensus

The goal of Bitcoin is to build a decentralised currency system in a permissionless environment, supported by a distributed ledger.

Mining nodes have the role of reaching agreement in a decentralised network. These nodes follow specific rules to validate new transactions and assemble them on blocks that can be added to the distributed ledger. New blocks with transactions appear in the network every 10 minutes, on average. This process is broadly known as *mining*, as the reward for accomplishing it is to give the miner new generated digital coins, creating, this way, new bitcoins. Additionally, transaction fees are collected by the miner who mines the block. These fees serves as an incentive to include a transaction into the next block and also to prevent abuse of the system by imposing a cost on every transaction.

In order to earn these rewards, miners need to compete among themselves to solve a mathematical problem based on a cryptographic hash algorithm. The solution for this problem, is called a Proof-of-Work (PoW). The PoW is included in the new block and acts as proof that the miner expended significant computing effort to come-up with the solution. Solving the PoW algorithm gives the right to record transactions on the blockchain. Mining is an important task towards solving the problem of network-wide consensus without a central authority.

Some nodes have a full copy of the public ledger that the node itself can trust. In these cases, the blockchain is assembled independently by each node, without any central authority intervening in the process.

Bitcoin's main innovation rests on the concept of a decentralised mechanism for emergent

consensus. It can be considered emergent since consensus is not fully expressed - i.e., there is no election, or a fixed moment that consensus occurs. Instead, the consensus is an emergent result of the asynchronous interactions of thousands of independent nodes, all following the same rules.

Bitcoin decentralised consensus occurs because each node executes, independently, the following processes:

1. Verification of each transaction received.
2. Aggregation of the valid transactions into new blocks, coupled with the Proof-of-Work.
3. Validation of new blocks.
4. Selection of the chain with the most accumulated work demonstrated by the Proof-of-Work.

The first process executes a verification on each transaction received by a node, applying the state transition function. If the STF returns an error, the verification of the transaction fails, and it is ignored by the node.

The second process is executed by a miner, responsible for collecting all verified transactions, and aggregating them into a *candidate block*. The block is only considered a valid block if the miner succeeds in finding a solution to the puzzle. The PoW algorithm uses the hash algorithm SHA256 to calculate the hash of the candidate block header and sees if it is smaller than the current target. Thus, the miner goes on calculating hashes with random nonces in the header of the candidate block until it outputs a hash function smaller than the current target. This target is periodically adjusted to meet the 10 minutes block interval on the network.

After the miner successfully finds a nonce that produces the hash that meets the target, it can broadcast its valid block to the network, and it will be added to the blockchain. From the perspective of other nodes in the network, if someone has succeeded assembling a block with the right nonce, and the hash of the block is below the current target, then this constitutes proof that a certain amount of work was done, and it only require a hash computation to verify if it is in fact the correct nonce. If this amount of work would not been taken, a Sybil attack [19] could be performed, by trying to subvert the network by creating a large number of nodes to gain a disproportionately large influence on the network.

The third process is an independent validation of new blocks. When a new block appears in the network, each node validates the block before propagating it to its peers. This process ensures that only valid blocks are propagated across the network. This helps to ensure that miners that act honestly have theirs blocks added to the public ledger and they end up receiving the reward. On the other hand, those who act dishonestly have their blocks rejected, losing

the reward and wasting computational resources finding a PoW solution. The algorithm for checking if a block is valid, works as follows:

1. Verify if the previous block referenced by the block exists and is valid.
2. Verify if the block header is less than the target (validating the PoW).
3. Verify if the block timestamp is greater than the previous block and less than two hours in the future (allowing for time errors).
4. Validate all the transactions within the block, using the state transition function.

The last and fourth process executed by each node in the network, is the assembly of valid blocks into chains, and the selection of the chain of blocks which has the most cumulative Proof-of-Work, known as the *longest chain* or greatest cumulative work chain. Blockhains are particularly susceptible to having forks. As an example, a node thinks block A is the latest block and others will think it is block B. This may occur due to an adversary attempting to disrupt the ledger or simply due to network latency.

Another type of temporary fork occurs when there are two roughly equally valid candidate blocks to be appended to the blockchain. This event can occur when two blocks are announced at the same time at distinct geographic locations, leading to the creation of *orphan blocks* or *stale blocks*. The probability of an orphan block in Bitcoin is between 1.69% and 1.741% [20, 21].

There are two additional types of forks, *soft forks* and *hard forks*. As expressed by Antonopoulos in [18] a soft fork "is a forward-compatible change to the consensus rules that allows unupgraded clients to operate in consensus with new rules". On the other hand, hard forks are non-backwards compatible, and may induce violation of safety.

2.1.4 Bitcoin Scripting Language

The Bitcoin scripting language, called *Script*, is a simple stack-based programming language. Besides being able to be owned by a public key, a UTXO may also be owned through a complex script. When a transaction is validated, the script that unlocks the UTXO must be satisfied. Therefore the transaction must provide data that satisfies the script.

For instance, all major transactions that occur on the Bitcoin network spend outputs locked with a Pay-to-Public-Key-Hash (P2PKH) script. Any UTXO locked by a P2PKH can be unlocked (spent) by submitting a public key and a digital signature created by the corresponding private key. More complex scripts already exist [18]. Multi-signature can be seen as an example. Multi-signature consists of a script that requires signatures from two of a given three private keys to validate a transaction.

The Script language contains many operators, but is intentionally limited. There are no complex flow control and loop instructions. This limited complexity ensures that the language is not Turing complete, and has predictable execution times. The Turing incompleteness ensures that the language cannot be used to produce infinite loops or logic bombs, which might lead to a denial-of-service attacks against the Bitcoin network, because every node validates every transaction.

A UTXO has only two states: spend and unspend. For this reason, there is no space for other multi-stage scripts that keep other internal states. The Script language also does not provide fine-grained control over the amount that can be withdrawn. Finally, the Script language does not have access to certain block data such as the nonce and previous block hash.

With all these limitations in Bitcoin scripting language, there was an open space for innovation, which has lead to the creation of Ethereum.

2.2 Ethereum

Ethereum [4] is a blockchain similar to Bitcoin. The main difference is that it supports smart contracts written in a Turing-complete programming language. *Smart contracts* are programmable and self-executing programs that automatically enforce properties of a digital contract.

Ethereum introduces the concept of accounts. There are two types: externally owned accounts (controlled by cryptographic keys, like those used in Bitcoin) and contract accounts (controlled by a contract itself). An externally owned account can send messages by creating and signing a transaction. On the other hand, when a contract account receives a message, it executes the smart contract code, and can perform read and write operations to internal storage, send other messages, and create more contracts.

A smart contract can be seen as an autonomous agent that executes a specific code when it receives a message, or transaction, and has direct control over their balance, keys and value store.

A transaction contains the recipient, the signature identifying the sender, and the amount of *ether* (the Ethereum digital coin). In comparison with Bitcoin, there was a need to add extra fields, such as: a data field, containing values that can be used as inputs for the smart contract; STARTGAS, representing the maximum number of computational steps that the transaction is allowed to take, and; GASPRICE, the fee that the sender pays for each computational step.

The STARTGAS and GASPRICE are crucial components to prevent accidental or hostile infinite loops or other computational wastage, therefore protecting the Ethereum network against denial-of-service attacks. Moreover, these parameters force attackers to pay for the resources

they consume.

Contracts have the possibility to interact with other contracts through messages. A message is similar to a transaction, except it is produced by a contract account.

The code in the contracts is written in a low-level, stack-based bytecode language, named *Ethereum virtual machine code* (EVM code). All operations can store data in the stack, memory or long-term storage. The EVM code can access all fields of the incoming message, but it can also read the block header and output a byte array of data. The contracts can also be written in a high-level language, such as *Solidity*, that is compiled down to EVM code.

The Ethereum state transition function (STF) calculates if the sender transaction specifies enough amount of ether to execute the smart contract. The amount needed is then sent to the contract account and the smart contract is executed. The amount of unused ether is sent back to the sender's account, and the resulting state is returned.

While Ethereum distributed ledger are similar to the Bitcoin, they differing in the sense that blocks contain recent states besides transactions. A contract code is executed when the STF is used to validate a transaction, therefore the execution of contract code makes part of the block validation algorithm. For instance, if a transaction is added into block A, the contract code that references that transaction will be executed by all nodes in the network that validate block A.

Ethereum also differs from Bitcoin by implementing the GHOST (Greedy Heaviest Observed Subtree) protocol, introduced by Sompolinsky and Zohar [22]. GHOST includes orphan blocks (known as *uncles* in Ethereum) in the chain with highest cumulative difficulty. Ethereum [4] incentivises miners to add uncle blocks, by rewarding them with 87.5% of its base reward, and the nephew block (child of the uncle block) receives the remaining 12.5%. Transaction fees are not awarded to uncles.

Smart contracts can be seen as a step forward for better automation in a range of industries, such as in financial system, suitable for savings wallets, hedging contracts or world-scale employment contracts, plus applications such as online voting, decentralised governance or health care.

2.2.1 Casper the Friendly Finality Gadget

Casper the Friendly Finality Gadget (Casper FFG) [23] is an implementation proposal for Ethereum, consisting in a partial consensus mechanism which combines *Proof-of-Stake* (PoS) and Byzantine fault tolerant consensus.

PoS allows ether owners to become a *validator*. A validator will receive a block from the proposal mechanism, perform its validation and then broadcast a *vote* to add it on the blockchain.

The validator funds are then locked into a *deposit* by a smart contract. Validators take turns proposing and voting on next blocks. The weight of each validator depends on the size of his deposit. If the block is accepted, the validator gets a reward. If not, the validator's deposit is lost.

Casper FFG provides a safety mechanism through PoS and Byzantine fault tolerance consensus to achieve strong finality guarantees, preventing the occurrence of forks within the blockchain. However, the liveness property of the consensus depends on the chosen proposal mechanism, that at the time of writing, it is by using PoW.

Shortly, Casper FFG introduces novel features that BFT protocols do not support, such as:

- *Accountability*: All actions made by a validator are recorded on the blockchain. As rules are violated, it is possible to know *who* and *how* it was done, enabling penalisation to offenders.
- *Dynamic validators*: Validators can change over time.
- *Modular overlay*: Designed in a modular approach, allowing existing PoW blockchain to be upgraded.

In future work, the proposal mechanism will change entirely to PoS, with the ambition to end with the energy consumption of PoW.

2.3 Comparison between Bitcoin and Ethereum

Bitcoin (presented in Section 2.1) is a currency system, Ethereum (presented in Section 2.2) on the other hand, focuses on building general applications using smart contracts. Both are built on top of PoW consensus, although Ethereum is starting to use PoS to achieve consensus finality, improving safety.

