Technische Universität München

Fakultät für Informatik

Master's Thesis in Informatik

Design and Implementation of an Extensible Large-Scale Blockchain Simulation Framework

Halil Can Salis

Technische Universität München

Fakultät für Informatik

Master's Thesis in Informatik

Design and Implementation of an Extensible Large-Scale Blockchain Simulation Framework

Entwurf und Implementierung eines erweiterbaren, umfangreichen Blockchain-Simulationsframeworks

Author: Halil Can Salis

Supervisor: Prof. Dr. Hans-Arno Jacobsen

Advisor: M.Sc. Pezhman Nasirifard

Submission: 15.11.2019

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, 15.11.2019

Halil Can Salis

Abstract

The global interest in blockchain technologies started with the introduction of Bitcoin in 2008, and it is growing ever since. Many of these technologies are still under active development, and researchers have been working on new proposals to improve their protocol designs. Every new change has the potential to bring both benefits and risks to the users. Hence, researchers use tools to prototype and test their ideas before changes land in protocols. Existing testnets and testing suites require many resources to simulate real networks. They are neither efficient enough to simulate networks on a single computer nor extensible enough to support more than one blockchain technology. The shortage of such testing tools led to the research and development of extensible and efficient simulators at TU Munich. Many of these simulators were efficient enough to simulate one blockchain technology, but they fell short of the promise of being truly extensible. This paper proposes a truly extensible simulation framework that aims to address the shortcomings of previous simulators while maintaining their efficiency. It also introduces two example simulations that demonstrate capabilities and possible use cases of the framework.

Inhaltsangabe

Das globale Interesse an Blockchain-Technologien begann mit der Einführung von Bitcoin im Jahr 2008 und wächst seitdem stetig. Viele dieser Technologien befinden sich noch in der aktiven Entwicklung, und die Forscher haben an neuen Vorschlägen zur Verbesserung ihrer Protokolldesigns gearbeitet. Jede neue Änderung hat das Potenzial, den Nutzern sowohl Nutzen als auch Risiken zu bringen. Daher verwenden Forscher Werkzeuge, um ihre Ideen zu prototypisieren und zu testen, bevor Änderungen in Protokollen landen. Bestehende Testnetze und Testsuiten benötigen viele Ressourcen, um reale Netzwerke zu simulieren. Sie sind weder effizient genug, um Netzwerke auf einem einzelnen Computer zu simulieren, noch erweiterbar genug, um mehr als eine Blockchain-Technologie zu unterstützen. Der Mangel an solchen Testwerkzeugen führte an der TU München zur Erforschung und Entwicklung von erweiterbaren und effizienten Simulatoren. Viele dieser Simulatoren waren effizient genug, um eine Blockchain-Technologie zu simulieren, aber sie blieben hinter dem Versprechen zurück, wirklich erweiterbar zu sein. Dieses Papier schlägt einen wirklich erweiterbaren Simulationsrahmen vor, der darauf abzielt, die Mängel früherer Simulatoren zu beheben und gleichzeitig ihre Effizienz zu erhalten. Es werden auch zwei Beispielsimulationen vorgestellt, die Fähigkeiten und mögliche Anwendungsfälle des Frameworks demonstrieren.

Acknowledgment

I would like to thank my supervisor Prof. Dr. Hans-Arno Jacobsen for giving me the opportunity to write this thesis at his chair. I would like to thank my advisor M.Sc. Pezhman Nasirifard for his help and guidance through each stage of this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A paper published by Satoshi Nakamoto titled "Bitcoin: A Peer-to-Peer Electronic Cash System"[9] has sparked a global interest in blockchain technologies. Even though the initial application area of the blockchain technology was cryptocurrencies, it has started to be used in other areas such as finance, logistics, supply chain, cross-border payments and global digital identity.

Like any other emerging technology, blockchain technologies are also evolving rapidly. Researches are developing new protocols and proposing changes to existing ones in order to improve the systems in terms of scalability, efficiency, decentralization and security. Every new extension or protocol also brings new risks and challenges to blockchain users. Correct assessment of the potential behavior of a distributed system requires an extensive effort for analysis and testing. Because of this reason, often times the safety and reliability properties of new protocols are not sufficiently analyzed.

Researchers and developers make use of simulators to test and analyze systems' behavior under certain conditions. These simulators are also useful for analyzing potential attacks. There are already simulators that exist to test blockchain systems such as testnets [10, 11, 12], Ganache [13] and Hive [14], but all of them serve different purposes and are neither extensible nor viable solutions to assess safety and reliability properties efficiently.

This paper mainly focuses on the previous research conducted at the Chair of Application and Middleware Systems at TU Munich for the development of extensible and scalable simulators and proposes a new approach that would allow rapid development of such simulators.

1.1 Motivation

The main motivation of this thesis is the lack of a general purpose, truly extensible and scalable simulation framework. Blockchain research is a relatively new topic, and blockchain developers are in need of tools that would assist them in their research. Tools such as simulators allow researches to test the scalability and security of the protocols and make prototyping and testing new proposals easier. Developing such simulators has been a very demanding task, because each blockchain is different, and they all have their unique characteristics. Thus, research and development of a distributed ledger simulation framework would be very beneficial to the community and would assist the development of such simulators. Allowing rapid development of such simulators benefits both the researches and blockchain users.

1.2 Problem Statement

Some simulators like Simcoin [15] are built for educational purposes and demonstrate how Bitcoin works to newcomers. There are also public testnets [10, 11, 12] that offer easy to use networks resembling their respective mainnets [16] with shorter block times and reduced maintenance costs so that developers can verify their transactions. Additionally, tools such as Ganache [13] and Hive [14] are also popular in the field. The former allows developers to deploy and test smart contracts while the latter is used for block-box testing of the clients against the Ethereum specification. These tools are built only to serve a single purpose and neither extensible nor efficient enough to simulate the whole network on a single machine.

There is also an ongoing research for blockchain simulators at the Chair of Application and Middleware Systems at TU Munich that has led to the development of simulators such as VIBES [5], BBSS [17], eVIBES [18], eVIBES Plasma [6] and CIDDS [7] that aim to fulfill the shortcomings of available tools in the field. However, all of them fall short of the promise of being truly scalable and extensible. 3 out of 5 of these simulators had to be implemented from scratch in order to support a new blockchain technology. Even though each simulator successfully manages to simulate what they are developed to simulate, they all lack true extensibility because of their design limitations. Another problem is that their ability to scale is very limited, and they do not allow running simulations on clusters.

1.3 Approach

Previous research for the simulation of different blockchain technologies at the Chair of Application and Middleware Systems at TU Munich is an excellent source of information to understand the requirements for a general purpose simulator. It also proves that building a simulator for a specific blockchain protocol and then trying to extend it to simulate different protocols or consensus mechanisms is a very hard task.

Under the light of this information, one can conclude that it is necessary to approach this problem with a different mindset. A better approach would be to build a foundation that can provide the common functionality needed in every simulator, and then to implement various simulators on top of this foundation. In addition to offering common building blocks, blockchain specific functionality can be provided via feature modules so that developers can include these feature modules when they are needed.

1.4 Contribution

This paper proposes a new framework named Distributed Ledger Simulation Framework (DLSF). DLSF's goal is to support a wide variety of simulations, and many of its features are already offered partially in existing simulators. DLSF combines all the common functionality needed to develop such simulations with a modular and efficient architecture. Main contributions of the framework and its differences are as follows:

- DLSF provides a foundation for simulation systems that could support any blockchain technology. This is what makes it unique compared to existing simulators. Existing simulators were all designed to simulate different scenarios on a single type of blockchain, which made them difficult to use for different blockchain simulations.

- DLSF can be thought of as a simulation runtime system where each simulation is added as an extension and managed by the framework. Unlike other simulators, the whole framework runtime architecture is decoupled from the implementation details of simulations and therefore highly extensible.

- DLSF includes a very efficient messaging backbone and state isolation mechanisms. Similar to other simulators developed at TU Munich, this is achieved by utilizing Akka framework [19]. In comparison, Simcoin [15] uses Docker for node isolation and Blockchain Simulator [20] uses ns-3 for simulated network messaging. Tools such as Docker and ns-3 only offer a small portion of the functionality needed to build blockchain simulations and often increase the complexity of simulators or reduce their efficiency. Besides the highly efficient Akka libraries, DLSF does not rely on any other component to provide these functionalities.

- All existing simulators have their own custom implementations for things such as: validating and parsing configurations, starting and terminating runs, sending real-time updates and providing results. Whether it is a CLI application like Simcoin and Bitcoin Simulator, or a system deployed on a server like VIBES, every simulation needs to handle some of these use cases. DLSF offers streamlined HTTP endpoints and web socket abstractions that support all possible user interaction patterns. Offering this feature out of the box reduces the development time of simulations. DLSF runtime acts as a broker between simulations and users by providing abstractions to the simulation code and HTTP endpoints to the outside world.
- DLSF is designed to be very modular and extensible. It includes feature modules for additional functionality that is not in the scope of the framework's core. One of these modules is the DLSF Bitcoin module, which includes Bitcoin related data structures and a neighbor discovery helper component. In comparison, existing simulators provide blockchain specific components as part of the main project, but in DLSF these are optional modules that can be included in any simulation. This design allows offering modules for different distributed ledger technologies without cluttering the core of the framework.
- Unlike many other simulators, DLSF allows running simulations both on a single machine and on clusters of servers. Section 4.4.1 discusses possible simulation system architectures.

In addition to the research and development of the framework, two example Bitcoin simulations are developed. The main purpose of these simulations is to demonstrate capabilities and possible use cases of the framework. Contributions of these simulations are as follows:

- Both simulations use a very efficient mining simulation process described in Section 4.9.5. In comparison to the Coordinator component used in VIBES and BBSS, this approach demonstrates how random numbers can be used to simulate proof of work consensus protocols more realistically and efficiently.
- Block explorer simulation discussed in Section 4.9 is very similar to the real world block explorers, but it allows exploring simulated chains, and includes more than 10 parameters to configure different aspects of the protocol.

- Transaction protocol simulations discussed in Section 4.10 are used to test the Erlay protocol against the existing Flood protocol. It is inspired by a similar simulation created by the authors of Erlay protocol paper [3] using Bitcoin Simulator.

1.5 Organization

This thesis is organized as follows: Chapter 2 discusses all the necessary topics to understand the fundamentals of blockchain. It explains individual components of the Bitcoin protocol and describes how it works. In addition to fundamental technologies, it also summarizes one of the work in progress Bitcoin improvement proposals called Erlay Protocol which is simulated by one of the example simulations. Chapter 3 gives an overview of the similar tools that acts as a foundation of this thesis. It gives an overview of the simulators developed to address the same problem and shortcomings of these simulators that needs to be addressed. Chapter 4 discusses the requirements and implementation details of both the framework and the example simulations. The Chapter 5 evaluates the approach taken and the end results. Finally, Chapter 6 summarizes the status of the framework and gives conclusions about the research that is done as part of the thesis.

Chapter 2

Background

This chapter discusses the topics that are necessary to understand the further chapters in the thesis. Topics that are covered here are necessary to understand both the problem and the approach taken to tackle it. It gives an overview of topics such as peer-to-peer networks, cryptography and Bitcoin. Finally, the chapter gives a summary of the Erlay protocol, which is a work-in-progress Bitcoin improvement proposal that aims to reduce the bandwidth usage and increase the network's security.

2.1 Simulation

A simulation is an approximate imitation of the operation of a process or system [21]. Simulation in the context of computer science can be used for many purposes. They can be used to understand how a system function, to test how a system behave under different conditions and to show how changes would affect the overall behavior. Simulations can also be used when running or testing systems are expensive in terms of resources. Simulations are expected to be similar to what is being simulated in areas relevant to what is critical for its purpose because they ultimately affect the validity of the outcomes.

2.2 Peer-to-peer Networking

Peer-to-peer networking is a type of application architecture that is distributed by nature. Compared to more traditional architectures such as client-server, its peers are mostly equally privileged. Peers make their resources, such as computing power or bandwidth available to others. Such networks are distributed in nature because they have no distinct roles such as a server or a client. In peer-to-peer networks, resources are shared and work is distributed among the participating nodes.

2.3 Cryptography

"Cryptography is the practice and study of techniques for secure communication in the presence of third parties called adversaries" [22]. Information security topics such as authentication, data integrity, confidentiality and non-repudiation are fundamental to modern cryptography [23].

2.3.1 Cryptographic Hash Function

Cryptographic hash functions are functions that take an arbitrary sized message as an input and output a fixed sized hash data. Moreover, cryptographic hash functions have the following characteristics:

- The same input always generates the same output. Therefore, they are deterministic.
- Computing a hash value of any given input is not a heavy task.
- Given an output, it is not computationally feasible to find an input that would generate the said output.
- Finding two different inputs that yield the same output is not computationally feasible.
- Even the smallest change - such as a single bit change - to the input would change the output significantly.

Thanks to these characteristics, cryptographic hash functions are essential and are heavily used in modern cryptography [24]. Some of the applications are file integrity verification, file identifier generation, signature generation and password verification.

2.3.2 Encryption & Decryption

Encryption is the process of generating meaningless output (Ciphertext) from an original message (Plaintext), and decryption is the reverse of encryption. A Ciphertext is impossible to understand unless decrypted and therefore used to hide information from third parties.

2.3.3 Digital Signature

Digital signatures are used to check the authentication and integrity of a message and make use of asymmetric cryptography algorithms like ECC or RSA. These schemes are used by many cryptographic protocols. An entity can only create signatures for themselves, but anyone can verify it. Entities can use signatures to sign different kinds of data. Digital signatures also make sure that once an entity signs a document, they cannot deny having done it later. This property is also called non-repudiation [25].

2.3.4 Public and Private Key

Public and private key pairs are used in asymmetric cryptography. Former can be openly shared to third parties and the latter should be kept secret to ensure security [26]. One of the pairs, the public key allows anyone to encrypt data that can only be decrypted by the private key holder. Unlike older symmetric algorithms where a single shared key is used for both encryption and decryption, public/private key pairs provide superior security and ease of use.

2.4 Distributed Ledger

Distributed ledger is an alternative to centralized data storages and does not require any central authority to function. Distributed ledger technologies use peer-to-peer networks and consensus mechanisms for consistent data replication across the nodes. Security is accomplished by using cryptographic keys and signatures [27]. Blockchain is the most popular type of distributed ledger, but there are other technologies that use different protocols and data structures such as IOTA's tangle [28].

2.4.1 Blockchain

Blockchain is both a data structure and the name of the widely used distributed ledger design. The chain consists of a growing list of blocks that are linked to one another. With blockchain, parties can access, record and verify transactions without needing to trust each other. Records in blockchain are permanent and cannot be tempered with. A simplified illustration of a Bitcoin blockchain is shown in Figure 2.1.