Bitcoin follows a transaction-based data model. Ethereum follows a more general account-based data model. Table 2.1 shows a high level comparison of the two main permissionless blockchains.

2.4 Simulators

Network and distributed system simulators are important tools to evaluate the performance of protocols and systems in a large set of conditions. One common approach to perform these evaluations is to use machines at universities or use a global research network such as PlanetLab

Table 2.1: Comparison between Bitcoin and Ethereum

	Bitcoin	Ethereum
Node identity	Open	Open
Application	Cryptocurrency	Smart contracts
Consensus	PoW	PoW/PoS (hybrid)
Data model	Transaction-based	Account-based
Smart contract execution	Native	EVM
Smart contract language	Script	Solidity, LLL

[14]. 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.

2.4.1 What is Simulation?

Simulation can be defined as an "imitation of the operation of a real-world process or system over time. Whether done by hand or on a computer, simulation involves the generation of an artificial history of a system and the observation of that artificial history to draw inferences concerning the operating characteristics of the real system." [24, 25]

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. These assumptions can be expressed in algorithmic (a sequence of steps), mathematical and logical relationships between entities of the system.

Simulation can be used to study a wide variety of questions about the real-world system. It can also be 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.

Computer simulation is resorted to when systems are too complex to be emulated. A computer simulation consists of the actual execution of the program which manifests the model or *simulation model*. This simulation models can be classified according to several independent pairs of attributes [24], stochastic or deterministic, static or dynamic and discrete-event or continuous.

A *stochastic* simulation model receives random input values, leading to random outputs. Therefore, they can only be considered as estimates of the true characteristics of a model. On the other hand, a *deterministic simulation model* does not contain random values.

A *static* simulation model, represents a system at a particular point in time. A *dynamic* simulation model represents a system as it changes over a certain time frame.

In a *discrete-event* simulation model [25] a system is described 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, the *continuous* simulation model tracks the system states over time.

Therefore, discrete-event simulations do not need to go by every time slice, they can typically run much faster than the continuous simulations.

2.4.2 Why Simulation?

There are numerous reasons to adopt simulation, as stated by Banks [25]. Among some:

- New designs or changes to systems can be tested without the necessity to allocate resources to a final system.
- Help determine why certain phenomenon occurs, through the reconstruction of elapsed events.
- Explore new policies, decision rules, operating procedures and information flows, without interrupting ongoing operations of the real system.
- Obtain insights and diagnose problems regarding interactions of variables and their impact to the performance of the overall system.
- Perform bottleneck analysis to discover where certain processes are being excessively delayed.
- Understand how systems operate, instead of preconceptions of these ones.
- Forecasting the impact of new systems - i.e., answering to "what if" questions.

The following sections explore some advantages of the use of simulation for blockchain.

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.

Due to time pressure, many blockchain implementations or even consensus protocols from the research community are not available at publication time. By recurring to simulation, the performance of such systems can be explored, without requiring a complete implementation.

2.4.3 Differences between Simulation and Emulation

Simulation and emulation are two of the most commonly ways of evaluating a system, in this section we will discuss emulation compared with simulation.

Emulation can be defined as "the process of implementing the interface and functionality of one system or subsystem on a system or subsystem having a different interface and functionality" [26]. For this reason, emulators incur on a large overhead to ensure the emulated interface and functionality runs in real time while providing the virtualization layers needed to emulate an entire system.

As a result, emulation is more *accurate* than simulation, but less *scalable*. Typically, emulators runs hundreds of nodes while simulators run thousands [27].

2.4.4 Simulators

Simulators like The ONE [15], PeerSim [16], CloudSim [17] and BFTSim [28] 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 [29], Shadow-Bitcoin [30] and a recent work by Stoykov and Zhang, who proposes VIBES (Visualisations of Interactive, Blockchain, Extended Simulations) [31]: a blockchain simulator capable of handling large-scale network simulations. BLOCKBENCH [13] is a framework that performs a series of benchmarking analysis, to stress test private blockchains already deployed.

We start by presenting briefly a few widely adopted simulators: The ONE, PeerSim, and CloudSim, then we present in more detail a set of simulators that are closer to our work: BFTSim, Bitcoin Simulator, Shadow, and finalize with an explanation about the BLOCKBENCH framework.

The ONE (Opportunistic Networking Environment) simulator was designed for evaluating the behaviour of protocols and routing strategies on DTNs (Delay-Tolerant Networks). These are networks in which end-to-end connectivity between a source and target node may never exist. The ONE offers an extensible simulation framework that support event generation, message exchange, notion of energy consumption, virtualization, an interface for importing and exporting mobility traces, events and messages.

PeerSim is a scalable and modular simulator of P2P systems. A P2P system splits the tasks between peers, and the peers are equally privileged participants in the system. Peers are both suppliers and consumers of resources and thus can share a portion of their own resources,

such as disk storage, network bandwidth or processing power directly with other participants, without the need for a central authority. These systems are expensive and difficult to reproduce and evaluate, due to their scale and dynamism. PeerSim attempts to overcome these problems by supporting dynamic scenarios and failures models with ease of configuration. Following a modular approach, PeerSim models the *network* component as a list of *nodes*, each node consisting of a list of *protocols*. Each component is modified and monitored by the simulation *initialisers* and *controls* components. The simulation engine can execute the protocols in a specific order (cycle-based) or by triggering events (event-based).

CloudSim is an extensible simulation framework that allows simulation of emerging cloud computing infrastructures and application services. It supports modelling of system and behaviour components such as data centers, virtual machines and policies for resource provisioning. CloudSim enables performance analysis of an application in a controlled, single host, easy to set-up environment, requiring less effort and time to test cloud-based applications. It also supports simulation of network connections among simulated systems.

BFTSim

BFTSim [28] is a single-thread and single-host BFT state machine replication (SMR) protocol simulator, allowing the rapid development and evaluation of protocols.

BFTSim is composed by several components. The interface uses pseudocode in a high-level declarative language, enabling the implementation of a BFT protocol. The back end is based on the widely used *ns-2* discrete-event based simulator, that simulates the network conditions.

The pseudocode is compiled using a declarative networking language called *P2*, to a software data flow graph responsible to capture timing characteristics of intensive CPU functions without necessarily performing them. BFTSim runs on the cost of small number of key primitives, such as cryptographic and network operations. The cost of such key is used by the simulator to appropriately delay message handling, thus, allowing BFT protocols that use such key primitives to be accurately simulated.

Bitcoin Simulator

The Bitcoin Simulator [32] was created to study how consensus parameters, network characteristics and protocol modifications affects the scalability and security of PoW-based blockchains.

Built on top of the *ns-3* network simulator, the Bitcoin Simulator runs on real Bitcoin network statistics, such as: network delays, block generation time, block size, number of nodes and their geographic distribution, and mechanisms for information propagation. The simulation

then measures the block propagation times, throughput and stale block rate, as input to a security model, based on Markov Decision Processes (MDP). These measures enable the study of optimal adversarial strategies, by comparing security and performance of PoW-based blockchains when subject to different parameters.

Shadow

With the goal of increasing the consistency, accuracy and scalability of experiments on real world applications, Jansen and Hopper [27] designed and implemented a discrete-event simulator capable of running real world applications, called Shadow. The project's main goal was to simulate *Tor*, in order to test new design proposals or attacks on the system without compromising users privacy. This allows to run a private Tor network on a single machine with thousands of nodes, while controlling all aspects of an experiment. The results can be easily repeatable and verifiable through independent analysis.

Shadow uses various techniques to run real world application. It started by encapsulating the application in plugins wrappers containing functions that allows Shadow to interact with the application. The application is loaded in memory and all variable addresses are registered by the plugin. Shadow then manages a copy of the memory regions, swaps a version of this state before passing control to the application, and when control returns, Shadow swaps out the state. It also applies the technique of *function interposition*, to replace calls to functions in dynamic libraries, such as event, crypto or socket libraries, with calls to a simulated counterpart. Therefore, it is not necessary, for instance, to perform cipher operations during encryption and decryption, because such CPU operations are already modelled and their respective delays are known. Therefore, this highly intensive CPU operations can be skipped without affecting the application functionality. All these techniques are applied without the need of modifying the source code of the application.

Shadow also simulates the network layer, by instantiating *virtual nodes* that represent a single simulated host. The virtual nodes communicate with each other through a *virtual network* which transfer packets and related events to the *Shadow scheduler* that delivers the events to another node after applying network latency.

They conducted successfully experimental analysis [27], verified the accuracy of running a Tor simulation, and compared the results with live statistics of the public Tor network.

Shadow was also used to simulate a Bitcoin network, in a research called *Shadow-Bitcoin* [30]. The authors developed a new methodology, allowing virtual hosts in Shadow to run multi-thread applications, as well as developing a new plugin to run the Bitcoin reference implementation (also

known as *bitcoind*). A Bitcoin model was used to bootstrap and instantiate a large Bitcoin test network, capable of running 6000 nodes inside of Shadow on a single machine. The performed experiments present a novel denial-of-service attack against the low-level implementation of Bitcoin client.

VIBES

VIBES [31] performs large-scale simulation of blockchain systems in efficient time, by receiving input parameters from the user. The input parameters include network characteristics (topology, latency, bandwidth, number of nodes) and blockchain system characteristics (number of miners, block size, block confirmation time, number of transactions per block, percentage of attacker nodes, percentage of failing nodes). To simulate a blockchain with thousands of nodes, they receive the previous input parameters coupled with empirical and theoretical results. With these informations, the simulator does not need to perform heavy computations and it is able to simulate ahead of time.

The mechanism works as follow: the simulations nodes calculate how much an operation would take and ask the orchestrator for permission to fast-forward. The *orchestrator* issues a timestamp *ts* for this operation, notifies and fast-forwards the entire network to time *ts*. When nodes finish their work, the *reducer* receives a global state from the orchestrator, including timestamped transactions. The reducer applies a stateless function and outputs the result to the user. These outputs can include metrics, such as: "total time to process, total number of transactions processed, throughput (transactions per second), block propagation delay for 10%, 50% and 90%, client bootstrap time, cost per transaction, probability of an attacker taking over at each stage, and a log of all transactions" [31].

BLOCKBENCH

BLOCKBENCH [13] is a framework that performs a series of benchmarking analysis through workloads, commonly used to benchmark databases, to stress test private blockchains, such as Ethereum, Parity and Hyperledger Fabric. These private blockchains can be emulated, or used nodes can be instantiated in the network. Therefore, this framework *cannot be considered a simulator*, instead it follows an emulation approach. Yet, we believe it has value to be explored.