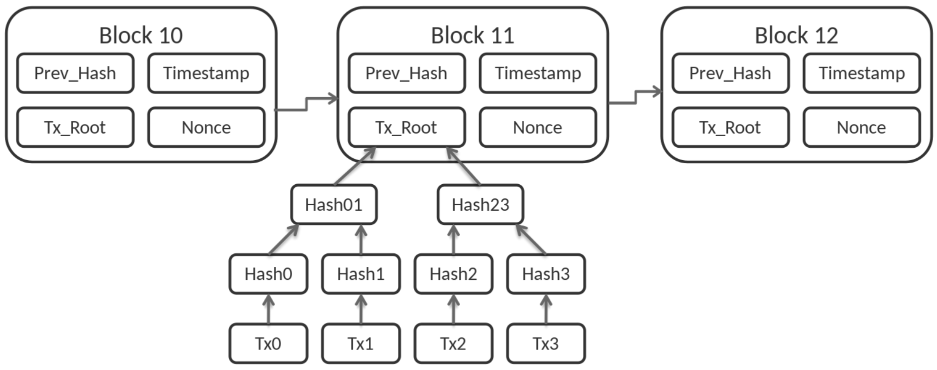


Figure 2.1: Simplified data structure of blocks in the ledger [1].

2.4.2 Consensus Protocols

Consensus protocols are used to guarantee the security and fault tolerance of distributed systems and are a fundamental part of any distributed ledger technology. It is still a heavily researched area of topic. Each consensus protocol inherently make a trade-off between consistency, availability and partition tolerance [29], and there is no silver bullet or a perfect protocol.

Consensus protocols need to have some sort of mechanism to deal with malicious actors. These actors can potentially weaken or compromise the consensus process and can also carry-on attacks on other honest nodes. This characteristic is even more important in public blockchains where access is not restricted to the network. This is also called the Byzantine generals problem, and each protocol addresses it differently.

51% Attack

One of the possible attacks on such networks is called 51% attack. Once an entity has control of the majority of the hash power or the total currencies in a network, it would have the ability to change the history or could block new transactions.

Proof of Work

Proof of work protocols rely on puzzles that are computationally hard to solve but easy to verify. One simplified example is mining in Bitcoin where miners try to generate the hash of a new block header that is prefixed with a certain number of 0s. As mentioned in the cryptographic hash function chapter, there is no easy way to find a block of data which generates hashes with certain characteristics, but on the other hand, it is very easy to verify received blocks. Moreover, the mining process in Bitcoin ensures trustworthiness and gives miners incentives to operate the network by rewarding them bitcoins.

Bitcoin uses a proof of work protocol and operates reliably since it first went online ten years ago. Yet, there is a growing concern for the energy use of the network [30]. One can expect this problem to be more concerning for the environment as the network grows bigger in the years to come.

Proof of Stake

Proof of stake is another consensus mechanism, where miners have a chance to mine a block based on their stake on the network. For instance, a miner with 5% of all the currency can theoretically only mine 5% of the blocks. Since miners do not compete to solve difficult puzzles, a PoS consensus is much more efficient compared to PoW.

A network with PoS consensus is also less vulnerable to a 51% attack since one needs to have 51% of all the cryptocurrencies to be successful. Attacking a network where the attacker has more stake than anyone else would not benefit them and would be very costly.

2.5 Bitcoin

Bitcoin is first described in a whitepaper published by an individual or a group named Satoshi Nakamoto [9]. Their real identity is still unknown to this date. First implementation of the protocol is released one year later. Bitcoin has been the most popular blockchain application to this date. It sparked great interest in the field of distributed ledger technologies as well as for cryptocurrencies.

2.5.1 Network Topology

Nodes in Bitcoin networks create a random topology by following a limited set of rules in the protocol. Each full node connects up to 8 other nodes when they first join the network. Connections that are initiated by a node called outbound connections. By default, a node can also accept connections from other 117 nodes making its total connection count 125. These connections that are initiated by other nodes in the network are called inbound connections.

2.5.2 Addresses

Bitcoin makes use of asymmetric cryptography. Entities that want to send and receive bitcoins need to create an address first. An address is created by first generating a public/private key pair and then running the public key through some function that generates shorter and less confusing addresses. It is not possible to revert this process to find the public key from an address. From then on, only the private keys and addresses are used to send and receive bitcoins.

Private keys are used to sign transactions originating from derived addresses and prove that they are coming from the owner. Once a private key is lost or forgotten, any bitcoins belong to an address derived from it are locked forever and cannot be spent.

2.5.3 Blocks

Blocks are one of the two main data structures used in Bitcoin, and each block consists of a header followed by a number of transactions. Each block is built on top of another block to collectively form a chain and the hash of the previous block is kept in the header.

Blocks act as permanent records of transactions in the network, and once confirmed, they cannot be altered or removed unless an entity gains control of the majority of the hashing power. Blocks are also tamper-proof because of the nature of cryptographic hash functions. Even a slight change would change the hash of a block and therefore, would render it invalid.

Block Header

Block header acts as metadata for the block and keeps information such as previous block hash, Merkle root of transactions, time of successful mining, target and nonce.

- **Version:** Version of the block.
- **Previous block hash:** Hash of the block that this block is built upon.
- **Merkle Root:** A hash of all the transactions within this block that can be used as a fingerprint. It allows efficient way of checking if a transaction is part of the block without knowing all the other transactions in the tree.
- **Time:** Unix time of successful hashing of this block.
- **Bits:** Short version of target, which is used to validate the hash and to check the difficulty.

- **Nonce:** Random bits that are used to alter the hash with the aim of getting it below the target.

Block Hash

Block hash acts as an identifier for a block in the blockchain. It is created by hashing the block header twice via the SHA-256 algorithm.

Candidate Block

Candidate block is a temporary block that is being worked on by a miner. It includes transactions that are announced to the network but not yet confirmed. Miners can choose which blocks to include and usually prioritize transactions with higher fees. Each miner then tries to mine this block to add it to the blockchain.

Block Time

Block time is the time it takes the blockchain network to generate one valid block and it is fixed to 10 minutes by the protocol. However in reality, total hashing power is never fixed and block times can vary. The network reacts to such changes by adjusting the difficulty to keep the block time close to 10 minutes.

Difficulty

In bitcoin, every valid block must have a hash below a certain target and the difficulty refers to the difficulty of generating hashes below this target. Difficulty regulates how long it takes to add a new block to the chain and it is adjusted every 2016 blocks in order to keep the block time around 10 minutes.

Block Reward

Bitcoin protocol gives new bitcoins to miners for each successfully mined block. These bitcoins are called block rewards, and they are found in the first transaction of a transaction list in a block. This first transaction is also called the coin-base transaction and it does not use any unspent transaction as input. The reward is halved after each 210,000 mined blocks and would eventually hit zero.

Block Size Limit

Blocks in bitcoin has a fixed size limit and cannot exceed 1 MB. Increasing this limit is a controversial topic and requires consensus among the miners. This is also the main limiting factor of the number of confirmed transactions per second in the network.

Genesis block

Genesis block is the only block that does not reference a previous block in its header, and it is the first block in the chain. It is usually hardcoded in client implementations, and its reward is not spendable.

Best Block Chain

Best block chain refers to a chain of blocks that are most difficult to recreate. It is also referred to as the active chain or the main chain.

Orphan block

Orphan blocks are well-structured blocks that are built on top of a block unknown to the receiving node. They are detached from the blockchain and might become either stale or even confirmed once the parent is known in some later time. Orphan blocks usually occur as a result of latency in the network.

Stale Block

Stale blocks are also valid blocks that are built on top of a known block, but they are not part of the best blockchain. Since they are not part of the active chain, their rewards are cannot be spent. Miners do not use stale blocks to built upon, and stale blocks might become un-stale at some time in the future as a result of chain reorganization.

2.5.4 Transactions

Transactions are the only way of transferring bitcoins from one address to another. They are broadcast to the network and then collected by miners into a block after they are validated and verified. They can also be used to create smart contracts on the blockchain.

Whenever a full node receives a transaction, it puts it through a set of tests defined by the protocol to ensure its validity. Invalid or unverified transactions are ignored, and not propagated to the network.

Each transaction consists of inputs and outputs.

Transaction Inputs

Transaction inputs, or TxIns in short, are used to specify the value being spent in a transaction. Transactions can have more than one input. A transaction input has the following fields.

- **Previous Tx Hash:** Hash of a transaction to be used.
- **Previous TxOut index:** Index of an output of a transaction to be used.
- **Script:** Transaction script.

Transaction Outputs

Transaction outputs, or TxOuts in short, are used to specify the amount being transferred as well as the recipients. Transactions can have more than one output.

- **Value:** Amount of Satoshi (0.00000001 Bitcoin) to be transferred.
- **Script:** Transaction script.

UTXO

Unspent transaction outputs, or UTXOs in short, are candidate transaction inputs that can be used at some later time. Since Bitcoin does not use balances or accounts, the entire state of the system consists of unspent transaction outputs. An address's total balance can only be calculated by computing the sum of unspent transaction outputs that are locked to that address. Bitcoin protocol prevents double spending of UTXOs, and therefore they can only be spent once. Once a UTXO has enough confirmations, it can be considered irreversible.

Transaction Scripts

Scripts are mini programs that are coded into TxIns and TxOuts. It is written in a language called Script that is a very basic, stack based programming language. Locking scripts can be found in TxOuts. Corresponding unlocking scripts are used to unlock these inputs and to use them as an output. UTXOs can only be spent when both of these scripts are valid together. Script makes creating different types of locks possible and opens up the way to smart contracts on the blockchain.

Transaction Confirmations

Whenever a new transaction is created, it is first published to a full node, and then it is propagated into the network for miners and other interested nodes to pick up. At this point, said transaction is deemed unconfirmed. Miners collect these new transactions and try to mine them into a block. When a transaction is mined into a block, it is considered to have one confirmation.

As the network accepts more transactions, new blocks would start to be mined on top of this block. Each block that builds upon this block increase the confirmation by 1. Users usually wait for one or more confirmations depending on the amount that has been transferred. After a transaction has enough confirmations, it can be deemed not reversible.

Transaction Fee

Transaction fee is the difference between inputs and outputs of a transaction. For instance, if a transaction uses 5 bitcoins worth of inputs and creates 2 bitcoin worth of outputs, the fee of this transaction would be 3 bitcoins. Miners claim fees from mined transactions through the coinbase transaction.

Transactions fees are one of the two incentives given to miners in bitcoin along with the block reward. Since block rewards are always halved and set to become zero in the future, fees will eventually become the sole incentive.

Transaction Pool

Transaction pool is the data structure that is used to keep unconfirmed transactions. Miners use this pool to pick transactions for their candidate block. Since block size is limited, most profitable blocks are prioritized to be confirmed first. Nodes other than miners can also keep a memory pool to gather insights about the network or to keep statistics. It is also called the memory pool because entries in this data structure are kept in memory and not on the disk.

Coinbase Transaction

Coinbase transaction is used by miners to claim the block reward and transaction fees. It is a special kind of transaction that has single blank TxIn and it is the first transaction in a block. In order to prevent invalidation of rewards in case of forks in the chain, coinbase transactions must have 100 confirmations before they can be spent.

Orphaned Transaction

Orphan transactions are valid transactions that use at least one unknown transaction input. This type of transactions cannot be added to the transaction pool, but they can be kept until all the unknown inputs are discovered. Once all the inputs are known, transaction is verified and either gets discarded or added to the transaction pool. As with any distributed system, latency also affects how transactions propagate in the network and could make input transactions propagate slower than the orphaned transaction.

2.5.5 Chain Reorganization

Chain reorganization is a situation when a node discovers a branch that is longer than its best block chain, which excludes some of the blocks in it. Node's internal state is re-organized to accommodate the new state. Excluded blocks become orphans, and some of the previously confirmed transactions might be reversed. These newly unconfirmed transactions are discarded if they are already spent in the new best block chain or added back to the transaction pool if they are unspent.

2.5.6 Node Types

There are three different types of nodes in any Bitcoin network, and nodes can act as more than one type. For instance, a node can be both a mining node and a full node.

Mining Nodes

Mining nodes confirm transactions and produce new blocks. Then, they send these blocks to full nodes in the network. In return, they receive rewards for every successful block they created. Mining nodes can also try to mine a single block by working together collectively.

Full Nodes

Full nodes are responsible for keeping a record of the entire blockchain ledger. They are capable of validating transactions and blocks, and keep the history of the blockchain. Full nodes communicate with each other and relay new information to its neighbors. They also validate transactions and blocks on demand. Having more full nodes keep the network more decentralized and help prevent some type of attacks.

Light Nodes

Light nodes are very similar to the full nodes with only one difference. Instead of keeping the whole ledger, they only keep a portion of it. They usually connect and rely on full nodes. Since they do not keep the whole ledger, they are less costly to run compared to full nodes.

2.5.7 Bitcoin Core

The reference implementation of Bitcoin is called Bitcoin Core. It is both free and open source, and the most popular software used within the network. Bitcoin Core acts as a node in Bitcoin network and can either run as a full node or as a miner. It is first published by the inventor of Bitcoin, Satoshi Nakamoto.

2.5.8 Bitcoin Improvement Proposals

Bitcoin is an evolving project, and its contributors conduct research to address its problems, shortcomings and to identify possible improvements. It has no formal structure, and there is no central standards body to steer the project direction. Instead, design documents called Bitcoin Improvement Proposals (BIPs) are used to describe features along with the reasoning and technical definitions.

BIPs allow project contributors and the community to discuss new proposal in detail and describes implementation details. Complete proposal process is defined in the first two BIPs and can be seen in Figure 2.2.

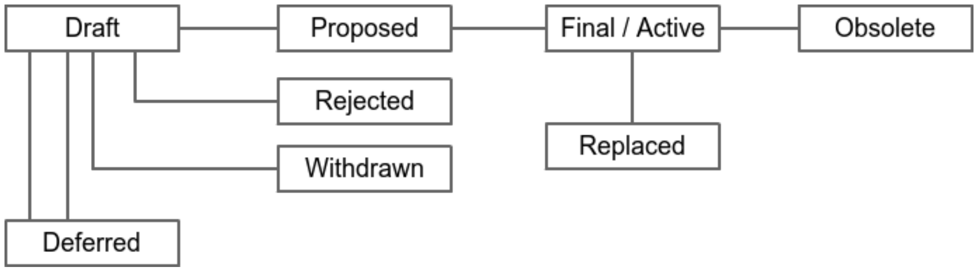


Figure 2.2: BIP Process [2].

Standards Track BIPs

This type of BIPs could be about the protocol, validation methods or interoperability, and requires community consensus.

Informational BIPs

These type of BIPs are created solely for information purposes. They can be about general guidelines, design issues or supporting information. They can either be acted upon or completely ignored.

Process BIPs

BIPs of these type propose a change to the process itself or describe it. Similar to standards track BIPs, they require consensus, but unlike them, proposals do not affect the protocol itself.

2.5.9 Erlay Protocol Proposal

Erlay is a protocol first proposed in May 2019 in a paper named Bandwidth-Efficient Transaction Relay in Bitcoin [3]. It is a work in progress Bitcoin improvement proposal draft created by a number of researches and maintainers. This new protocol proposes an optimized relay algorithm that would decrease the overall bandwidth usage. Authors argue that higher connectivity among nodes results in better security, but the current transaction protocol used by Bitcoin named BTCFlood uses almost half of the total bandwidth of the whole network and consequently prevents increasing the connection among nodes.

Shortcomings of Transaction Flooding

Current transaction relay protocol used in Bitcoin relies on flooding. It publishes announcements to every connection for every transaction it discovers. This results in many redundant messages in the network and makes transaction relaying very inefficient. Authors' experiments show that 55% of the total traffic of a typical node in the network is redundant.