The blockchain architecture is split into four modular layers: consensus, data model, execution and application. The *consensus layer* gathers all nodes in the network to agree on the ledger. The *data model layer* is where and how information is stored. The *Execution layer* executes the smart contracts. Finally, the *application layer* represents the application that a smart

Table 2.2: High level comparison between simulators.

	Application	Specification	Stochastic	Deterministic	Static	Dynamic	Discrete-event	Continuous	Model of network latency	Model of network bandwidth	Model of ad-hoc networks	Model of CPU
The ONE [15]	Opportunistic networks	programmatic	✓			✓	✓				✓	
PeerSim [16]	Peer-to-Peer systems	abstract	✓			✓	✓	✓	✓			
CloudSim [17]	Cloud computing systems	programmatic	✓			✓	✓		✓	✓		
BFTSim [28]	BFT SMR protocols	programmatic	✓			✓	✓		✓	✓		✓
Bitcoin Simulator [29]	Bitcoin network	abstract		✓		✓	✓		✓	✓		
Shadow [27]	General applications	programmatic		✓		✓	✓		✓	✓		✓
VIBES [31]	Blockchain systems	abstract		✓		✓	✓		✓	✓		

contract is designed for. For each one of these layers, there is at least one micro-benchmark workload to measure its performance in isolation to other layers. This allows the assessment, for instance, of smart contracts, input/output and computation speed by performing stress tests using the workloads.

New workloads can be created and added to the framework by using a simple application programming interface.

2.4.5 Discussion

In the previous section we have explored a number of simulators. A high level comparison is made in Table 2.2, exploring their simulation model attributes, respective, as discussed in Section 2.4.1.

In Table 2.2, one can observe that all simulators follow a discrete-event simulation (DES) model, as systems events are simulated at a discrete point in time. By skipping the simulation time to the next event we can obtain the same results in less time, instead of keeping track of changes continuously over time. The DES models are dynamic, "i.e., the passage of time plays a crucial role. Most mathematical and statistical models are static in that they represent a system at a fixed point in time." [25].

There are simulators that uses stochastic models such as: The ONE, which uses models for random node movements; PeerSim, that creates random network topologies; CloudSim, where systems components can join, fail, or leave the system randomly; and finally, BFTSim, that uses random packet loss.

All the simulators studied in the previous Section 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. The specification of these parameters can be made either using programmatic interfaces or using abstract ways such as text files.

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.

Chapter 3

BlockSim

BlockSim 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.

A simulation model can be classified by certain attributes, as mentioned in Section 2.4. This simulator follows a stochastic simulation model, being able to represent random phenomena by introducing probability distributions for certain events. Our solution consists in the introduction of random phenomena in terms of probabilities of events, outlined in Section 3.1.

Our models are considered dynamic; they can represent the system over a certain interval determined by the user interacting with BlockSim.

A discrete-event simulation (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 (as in continuous simulation model). Therefore, the simulator can keep track of thousands of nodes and events that only change states.

BlockSim code repository is available[1], under MIT license.

3.1 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 intend always to mimic the behaviour of the entities. For in-

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

stance, 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 collect the iPerf3 logs from the two instances. For the current available locations in BlockSim (Ireland, Ohio and Tokyo) our repository store all the measurements collected[2].

4. We 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 (*cf.* Section 3.2.2) in order 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.

3.2 Architecture

BlockSim follow a single process architecture, represented in Figure 3.1, where we illustrate the main components, connectors and interfaces of the implementation. The following sections present the functionality of each one.

[2]Available at: `https://github.com/BlockbirdLabs/blocksim/tree/thesis/raw-measurements/iperf3`

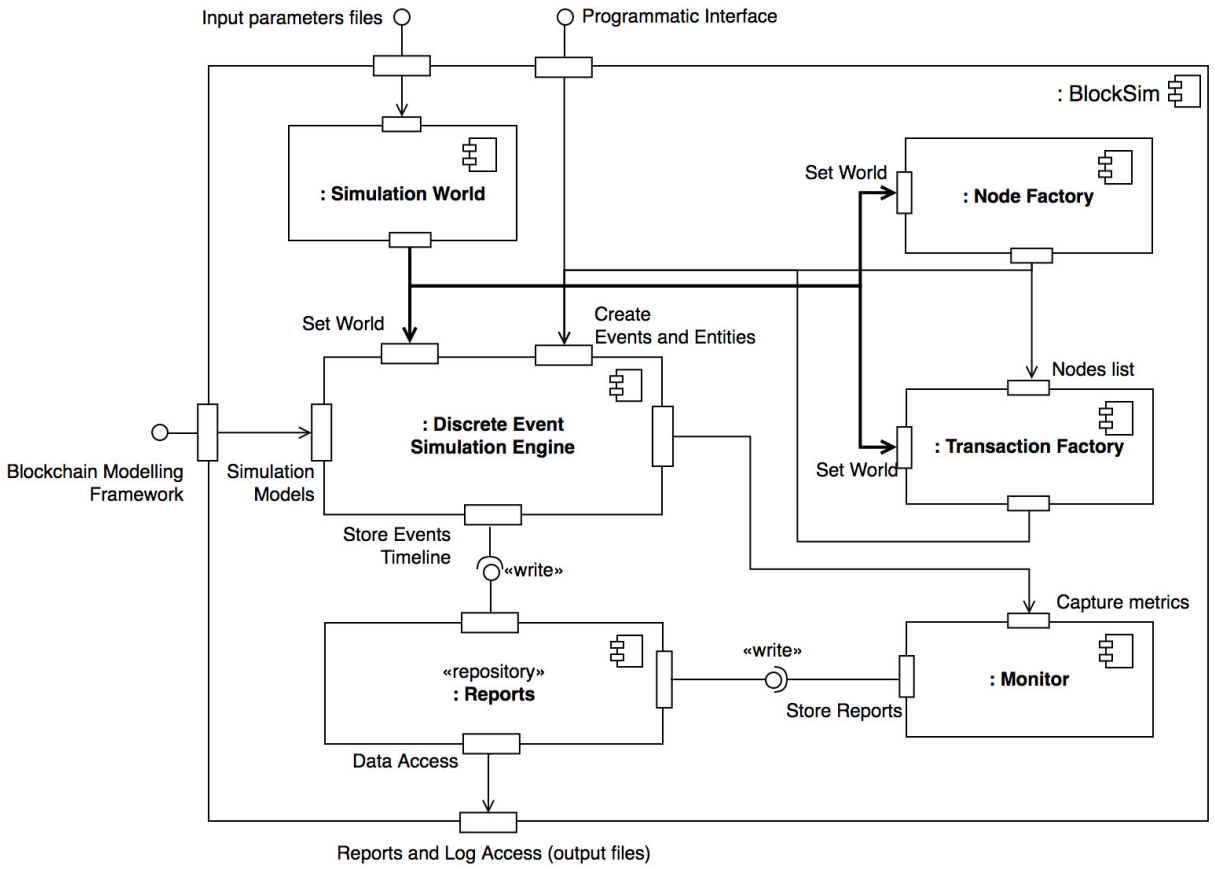


Figure 3.1: Architecture of BlockSim showing the main components, connectors and interfaces.

3.2.1 Discrete Event Simulation Engine

We use *SimPy* [33] 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.

All processes live in an environment. They interact with the environment and with each other via events. The events are generated and scheduled at a given simulation time. Events are sorted by simulation time and priority. An event can also execute predefined functionality when the event is triggered and processed by the event loop.

Additionally, simulations in SimPy can run as fast as possible, in real time or by manually stepping through the events. SimPy also provides numerous types of shared resources to model limited capacity congestion points, such as: servers, connection channels or queues.

In short, our DESE using SimPy supports several core functionalities [24], 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.

3.2.2 Simulation World

The *Simulation World* component (see Figure 3.1) is responsible to manage the inputs of the simulator, which mainly are the probability distributions mentioned in Section 3.1. 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 3.3). These input parameters consist on the following files:

- *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.
- *Delays file*: contains the probability distributions corresponding to the time to validate a transaction, block and time between blocks, for each blockchain.
- *Latency file*: contains the probability distributions corresponding to the latency between possible locations for nodes.
- *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 (as showing in Figure 3.1), making available all the attributes (configurations and probability distributions) which characterise the world of the simulation.

3.2.3 Transaction and Node Factory

The *transaction factory* is responsible for creating batches of random transactions. Depending on the blockchain being simulated, the transaction factory will create transactions according to the transaction model. Moreover, the created transactions will be broadcasted when simulation

is running by a random node on a list, or by a chosen node. Additionally, the user needs to specify the number of batches, number of transactions per batch and the interval in seconds between each batch.

The *node factory* creates and initiates nodes that are used during the simulation. Depending on the blockchain being simulated, the node factory will create nodes according to the node model. The user can specify the location, number of miners and non-miners (or full-nodes *cf.* Section 2.1), and the range of hash rate for the miner nodes. When nodes are created, is chosen a random hash rate from the range inputed. The location of each node needs to be recognised by the simulator, meaning that it needs to exist input parameters about latency and throughput.

3.2.4 Programmatic Interface

The programmatic interface is the main interface available to the user. Using *Python* language and SimPy [33], 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-parameters/config.json',
    'input-parameters/latency.json',
    'input-parameters/throughput-received.json',
    'input-parameters/throughput-sent.json',
    'input-parameters/delays.json')

# Create the network
network = Network(world.env, 'NetworkXPTO')

miners = {
    'Ohio': {
        'how_many': 0,
        'mega_hashrate_range': "(20, 40)"
    },
```

```
21          'Ireland': {
                'how_many': 2,
23              'mega_hashrate_range': "(20, 40)"
            }
25      }
        non_miners = {
27          'Tokyo': {
                'how_many': 3
29          },
            'Ireland': {
31              'how_many': 2
            }
33      }