Current protocol also does not scale well. Flooding sends announcement messages to each and every connection, thus increasing the number of connections also increases bandwidth usage linearly. This makes operating nodes more costly and prevents the community from increasing the connection limit. In turn, this results in weaker security because of the limited connectivity and potentially less number of nodes.

Protocol Requirements

To address the current protocol's shortcomings, Erlay proposal has set itself the following requirements:

1. It should scale better than linear as the number of connections increases. This would increase the security of the network without hindering the performance.
2. Mandating a structure overlay on top of the network in order to increase the bandwidth efficiency is one of the possible solutions, but this might result in partitioning or censorship. Therefore, the protocol should maintain the current network topology. Bitcoin's topology is intentionally unstructured, and each node has the freedom to choose which nodes to connect or how to route messages.
3. Propagation delay should be similar to that of the existing protocol.
4. Thread model of the existing protocol assumes that an attacker can have control over a number of nodes which does not exceed 50% of the network, and this attacker is free to generate traffic or intercept messages of its neighbors. Also, it has measures for denial of service attacks, should not leak any information and try to prevent client deanonymization. The new protocol should also be robust under these conditions.

Protocol Design

In order to be a better alternative, Erlay protocol relies on two techniques: delay and batching. Nodes announce new transactions only to the subset of its neighbors. Then to make sure transactions reach every node in the network, it periodically contacts one of its peers and tries to find missing transactions. The former technique is called low-fanout flooding and the latter is called set reconciliation.

Erlay uses its own custom library called Minisketch [31] for set reconciliation. It is an implementation of the PinSketch [32] algorithm and capable of finding differences in a set up to a certain threshold without needing to send the whole set, resulting in very small bandwidth usage. In Erlay protocol, nodes only announce transactions to 8 of their outbound connections and do the set reconciliation with a randomly selected connection in one second intervals.

Simulation Results

Authors focused on answering the following questions with their tests:

- How does it compare to the existing protocol in terms of latency and bandwidth usage.
- How does it scale with increasing number of connections, number of nodes and transaction rates.

- How malicious actors affect its performance.

In order to address these, the authors tested the protocol in two different settings. First by creating a simulation of a Bitcoin network with Erlay protocol using both ns3 [33] and a modified Bitcoin Simulator [20], and then with a reference Bitcoin Core implementation that ran on a 100 public cloud nodes across 6 different regions.

Simulation results show that Erlay significantly reduces the overall bandwidth usage. Compared to the existing flooding protocol, when the connectivity among nodes increases Erlay's bandwidth usage grows slowly as shown in Figure 2.3.

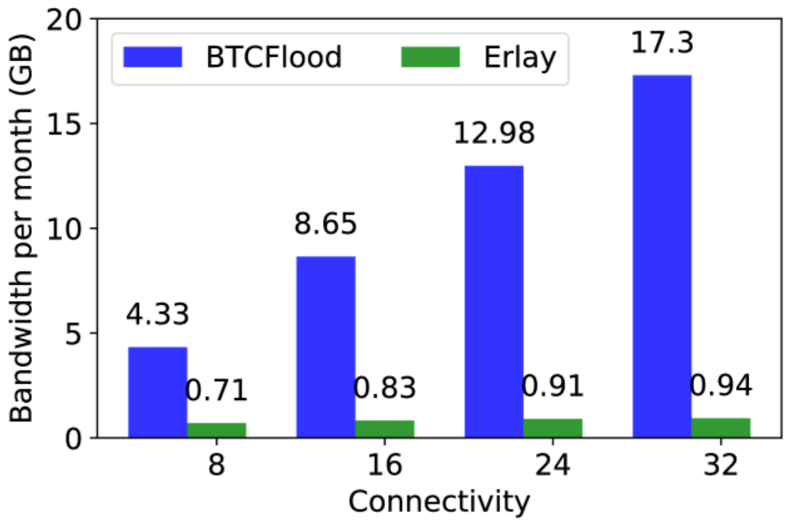


Figure 2.3: Erlay Simulation: Average bandwidth one Bitcoin node uses per month to announce transactions [3].

On the other hand, there are also downsides of using this new protocol. Authors observed that the time it takes for a transaction to be relayed to all the nodes is greater when Erlay is used. Therefore, Erlay increases the delay of transaction propagation. Furthermore, Erlay is more susceptible to spying by private nodes but the difference compared to the old protocol is very small and the success rate of the attack is below 50% for both protocols.

Chapter 3

Related Work

This chapter discusses the tools and simulators created to simulate blockchains and peer-to-peer networks, including the simulators created at the Chair of Application and Middleware Systems at TU Munich.

3.1 Simcoin

Simcoin [15] is a simulation framework that is developed to simulate a Bitcoin network on a single machine. It skips computationally heavy parts of the protocol, such as mining and other aspects of PoW for better efficiency. It is a command line tool, which stores results to a file on the local file system. Its output includes statistics such as stale block rate and block propagation times. Simulation nodes are real Bitcoin Core applications running in Docker containers, and they use TCP for communication among themselves. Networking among nodes is virtualized by utilizing networking features of Docker. Simulated nodes depend on another simulation component that instructs them when to mine blocks or create transactions. Simulation architecture is shown in Figure 3.1.

As already mentioned, Simcoin currently only supports Bitcoin by utilizing Bitcoin Core within containers. Even though Simcoin simulations are more efficient to run than a real Bitcoin network, its design relies on resource heavy components. It uses Docker to isolate nodes, which means each node runs on a container that not only simulates a node, but also virtualizes a whole operating system with its dependencies. Usage of TCP messages for inter-node communication also introduces an overhead. Even though all nodes reside on the same hardware, TCP requires each message to be serialized, sent over the network and deserialized at the target node. In addition to that, it's user experience is not the best. Having to use a CLI tool and dealing with output files and commands is not something a

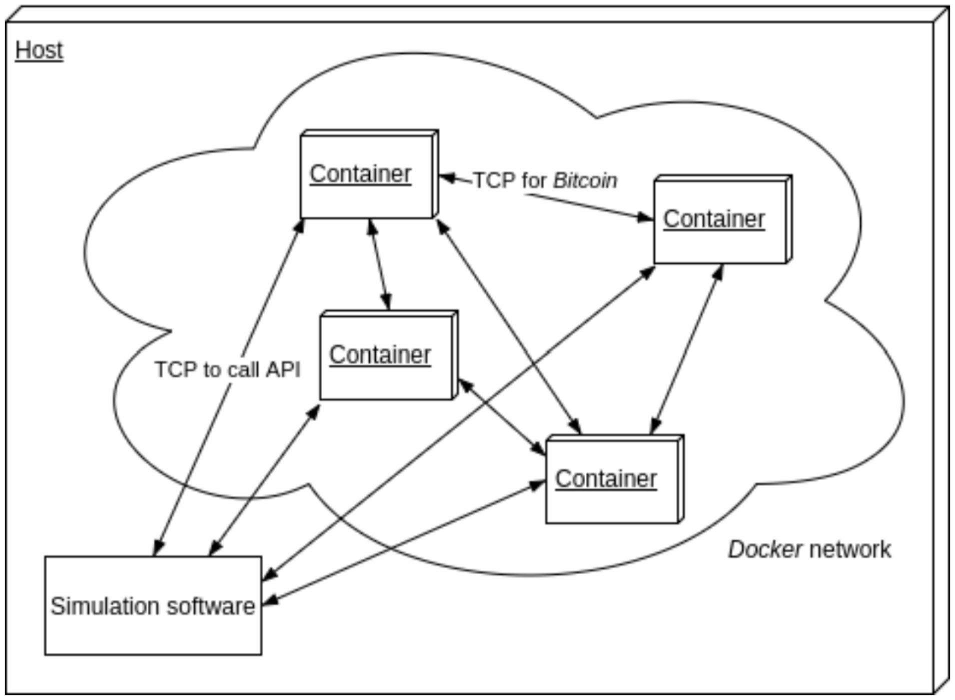


Figure 3.1: Architecture of a Simcoin simulation [4].

regular user can do. Finally, the configuration of some parameters such as difficulty, block size and throughput cannot be done without forking and changing the source code.

3.2 Testnets

Testnets [10, 11, 12], as the name implies are test networks which follow the certain characteristics of the mainnet [16]. Nodes on these networks run identical or slightly modified versions of software that runs on the mainnet. Thus, they have similar hardware requirements and are not efficient enough to run on a single machine for simulation purposes. Configurability of the nodes depends on the configurability of the existing protocol software. Coins on these networks do not have any value and contracts on these networks are only meant to be tested. This allows developers to experiment without fearing of the costs.

A blockchain can have many testnets that run on different configurations. Testnets can be configured to have less difficulty, less transaction fees and faster block times. Moreover, they could include proposals that are not part of the protocol or even be using a different consensus mechanism altogether. Since the whole purpose of these networks are experimenting and testing, they receive fewer transactions compared to the real network.

Almost every blockchain network has testnets in addition to their mainnets. Bitcoin had and still has a number of testnets that follow the naming scheme testnet1, testnet2 and so on. SegNet is another example of Bitcoin testnets that is created to test segregated witness proposal. Ethereum also has its own testnets. Ropstein, Kovan and Rinkeby are the names of some popular ones.

3.3 Block Explorers

Block explorers are tools used to both visualize and dig into the details of a blockchain. Explorers can be used to track individual blocks and transactions to see if they are confirmed. Also, they can be used to keep statistics and allow people to obtain insightful data derived from the network. Block explorers do not simulate a blockchain network by themselves; they are only used to gain insights into a blockchain constructed by the nodes of an already running system. They are also very useful tools for learning purposes.

3.4 Ganache

Ganache [13] is a software that allows users to create and operate their own Ethereum blockchain network. It is mainly used for creating suitable environments to test smart contracts on the blockchain. Ganache allows smart contract developers to test the chain by creating transactions, blocks and accounts.

Its feature set also includes:

- Block time configuration with its built-in user interface.
- Block explorer that allows checking blocks and transactions.
- Latest version of Ethereum.
- Log output for debugging that includes blockchain info and response data.

3.5 Hive

Hive [14] is an end-to-end test framework for Ethereum clients. It allows developers to facilitate black-box testing for their clients. In order to test clients under realistic conditions, Hive simulates a network of clients and allows testing of interactions between these clients. With Hive, users can create any number of tests in any language and validate their client's behavior under different circumstances by these tests.

3.6 Bitcoin Simulator

Bitcoin simulator [34] is an open source project that is "built to study how consensus parameters, network characteristics and protocol modifications affect the scalability, security and efficiency of Proof of Work powered blockchains" [20]. It is developed in C++ and built on top of a network simulator named ns-3 [33].

ns-3 is a very efficient and well established tool. The authors configured it with the measured network statistics of the blockchain mainnet. As a result, simulator was able to simulate both bandwidth and latency of the network very similar to the real world. Simulator includes implementations of common Bitcoin data structures, and supports different types of nodes like full node and miner. Each node can be configured with parameters such as connections and mining power. For their simulation, authors measured metrics of the mainnet such as block generation, block size distribution, total node count and their geographic distribution, then incorporated these findings into the simulator to simulate the whole blockchain network as realistic as possible.

Bitcoin Simulator is capable of simulating other proof of work based systems such as Litecoin [35] and Dogecoin [36], but does not support protocols that use different consensus mechanisms. It can be configured with wide variety of parameters [37] and includes some attack simulations out of the box. The simulator is command line tool, which accepts configuration parameters as arguments.

3.7 Simulators developed at TU Munich

Lack of simulators that are both efficient enough to be run on a single machine and extensible enough to support extensions to support different protocols led to the research of such simulators at Technical University of Munich. This ongoing research resulted in the following projects and produced invaluable research papers.

3.7.1 VIBES

VIBES is the first project that aims to offer an efficient and extensible simulator. It pawed the way for other simulators for testing different blockchains and also extensions to extend its scope. VIBES is implemented with the Akka framework in Scala. Actor model used in Akka allowed VIBES to simulate stateful Bitcoin node processes using a lightweight concurrency model.

Proof-of-work related tasks such as mining in Bitcoin is very computationally heavy and require too many resources. VIBES tackles this problem by offloading these computationally heavy tasks to a component called the Coordinator that can be seen in Figure 3.2. Coordinator acts as a scheduler and decides which simulated node mines the next block as well as which transactions to be propagated. Each node requires an approval from the Coordinator to work on a task. In other words, the Coordinator acts as a central authority to simulate a protocol that does not have a central authority to trust. Drawback of this architecture is that the Coordinator becomes the bottleneck for the whole simulation and prevents the software from utilizing all the CPU resources available on the hardware it is running on.

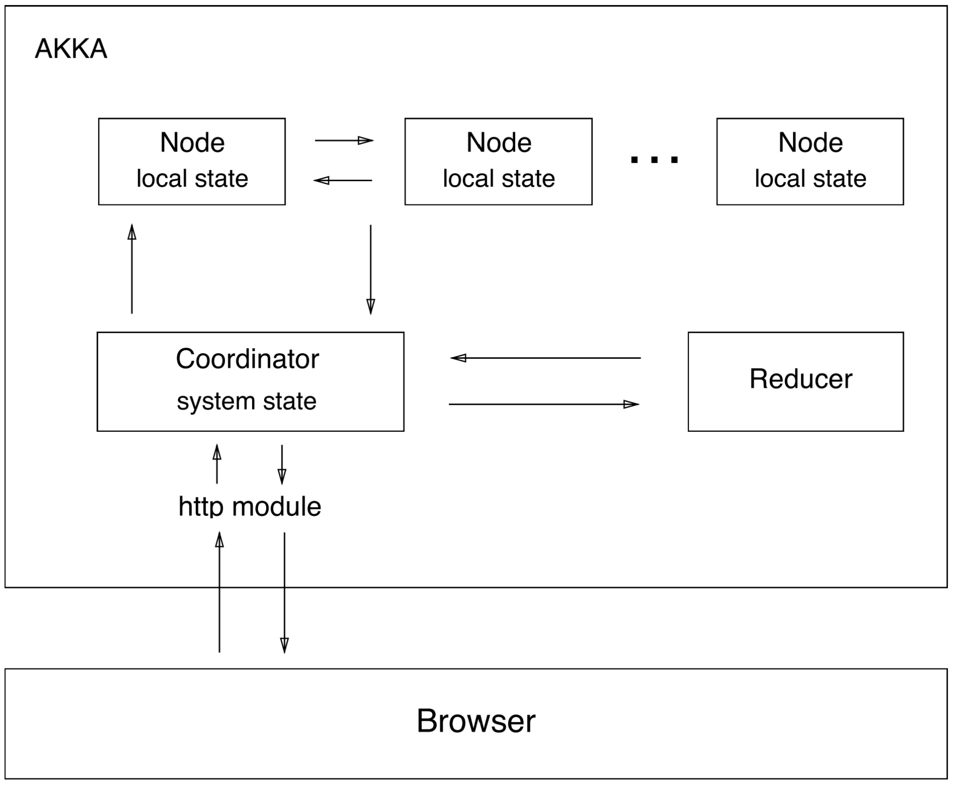


Figure 3.2: System architecture of VIBES [5].

VIBES offers configuration parameters for things such as block-time and transaction size. It significantly speeds up the network compared to the mainnet's block time. VIBES also falls short on some aspects. It is only capable of running generic simulations that does not resemble the real Bitcoin network. Moreover, it is not extensible as it is expected to be. Another drawback is the lack of updates during the simulation. Simulations in VIBES are only capable of notifying users once the simulation is finished. This design limitation prevents users from seeing what is happening under the hood while a simulation is running.