35      node_factory = NodeFactory(world, network)
        # Create all nodes
37      nodes_list = node_factory.create_nodes(miners, non_miners)
        # Start the network heartbeat
39      world.env.process(network.start_heartbeat())
        # Full Connect all nodes
41      for node in nodes_list:
            node.connect(nodes_list)
43
        transaction_factory = TransactionFactory(world)
45      # Broadcast a batch of 6 transactions every 5 mins, 7 times
        transaction_factory.broadcast(7, 6, 300, nodes_list)
47
        world.start_simulation()
```

Listing 3.1: A demonstration of how to define the simulation using different models.

Listing 3.1 demonstrates how the user can instantiate the simulation, starting by creating the Simulation World (*cf.* Section 3.2.2). The Simulation World is then instantiated with the simulation duration in seconds, timestamped when the simulation starts (in this example, we set the current time) and finally the file path to each input parameter. After creating the network, the user uses Node Factory (*cf.* Section 3.2.3) to create the nodes for the simulation. The user then starts the network heartbeat (*cf.* Section 3.3) in line 39 and connects all nodes with each other. Using the Transaction Factory (*cf.* Section 3.2.3), 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 in line 48 gives the order to DESE start the simulation.

3.2.5 Monitor

BlockSim has a logging system that captures the time and origin node in which certain events occur. This helps during debugging to see each step in the simulation, what nodes are sending or receiving, and many other events. However, logs are not particularly useful when the simulation does work correctly, logged events are not statistically relevant, and tracing certain events in log files can quickly become unwieldy.

The goal of the *monitor* is to capture arbitrary metrics during the simulation at any part of the models. It should be easy for the user to update metrics wherever needed, and have them automatically collected and stored.

We use this functionality to capture the number of transactions each node broadcasts or receives, transactions added to the queue, blocks processed, and time to propagate transactions or blocks. This component is used to perform measurements in order to evaluate the simulator in Section 4.

3.2.6 Reports

As mentioned in the previous section, BlockSim logging system is essential to have a perception when certain events occur during the simulation. However, these logs need to be stored and made available at the end of simulation.

Reports component is responsible to write logs to a log file at the end of the simulation. Additionally, it also stores the metrics from the monitor in JSON files (*cf.* Section 3.2.5).

3.3 Blockchain Modelling Framework

In order 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.

The developed framework is inspired on the concepts and design decisions of CloudSim [17] and The ONE [15]. The previously defined layers can be expressed in models that are used to create classes, toward extending and implementing modelling needs.

Models such as: Node, Transaction, Block, Consensus, and Network are available as classes that can be extended by the user. These classes are then used by the DESE to create blockchain system entities, which interact within events defined in the models. For instance, a Node class can define an event to broadcast transactions to other nodes in the simulation.

The class diagram in Figure 3.2 represents the basic classes available in the framework. These basic models can be extended to simulate specific blockchain implementations (such as Bitcoin and Ethereum, which we will discuss in the next section).

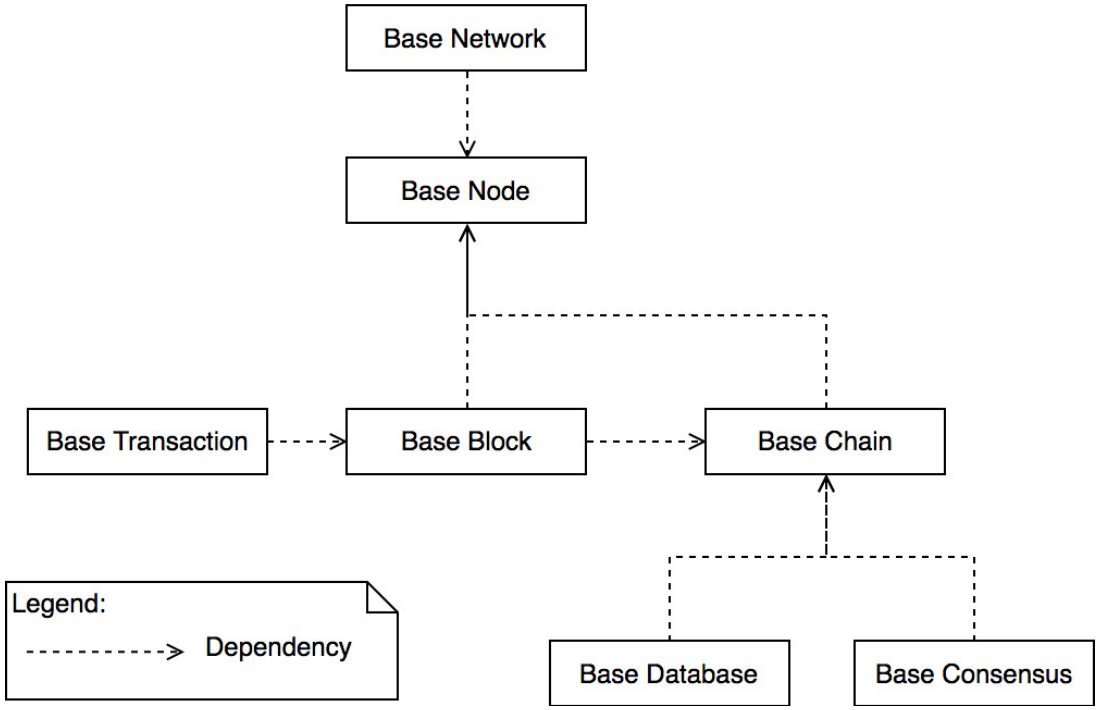


Figure 3.2: Class diagram of the modelling framework.

We will explore, in the following sections, the basic models on the framework that will be an important resource to model different blockchain systems.

Chain Model

Responsible to mimic the behaviour of a chain. In this model, we implement an abstract functionality that works across different blockchains, as described in Section 2.1.2.

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, 4]. If this is the case, it accepts the secondary chain as the main chain.

Database Model

Defines a data structure, a simple key-value store, responsible to define an interface to how a node can access blocks on this chain. Each node stores is own chain in this data structure.

Consensus Model

Responsible to introduce 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:

$$difficulty = P_d + (B_{TS} - P_{TS}) \tag{3.1}$$

Equation 3.1 simplifies and resembles the ideals of Ethereum [4] 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.

Block Model

A block model defines the structure of a block, divided in its header and associated list of transactions. The header of a block contains the previous block hash, number, timestamp, *coinbase address* (miner address), difficulty and nonce. We do not use merkle trees to store transactions in a block, because we are not interested in performing exhaustive validations in this simulation.

Transaction Model

The transaction model defines the structure of transactions, containing the destination and sender address, value, signature, and fee. We do not sign transactions because its not a task for a node in a blockchain network.

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 (*cf.* Section 3.2.2).

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 input as parameters (*cf.* Section 3.2.2), 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 (*cf.* Section 2.1.3). The network model simulates this behaviour by selecting two nodes to broadcast its candidate blocks. This event only occurs with a predefined probability [20, 21], set in configuration (*cf.* Section 3.2.2).