3.7.2 BBSS

BBSS is developed on top of VIBES and extends it with the aim of allowing more realistic simulations that resembles the real network. Moreover, it is implemented to make various attack simulations possible. It successfully runs attack simulations as well as some proposals such as segregated witnesses. Changes done with BBSS increased the extensibility of the VIBES to make further improvements possible. BBSS is affected by the design choices in VIBES and suffers from the same performance penalties as a result of the bottleneck created by the Coordinator. It is extensibility is also not as good as expected.

3.7.3 eVIBES

eVIBES is created to simulate the Ethereum network. It is inspired by VIBES and developed with similar goals. VIBES is developed to support further extensions such as the support for different blockchains. Even though it became more extensible after the changes done in BBSS, it is still deemed not extensible to support different blockchain protocols such as Ethereum by the author of eVIBES. The reason for this is VIBES' architecture is heavily shaped by the characteristics of the Bitcoin protocol and even though there are similarities between Bitcoin and Ethereum, they differ drastically on the following areas:

- Compared to Bitcoin, Ethereum relies on a different type of block verification protocol. This renders the block verification logic implement in the Coordinator component unnecessary.
- Coordinator in VIBES has more than one responsibility. It controls both the simulation and the chain direction. This coupling of functionality makes simulation of the sidechains that can be seen on Ethereum a very hard task without changing the most fundamental part of the simulator.
- Finally, the main goal of VIBES is to fast-forward the whole network to not be limited by the fixed block time in Bitcoin. Ethereum on the other hand uses another system that uses gas limits to decide when to mine a block. The difference as this big between the protocols renders one of the main responsibilities of the Coordinator unnecessary.

Because of these reason, eVIBES is not built on top of VIBES and implemented from scratch using the same language and the framework with a different architecture.

3.7.4 eVIBES Plasma

eVIBES Plasma is another framework that is created to simulate the Plasma framework on Ethereum blockchain. Plasma framework is a concept created to increase the scalability of the Ethereum blockchain by side chains. eVIBES claims to support side chain simulations but since both eVIBES and eVIBES Plasma developed on overlapping time-frames, they are both developed as green field projects and do not share any code. Unlike VIBES, BBSS and eVIBES, eVIBES Plasma is developed using a different framework and a language. It is developed in Kotlin and uses Vert.x, another popular framework that is inspired by the actor model.

eVIBES Plasma is unique among these simulators because of its ability to simulate both the main and side chains. Its architecture as shown in Figure 3.3 indicates what type of components needed when two different chains need to be simulated at the same time.

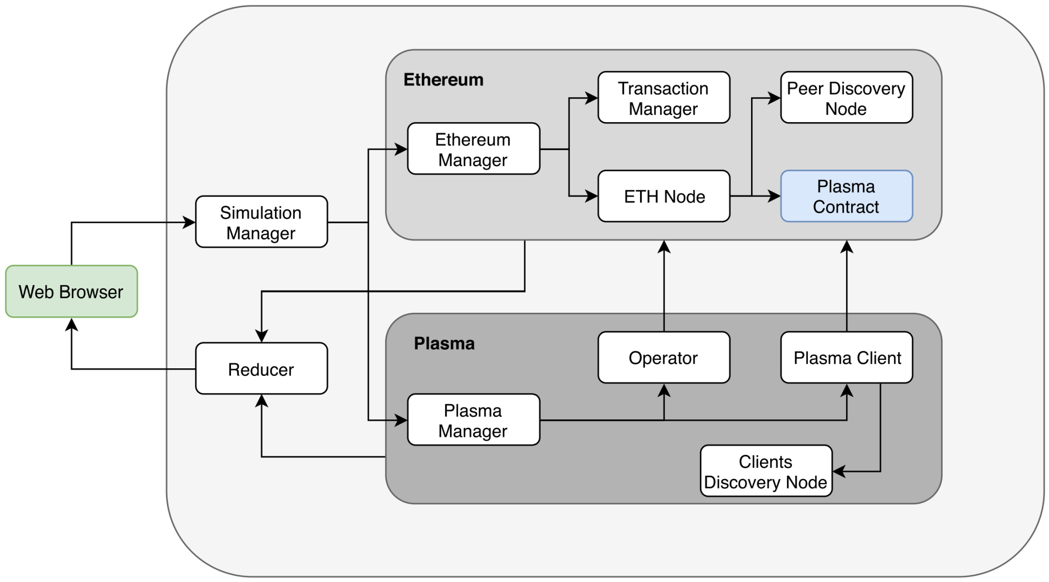


Figure 3.3: System architecture of eVIBES Plasma [6].

3.7.5 CIDDS

CIDDS is the first simulator that simulates a tangle instead of a blockchain and it is implemented in Python using Django framework. Initially it was planned as an extension to VIBES but due to it being a different distributed ledger technology it was then decided to be implemented as a green field project. IOTA simulations implemented with CIDDS do not require high concurrency compared to the blockchain simulators and the ledger is kept in a shared memory to be used by all the nodes as shown in the Figure 3.4.

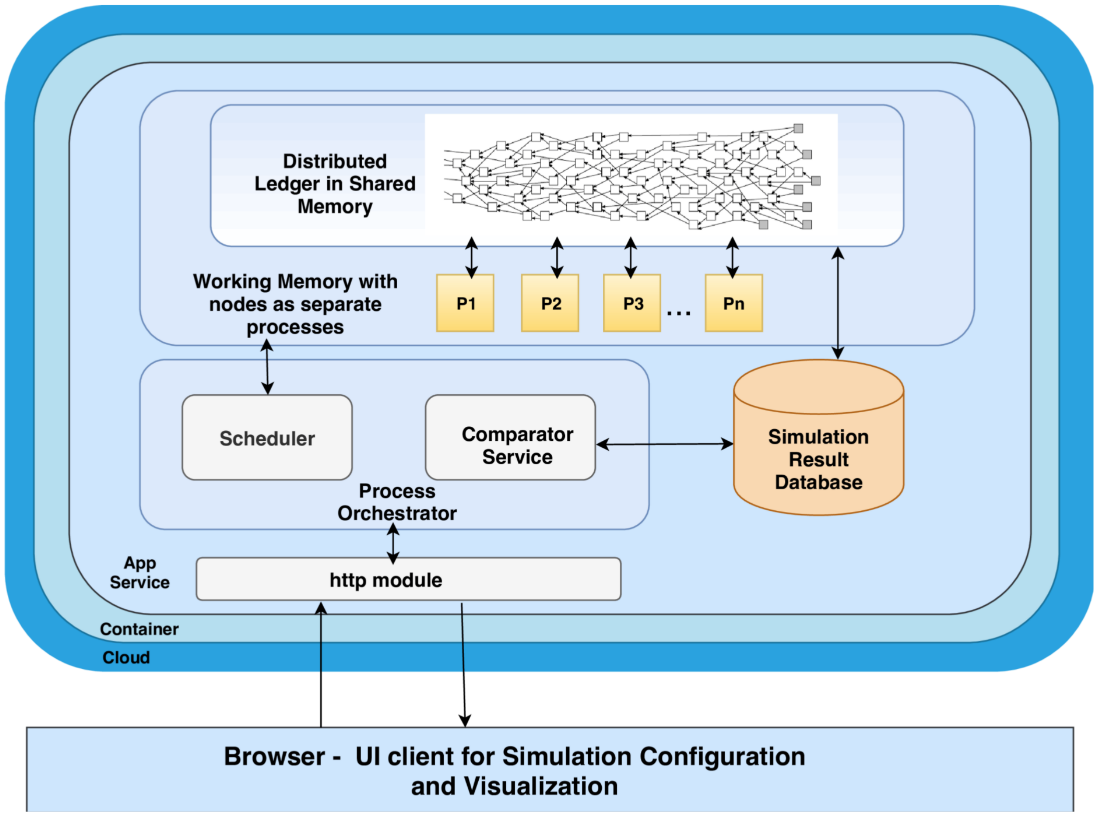

Distributed Ledger in Shared Memory
Working Memory with nodes as separate processes
P1
P2
P3
...
Pn
Scheduler
Comparator Service
Process Orchestrator
Simulation Result Database
App Service
http module
Container
Cloud
Browser - UI client for Simulation Configuration and Visualization

Figure 3.4: System architecture of CIDDS. [7]

3.7.6 Shortcomings and Limitations

Tools and simulators, besides the ones developed at TU Munich either serve a single purpose and not extendable, or require too many resources. This is the reason why the aforementioned efficient simulators are created in the first place. But, even the simulators developed at TU Munich have their fair share of shortcomings and limitations. Rest of this section refers to the simulators created at TU Munich as *simulators* and discusses their shortcomings.