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 intends to send or receive the message. These delays are obtained by probability distributions, as mention in Section 3.1.

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 (*cf.* Section 3.3).

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.

3.4 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 3.3.

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 [34]. 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.

The following sections will present the models for the *Bitcoin Network Messages* and the *Bitcoin Node*.

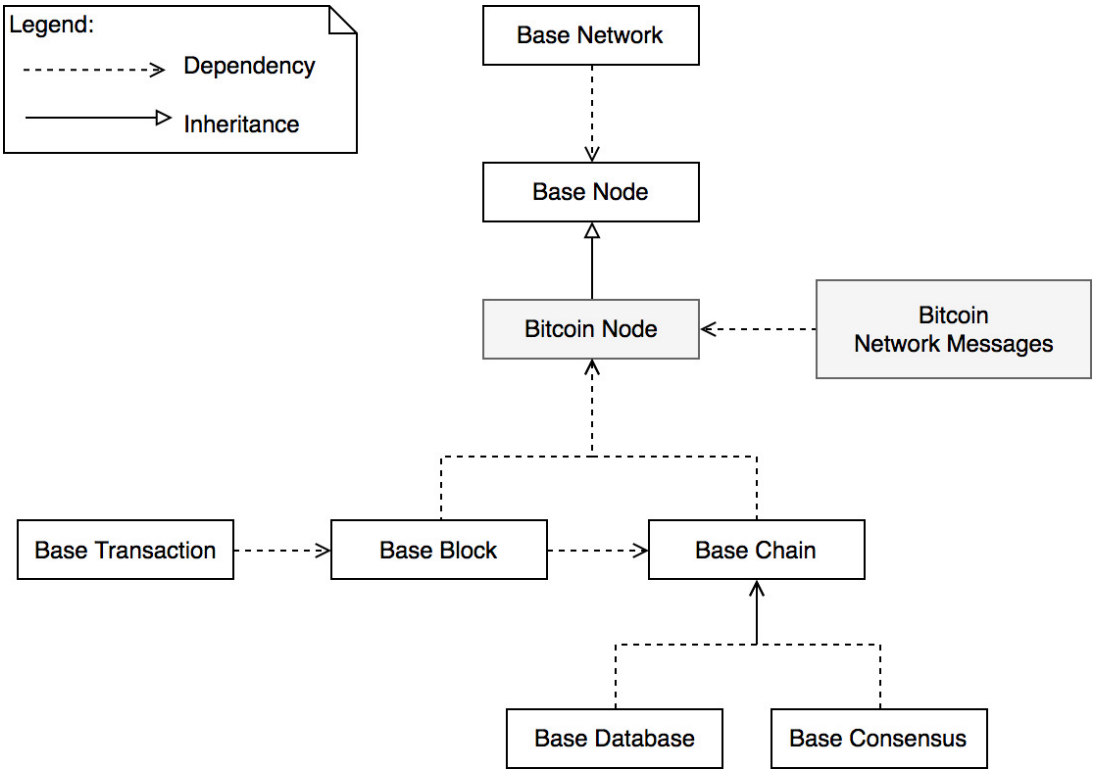


Figure 3.3: Class diagram for the Bitcoin modelling.

3.4.1 Bitcoin Network Messages Model

The Bitcoin network protocol [35] 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 3.2.2) to calculate the size of each message. The sizes are taken from the documentation [35].

3.4.2 Bitcoin Node model

The Bitcoin Node model inherits the base node model (*cf.* Section 3.3), as shown in Figure 3.3. 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 are 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 (*cf.* Section 3.3); 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 3.1).

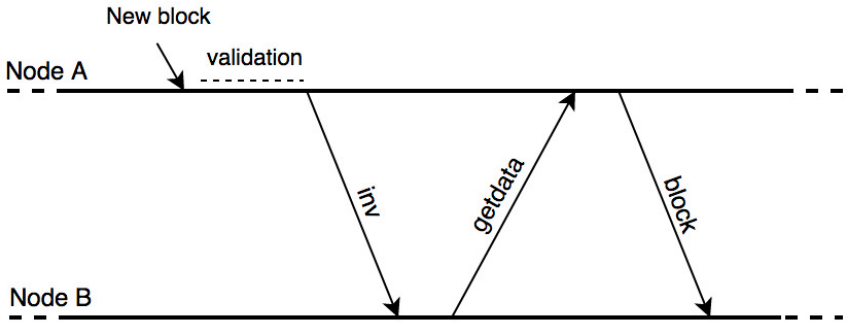


Figure 3.4: Messages exchange in Bitcoin protocol between nodes in order to obtain a new block.

The process of announcing a new block is illustrated in Figure 3.4, 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.

3.5 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 3.5.

Additionally, in the Simulation World component, we input the block gas limit and the start gas for every transaction (*cf.* Section 2.2). 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.

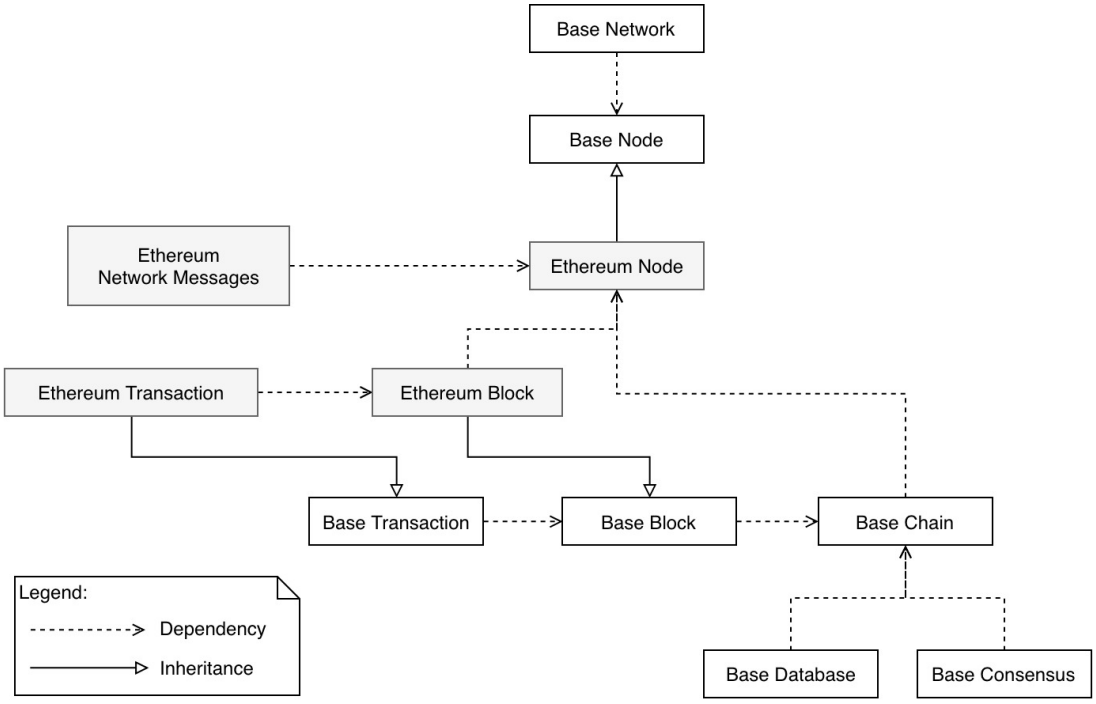


Figure 3.5: Class diagram for the Ethereum modelling.

The following sections will present the models for the *Ethereum Network Messages* and the *Ethereum Node*.

3.5.1 Ethereum Network Messages Model

The Ethereum Network protocol (PV62) [36] 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 (*cf.* Section 3.2.2) to calculate the size of each message. The sizes are taken from the documentation [36].

3.5.2 Ethereum Node model

The Ethereum Node model inherits the base node model (*cf.* Section 3.3), as shown in Figure 3.5. 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 (as the same as Bitcoin nodes) can be divided into two groups: a miner node or a non-miner node (or full-node).

A non-miner 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 into a block. The node only broadcasts its candidate block to the network when selected by the Network base model (*cf.* Section 3.3). This process simulates the mining of a new block.

We do not perform any type 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 3.1).

The process of announcing a new block is illustrated in Figure 3.6 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

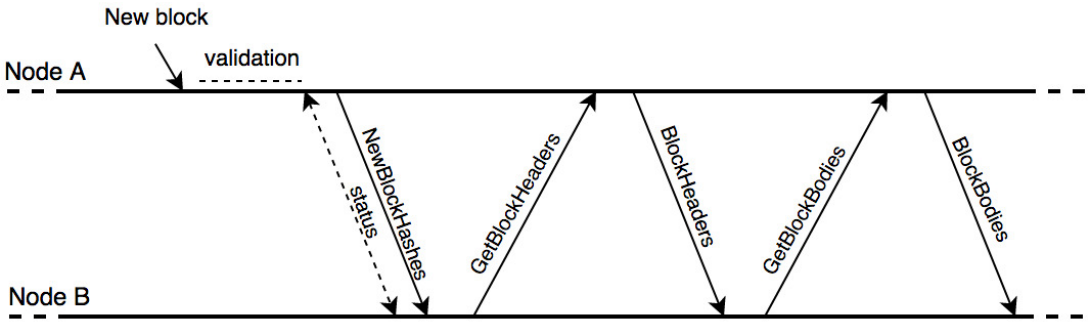


Figure 3.6: Messages exchange in Ethereum protocol between nodes in order to obtain a new block.

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 (*cf.* Section 3.3) 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 (*cf.* Section 3.3) by only adding new attributes, such as the gas limit and gas used.

3.6 Summary

This chapter presented the BlockSim implementation.

In Section 3.1, we created a methodology to measure, collect, and extrapolate a probability distribution in order to predict the next outcome, helping us and the user to build models that represent a real system with a degree of confidence.

The core components of BlockSim, explored in Section 3.2, are intrinsic parts of our solution that allow us easily to create new nodes, transactions, and to run and monitor the simulation.

The Blockchain Modelling Framework, presented in Section 3.3, offers to the user a mechanism rapidly to implement new models by extending the already existing models in order to promote reusability. We finish this chapter by showing how we use our modelling framework to model two existing blockchains, Bitcoin and Ethereum.

Chapter 4

Evaluation

This chapter presents the evaluation of BlockSim. Recall that the objective is to provide an accurate representation of a real blockchain system. Therefore, Section 4.1 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 4.2 explores and evaluates real use cases for BlockSim.

4.1 Verification and Validation

In order to evaluate BlockSim, it is necessary to perform a *verification* and *validation* of the simulation models as stated by Banks [25]. This require the replication of simulation models in a real private Ethereum network.

4.1.1 Simulation Study

We use BlockSim to perform a simulation study of the Ethereum reference implementation (as studied in Section 2.2), by using the models presented in Section 3.5. The steps to create a simulation study of Ethereum are:

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 3.1 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 (*cf.* Section 3.3) to create the specific Ethereum models for our study (*cf.* Section 3.5).

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.

By following this methodology we are verifying and validating BlockSim along with our Ethereum models.

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 (*cf.* Section 3.2.5), 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[1] presented in Table 4.1, 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.

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

[1]Available at: `https://github.com/BlockbirdLabs/blocksim/tree/thesis/input-parameters`
[2]Available at: `https://github.com/carlosfaria94/go-ethereum`

Table 4.1: Input parameters for the probability distributions used in the Simulation Study

	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	$\alpha = 0.463$ $\beta = 0.461$
Latency between Ireland and Tokyo	Normal	105.42 ms	0.23 ms	-
Throughput between Ireland and Tokyo	Beta	-410160.67 Mbps	410197.05 Mbps	$\alpha = 272250.32$ $\beta = 3.69$

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.

4.1.2 Results

At the end of the simulation study, described in Section 4.1.1, 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.

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

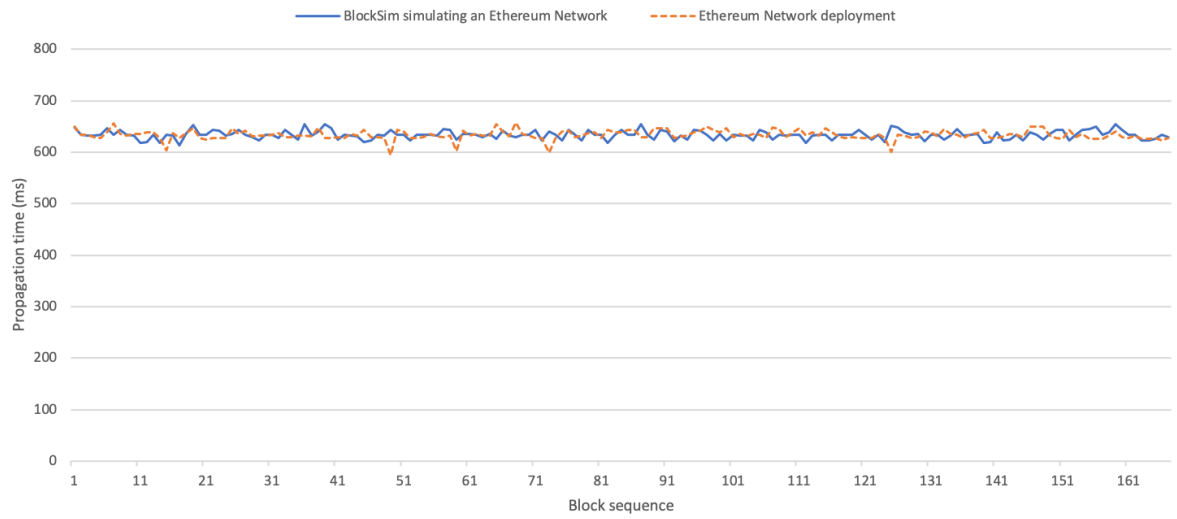
Block Propagation

The block propagation time starts when the origin node sends its 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 (i.e. do not contain any transaction)

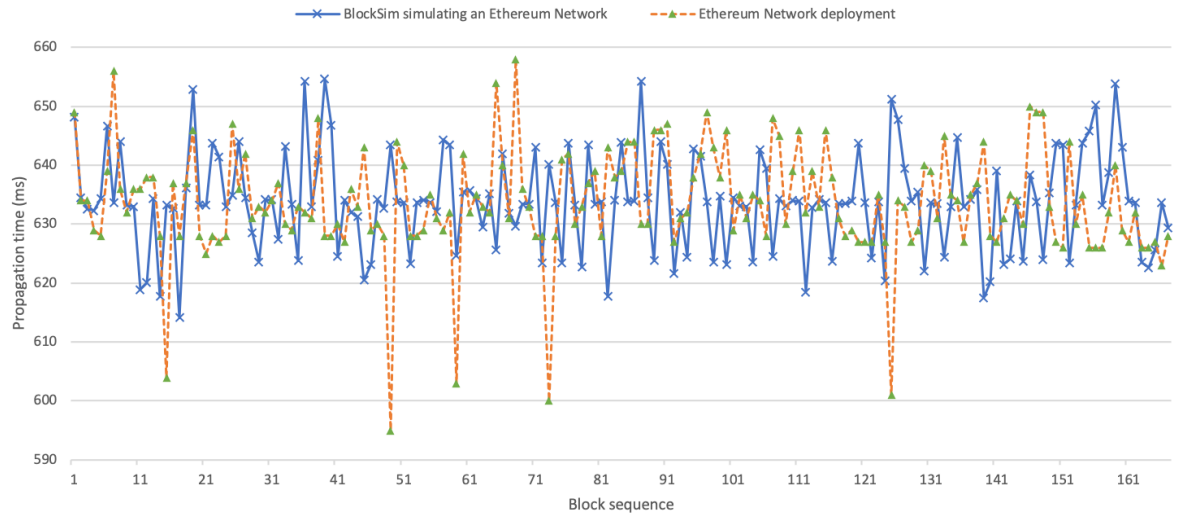
Figure 4.1 shows the time in milliseconds to propagate a block created by a miner in Ohio and a node receiving the block in Ireland. We achieved an average of 634 ms in the real Ethereum network with a standard deviation of 9.2 ms. The BlockSim simulating the Ethereum models achieved the exact same average of 634 ms, with a similar standard deviation of 8.28 ms.

Figure 4.2 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 4.2 shows that BlockSim runs our Ethereum models presenting slightly



(a) Overview.



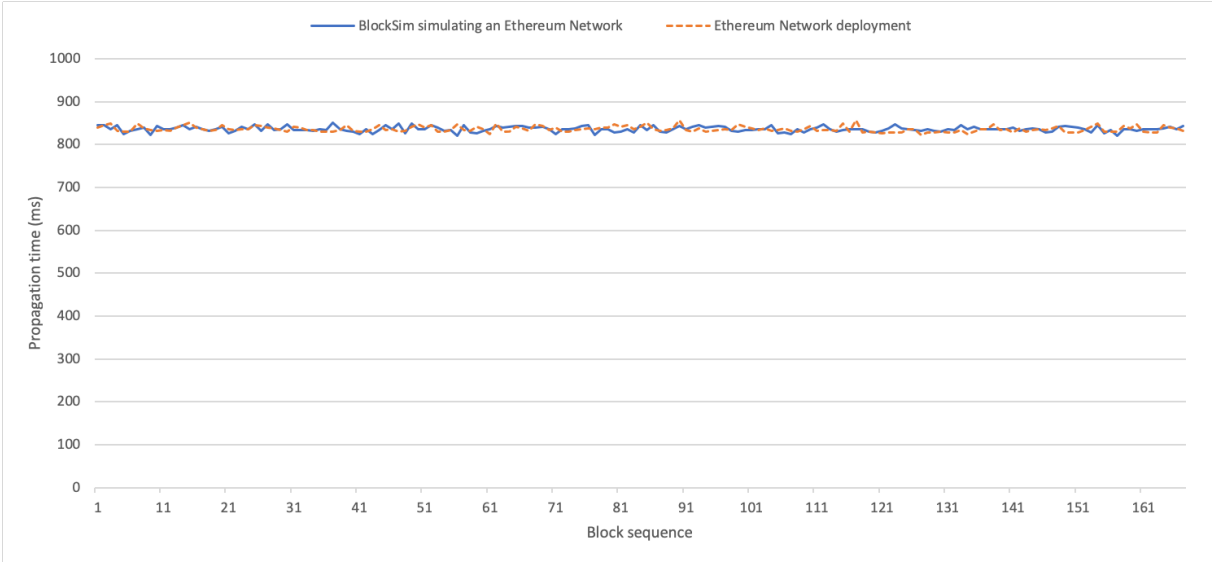
(b) Detailed view.

Figure 4.1: Block propagation between Ohio and Ireland.

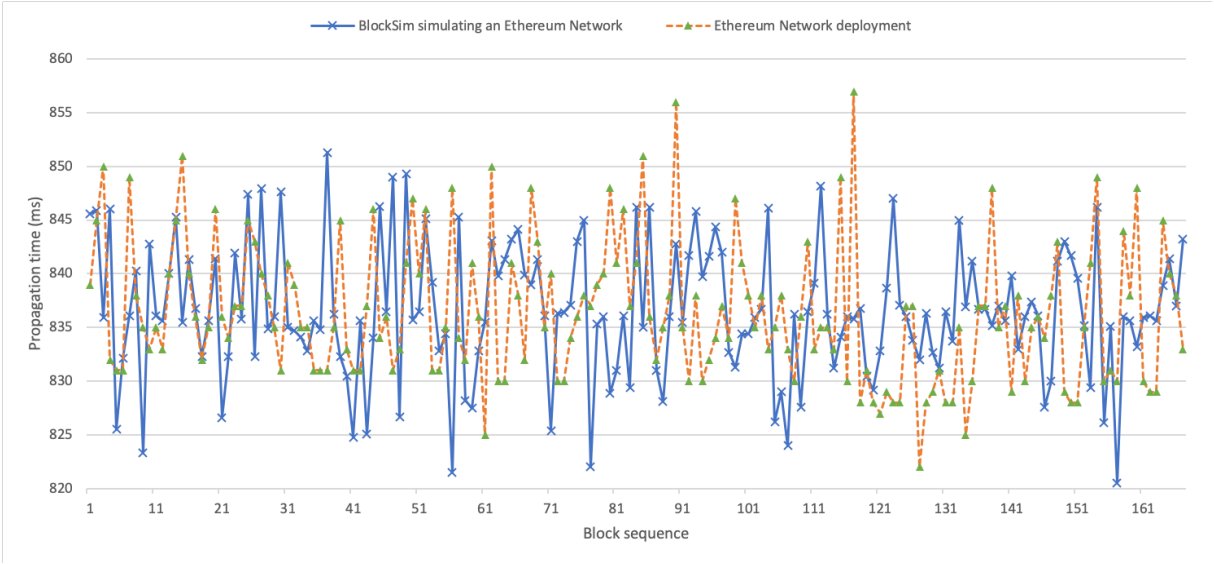
low values for standard deviation compared to a real network. These results were expected because our network model does not consider packet loss, routing and other variations that influence packets deliver in a wide area network (WAN).

Table 4.2: 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
Between Ohio and Ireland	634 ms	634 ms	9.2 ms	8.28 ms
Between Ireland and Tokyo	836 ms	836 ms	6.51 ms	6.17 ms



(a) Overview.



(b) Detailed view.

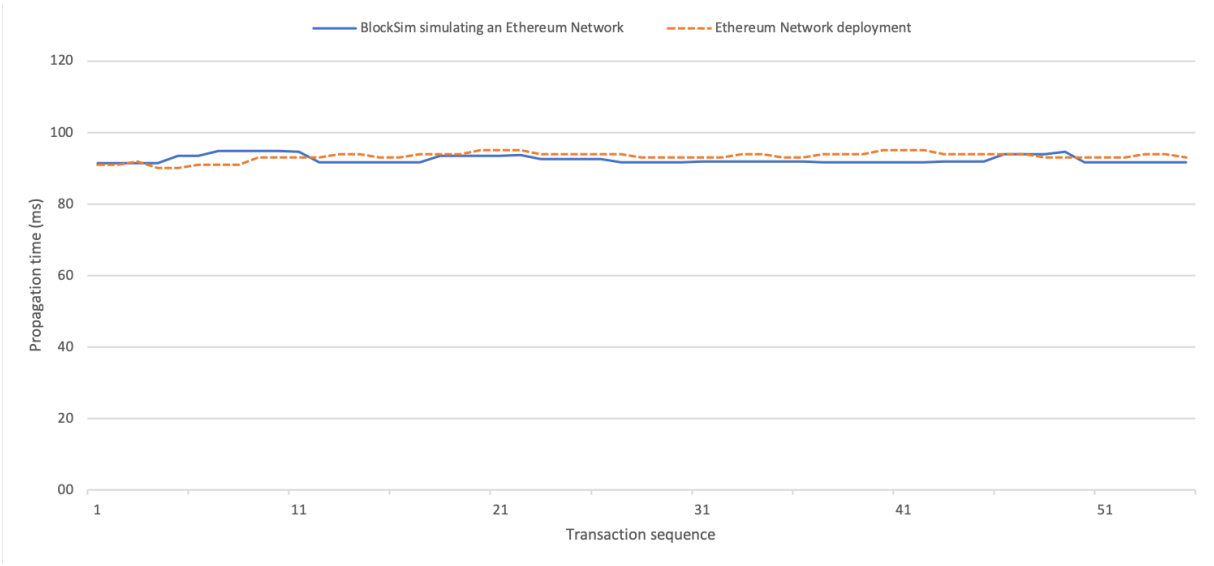
Figure 4.2: Block propagation between Ireland and Tokyo.

Transaction Propagation

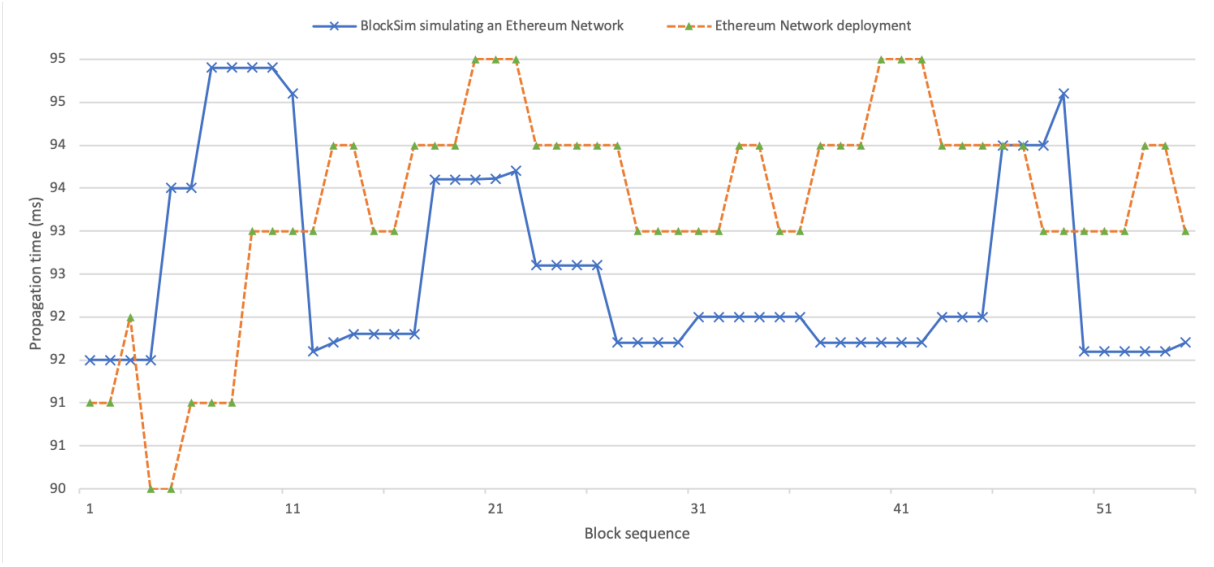
The transaction propagation time starts when the origin node sends a new transaction to a peer, and stop when the transaction is processed and validated by the peer.

Figure 4.3 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 BlockSim simulating the Ethereum model achieved the exact same average of 93 ms, with a similar standard deviation of 1.12 ms.

Figure 4.4 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



(a) Overview.

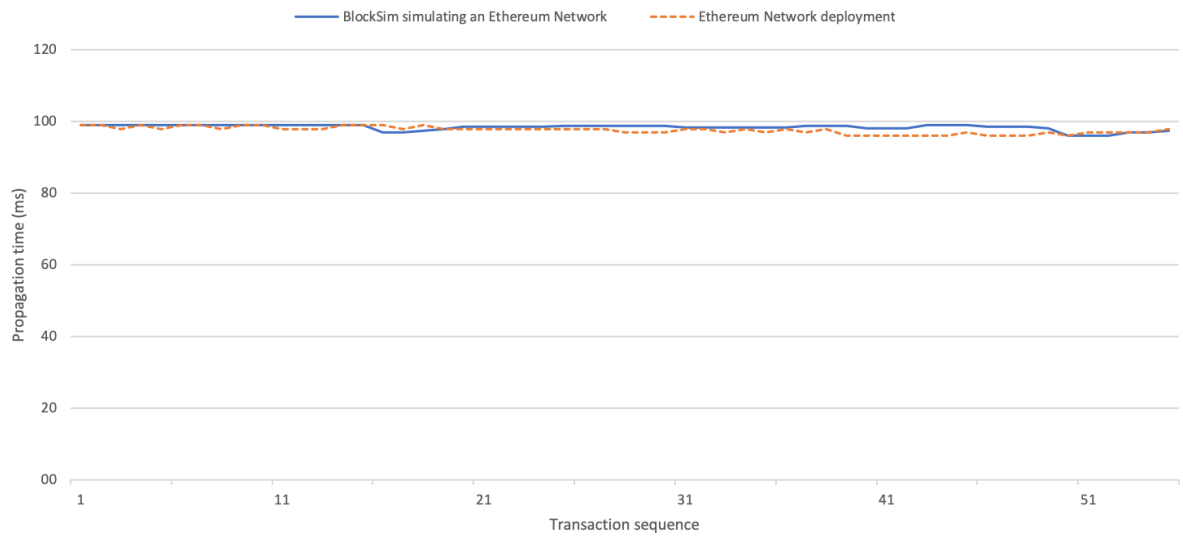


(b) Detailed view.

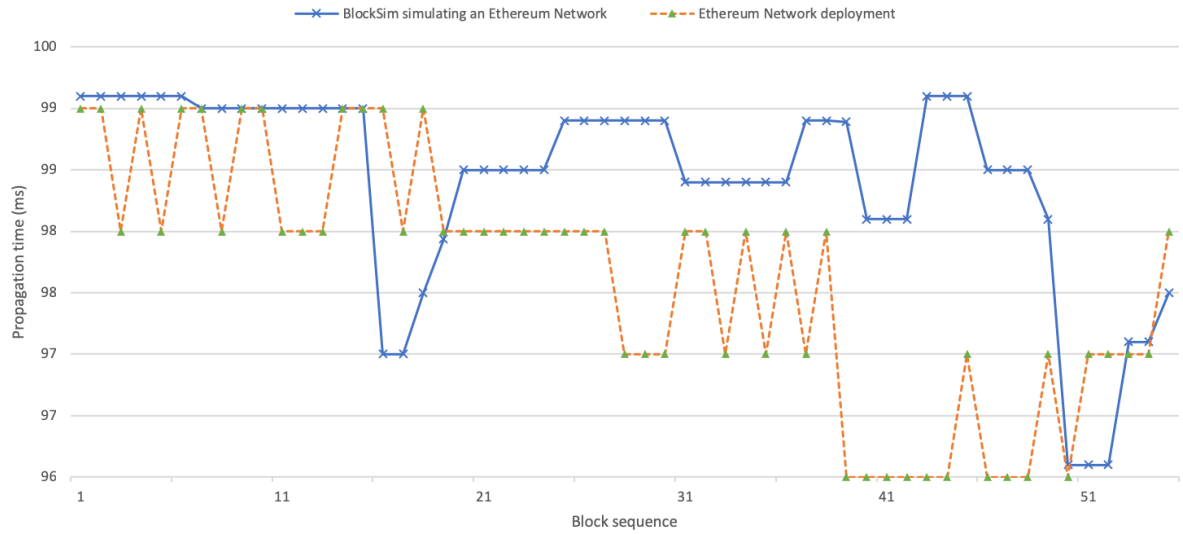
Figure 4.3: Transaction propagation between Ohio and Ireland.

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 4.3 and Table 4.2 answer to our study question (*cf.* Section 4.1.1). 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.*



(a) Overview.



(b) Detailed view.

Figure 4.4: Transaction propagation between Ireland and Tokyo.

4.2 BlockSim Use Cases

We have demonstrated in Section 4.1.2 that BlockSim runs our Ethereum models in an accurate representation of a real Ethereum network with regards to block and transaction propagation. Hence, in this section we will explore and evaluate real use cases for BlockSim. Therefore, Section 4.2.1 explores the contrast of block propagation time with different block gas limits. Section 4.2.2 inspects the impact of encrypting all the network messages. Section 4.2.3 creates a simplified version for block delivery. Finally, Section 4.2.4 studies the impact of transmitting a transaction only once in the network.

All the BlockSim executions were conducted on a computer with 2 GHz Intel Core i7 pro-

Table 4.3: 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
Between Ohio and Ireland	93 ms	93 ms	1.22 ms	1.12 ms
Between Ireland and Tokyo	98 ms	98 ms	1.01 ms	0.81 ms

cessor and 8 GB RAM.

4.2.1 Different Block Gas Limits

In this use case we show the impact in the block propagation time when increasing the block gas limit.

BlockSim was configured for this use case to create 10,000 transactions in a network with a total of 300 nodes: 100 non-miner nodes in Tokyo and 100 in Ireland; 50 miner nodes in Ireland and 50 in Tokyo. We used the same input parameters in Table 4.1.

A standard transaction in Ethereum has a 21000 gas limit [4] and we consider a standard transactions to have 200 Bytes. These values can be adjusted in the configuration file as input parameter (*cf.* Section 3.2.2).

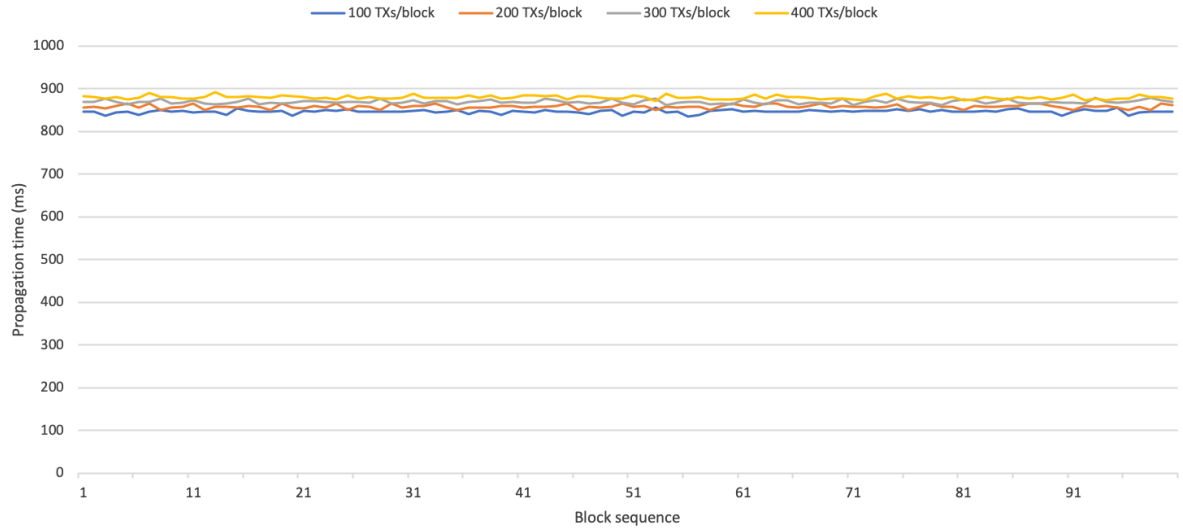
The block gas limit represents the maximum amount of gas allowed in a block, it determines how many transactions can fit into a block. For instance, if block gas limit is set to 100, we can fit four transactions with a gas limit of 10, 20, 30 and 40, or only two transactions with 50 gas limit. On public Ethereum blockchain the block 6441886 [37] 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 [4]. The block gas limit can be set in the configuration file as input parameter (*cf.* Section 3.2.2).

In order to simulate this use case, 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, thus including more transactions per block. In Table 4.4 it shows the average time for block propagation between Tokyo and Ireland, when increasing the number of transactions per block, in other words, when increasing the block gas limit.

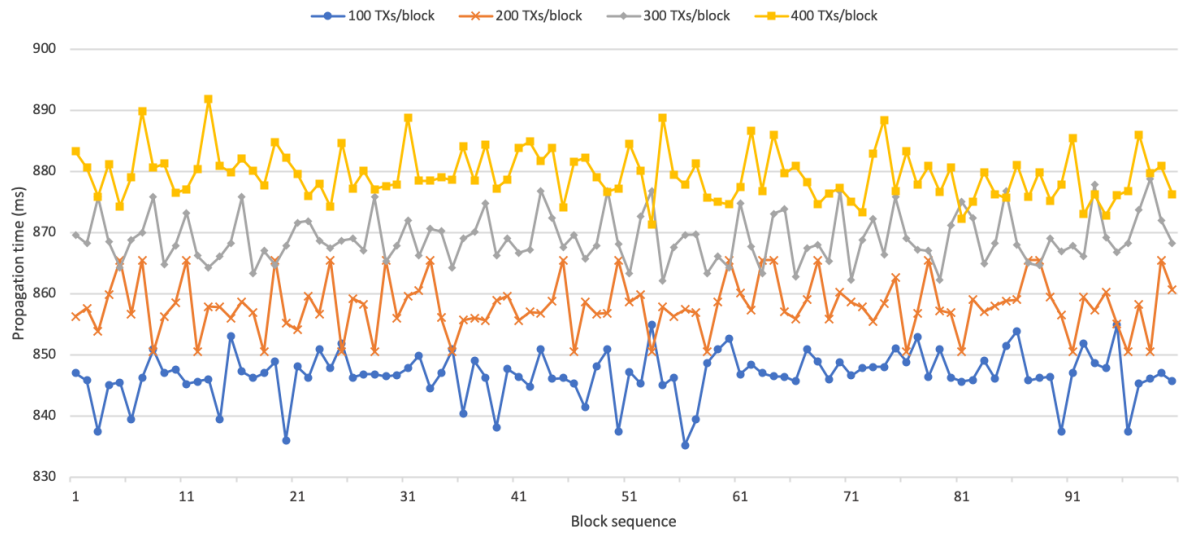
BlockSim successfully simulated this use case in 55 minutes and 31 seconds. The results in Table 4.4, that are shown in Figure 4.5, reveal an expected grow of 20 kB block size between each execution. This grow corresponds to an additional number of 100 transactions. Additionally, we can observe for each execution an increasing raise in propagation time of approximately 10 ms.

Table 4.4: 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 time	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



(a) Overview.



(b) Detailed view.

Figure 4.5: Block propagation for different number of transactions per block between Tokyo and Ireland.

4.2.2 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.

BlockSim was configured for this use case to create 2,000 transactions in a network 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 4.1.

A node in order to obtain a new block receives four distinct messages: Status, NewBlockHashes, BlockHeaders and BlockBodies. Also, the node needs to send two messages: GetBlockHeaders and GetBlockBodies (*cf.* Section 3.5.1).

To simulate this behaviour we have added to our basic node model (*cf.* Section 3.3) a fixed delay when receiving and sending a network message.

Table 4.5 presents the impact in block propagation for two different delays to encrypt and decrypt each message: 100 ms and 50 ms.

Table 4.5: 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

BlockSim successfully simulated this use case in 27 minutes and 10 seconds. The results captured between Tokyo and Ireland in Table 4.5 shows a 25.8% increase in the block propagation time when encrypting and decrypting messages with a delay of 50 ms. Furthermore, a 51.6% increase is observed when encrypting and decrypting messages with a delay of 100 ms.

4.2.3 Simplified New Block Delivery

In this use case we model a new message exchange protocol, used to a node obtain a new mined block.

BlockSim was configured for this use case to create 2,000 transactions in a network 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 4.1.

We presented in Section 3.5.2 the PV62 [36] message exchange protocol, that was illustrated in Figure 3.6. This protocol is the standard protocol used, and the one that we model.

We adapted and simplified our model to make the node request the full blocks (headers and bodies) when the message NewBlockHashes is received, as illustrated in Figure 4.6. In order to adapt our model, we simply needed to create two new network messages[3]: *GetBlocks* that

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

requests the full blocks by the hashes; *Blocks* that sends the requested full blocks. Additionally, the Ethereum node model[4] was changed to respond the new network messages accordingly.

This simplified new block delivery is similar to the Bitcoin protocol (*cf.* Section 3.4.2). Also, the PV63 Ethereum protocol [36] follow a similar design, and is only used when the node is not synchronise with the rest of the network.

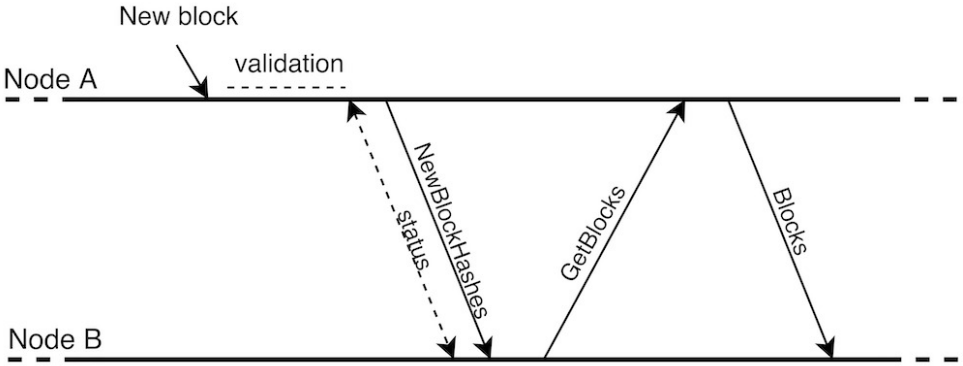


Figure 4.6: Adapted message exchange protocol to obtain a new block.

BlockSim successfully simulated this use case in 20 minutes and 12 seconds. The results captured between Tokyo and Ireland in Table 4.6 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 verifications and 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 [38].

Table 4.6: 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 4.6)			610 ms

We can observe that our simulator, with the same initial conditions, has simulated this use case in less time than the previous (*cf.* Section 4.2.2), because there is less overhead with the message exchange.

[4]Available at: `https://github.com/BlockbirdLabs/blocksim/blob/ethereum-use-case/blocksim/models/ethereum/node.py`

4.2.4 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 (*cf.* Section 3.5.2).

In order to avoid propagating each transaction twice, 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 [39, 40, 41, 42].

However, before implementing a reconciliation protocol 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 4.7. The adapted message exchange protocol[5] simulates the best scenario of a reconciliation protocol when Node A owns all the transactions in the newly mined block.

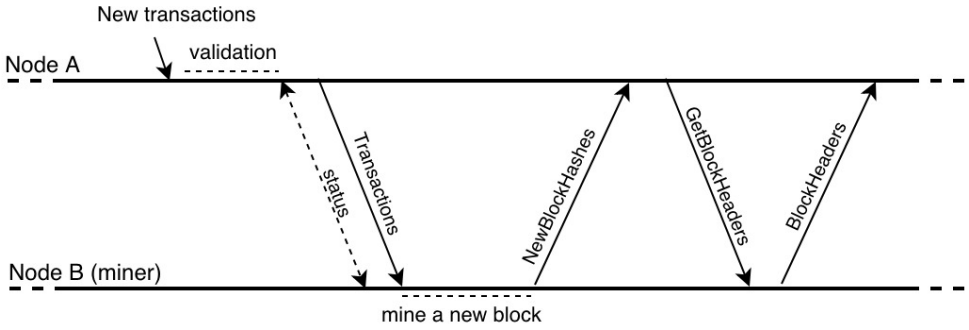


Figure 4.7: Adapted message exchange protocol that only deliver block header.

BlockSim was configured for this use case to create 9,000 transactions in a network 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 4.1.

Table 4.7: 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 4.7)			600 ms

BlockSim successfully simulated this use case in 35 minutes and 2 seconds. The results captured between Tokyo and Ireland in Table 4.7 shows a 29.2% decrease in block propagation

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

time when simulating the impact for one transaction propagation policy.

4.3 Summary

This chapter presented evaluation conducted on the BlockSim, a blockchain simulator.

The first section of this chapter performed a verification and validation of the Ethereum models by comparing the block propagation time in the simulation with a real private Ethereum network. The results showed that BlockSim runs our models with an accurate representation of the Ethereum system with regards to block and transaction propagation.

The second section of this chapter successfully and easily use BlockSim to study four real use cases.

The first use case (*cf.* Section 4.2.1) measured the impact of increasing the number of transactions inside a block. The results showed that when increasing the number of transactions in a block, it takes more time to propagate each block.

The second use case (*cf.* Section 4.2.2) demonstrated that encrypting and decrypting all messages may have a minimum increase of 25.8% on block propagation time.

The third use case (*cf.* Section 4.2.3) creates a simplified version for block delivery which have decreased the block propagation time in 27.9%.

The last use case (*cf.* Section 4.2.4) studies the impact of transmitting a transaction only once with a 29.2% decrease in block propagation time.

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 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 4.1) 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.

Chapter 5

Conclusions

Blockchain systems are becoming complex distributed systems. There is a broad interest in developing methods to evaluate these systems. One alternative is to use emulation; however, this approach incurs a large overhead, lacks in scalability to real world deployments, and has high power consumption. Another alternative is to use simulation. This method enables the evaluation of a large-scale system in a reasonable time.

BlockSim, to the best of our knowledge, is 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.

Additionally, the BlockSim 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.

5.1 Achievements

Our main goal for this thesis, as previously mentioned, was to provide a simulator capable of evaluating different blockchains in different environment conditions, which we have accomplished in this work.

Our work on implementation and design of a blockchain simulator, made this simulator flexible to extend or replace different models, and to easily simulate other blockchain systems

following different consensus models or protocols. Besides all that, BlockSim is also capable to run thousands of nodes in a single host in reasonable time, compared to other mechanisms, like emulation.

Additionally, we have shown an accurate representation of the Ethereum system and how easy it was to change the simulated environment conditions and models to study peculiar use cases.

Lastly, we used BlockSim to explore a range of intriguing real use cases, each one with an important role on understanding more from the blockchain systems.

5.2 Future Work

To enhance this work, we believe some improvements could be made in future work. As an example of it, we propose the following points:

- Use PeerSim [16] to model a node discovery protocol to enable an accurate simulation of a real P2P network.
- Introduce more fine-grained capabilities inside the node model; for instance, CPU use and power consumption modelling for certain cryptographic operations.
- Improve the Ethereum model adding the GHOST protocol.
- Perform validation and verification of Bitcoin models, as it was made to the Ethereum models, by following the same process. However, the Ethereum simulation uses the majority of Bitcoin base models, which have been successfully validated.
- Use BlockSim through the framework to model new blockchain systems.

Bibliography

[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*, pp. 106–125, Springer, 2016.

[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*, pp. 51–68, ACM, 2017.

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

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

[5] 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.

[6] 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*, p. 30, ACM, 2018.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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*, pp. 135–144, 2009.

[13] 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*, pp. 1085–1100, ACM, 2017.

[14] 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.

[15] 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*, p. 55, 2009.

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

[17] 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.

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

[19] J. R. Douceur, "The sybil attack," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pp. 251–260, Springer-Verlag, 2002.

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

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

[22] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *International Conference on Financial Cryptography and Data Security*, pp. 507–527, Springer, 2015.

[23] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

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

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

[26] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[27] R. Jansen and N. Hopper, "Shadow: Running tor in a box for accurate and efficient experimentation," in *Proceedings of the 19th Symposium on Network and Distributed System Security (NDSS)*, Internet Society, February 2012.

[28] 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, pp. 189–204, 2008.

[29] 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*, CCS '16, pp. 3–16, ACM, 2016.

[30] 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.

[31] 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*, Middleware '17, pp. 19–20, ACM, 2017.

[32] A. Gervais, "arthurgervais/Bitcoin-Simulator: Bitcoin and Blockchain Simulator." `https://github.com/arthurgervais/Bitcoin-Simulator`, 2018. [Online; accessed 06-Sep-2018].

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

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

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

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

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

[38] "Light client protocol · ethereum/wiki Wiki." https://github.com/ethereum/wiki/wiki/Light-client-protocol, 2018. [Online; accessed 10-Oct-2018].

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

[40] 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.

[41] 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.

[42] 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*, p. 6, ACM, 2008.