Lack of true extensibility

First and the most important problem of these simulators are the lack of true extensibility. Both the drawbacks and improvements over the previous architectures are evaluated in detail on respective papers by their authors. Finding a single architecture to support all possible use cases is hard, especially when the system needs to be extended to support different consensus mechanisms. One can conclude that this is the reason why the support for multiple consensus algorithms was a future work in all simulator papers.

For instance, both BBSS and eVIBES affected by the limiting architecture of VIBES. BBSS extended VIBES to support Bitcoin-like blockchain protocols in addition to generic blockchain simulation, but it failed to utilize more than 20% of CPU because of the existing architecture. Another example that demonstrates this problem is eVIBES. It is implemented as a new system altogether because of the limitations of VIBES' architecture.

Lack of true scalability

Another recurring problem of these simulators is the consideration of scalability only on a single hardware instance. Because of the limitations of vertical scaling [38], this approach makes the hardware running the simulation a natural bottleneck. Even though achieving good scalability on a single instance should be one of the main goals of a simulator, it is not enough to run simulations on a large scale with thousands of nodes. In order to enable true large scale simulations, horizontal scaling [38] of the system should be a forethought instead of an afterthought.

Prioritizing the presentation layer

Final problem that can be observed in these simulators is too much focus on the presentation components. It is important to be able to easily modify configuration parameters and show meaningful data to the user while running the tests. Yet, presenting real-time data is an already solved problem with existing modern UI frameworks [39, 40] and therefore should not be one of the main focuses of a minimum viable product that is developed within a limited time and resources. Instead, a true extensible simulator should implement mechanisms that can enable these presentation layers as an extension.

Chapter 4

Approach

This chapter discusses the requirements, technology choices, design and implementation details of the distributed ledger simulation framework developed as part of this thesis. The framework consists of different modules with different feature sets and each module is discussed in detail under their respective sections. Finally, the architecture and implementation details of the two example simulations that are created using the framework are discussed.

4.1 Requirements

Coming up with a correct set of requirements is crucial for any software project. First step to build a framework such as DLSF is to identify the common requirements in distributed ledger simulations. We consulted to the previous research done on simulators at TU Munich to learn from their requirements and to identify points where they fall short. In addition to those, things we believe that is necessary but not addressed before such as true scaling are also included. The requirements of a distributed ledger simulation based on our research can grouped under these categories:

4.1.1 Extensibility and Modularity

Due to the distributed nature of the blockchain technology, every simulator needs nodes to implement the logic of the simulated protocol. Types and the number of these nodes varies depending on the protocol and the consensus mechanism. Simulators also need additional nodes just for the sake of simulations. Examples for such nodes would be the reducer and the orchestrator in VIBES or the operator and client discovery node in eVIBES Plasma.

Framework should be able to support the implementation of such nodes without imposing any design architecture.

4.1.2 Configurability

Framework should handle the distribution of arbitrary number of configuration parameters to all deployed nodes before each run. In addition to required parameters that would be used to configure the built-in functionality of the toolkit, wildcard parameters should also be supported to allow the configuration of user implemented custom nodes.

4.1.3 Automatic Deployment & Scaling

Framework should provide a mechanism for seamless deployments of both custom and built-in nodes with a simple software configuration. This configuration would be used to set up all the necessary number of nodes needed for simulation. Unlike previous simulators, framework should also handle the deployment of nodes on multiple hardware instances to allow clustered simulations.

4.1.4 Messaging Backbone

Messaging mechanisms are the backbone of every large-scale event-driven application. Framework needs to offer efficient publish/subscribe and request/response messaging among the nodes to support all possible simulation architectures. All the functionality that is provided by the framework should also be offered as an API accessible via this built-in messaging system.

Communication among the nodes in a large scale simulator is expected to be the biggest performance bottleneck. Therefore, the toolkit should offer a way to deliver messages in most efficient way. To achieve this, messages can be divided into two categories: inter-thread or local messages and cluster-wide messages. When the messaging parties are in the same instance, messages should be delivered using in-memory channels and when they are deployed on different instances, TCP or similar messaging protocols should be utilized.

4.1.5 Network Simulation

Since every blockchain protocol deals with distributed nodes, network simulation plays an important role to observe the behavior of the whole system under different network conditions. Framework should be able to simulate common network characteristics such as latency, package drops, connection drops and bandwidth when needed.

4.1.6 Being Approachable

In order to achieve higher adoption rates, the framework should be approachable and easy to use. The framework needs to be programmable in one of the most popular languages. In addition to the programming language of the underlying framework, APIs and concepts introduced should be easy to learn. Following these rules during the research for a candidate language and the framework would result in a project with a low learning curve.

4.1.7 Reporting

Framework should provide a mechanism to publish both intermediate and final results to its users. Contents of these reports can be unique to each simulator. The framework should also provide an API that abstracts underlying protocol and is capable of collecting real-time report messages and delivering it to subscribed clients via web sockets or over TCP connections. These abstractions would give a head start to developers so that they can focus on what to report instead of how to report them.

4.2 Technology Stack

Simulators and tools described in the earlier chapters use wide variety of languages, libraries and frameworks. Distributed ledger based technologies, as the name suggests, designed to be used by many nodes scattered across the globe with each node having its own state. In order to simulate such network efficiently, simulators need to be able to run many concurrent processes with isolated internal states while making use of all the available resources.

4.2.1 Possible Approaches

A traditional way to solve this problem would be to use multi-threading or multiple processes, and try to simulate states of nodes on a shared memory with locks. Even though a simulator designed with this approach would be capable of simulating distributed ledger network, it would not be the most efficient. A computer has a limited number of processes and threads and locking threads just to keep the shared state in order creates an unnecessary overhead. Creating and running a process for each simulated node would also be a waste of resources, because each running thread has its own memory overhead. Moreover, using synchronization and locks might lead to race conditions and hard to debug source code.

Another way to approach this problem is to use the actor model. In actor model, each actor has its own state and only communicates with the outside world via messages. Modeling simulated nodes as actors also works perfectly for our use case because nodes also have their own state and communicate with each other using messages. On a related note, advantages of using architectures based on the actor model are also mentioned in previous simulator research papers. Thus, many of the simulators used frameworks designed based on the actor model.

4.2.2 Framework and Language Selection

Combining the reasons mentioned above with the requirement of being approachable, research has been conducted on the available frameworks that use the actor model. Another deciding factor was the support of one of the popular programming languages listed in the Stack Overflow 2019 Developer survey [41]. The research resulted in two candidates: Akka [19] and Vert.x [42] because of the following features they offer:

- Both can handle a lot of concurrencies using small number of threads with great efficiency.
- Both come with scalable, actor-like deployment and concurrency model out of the box.
- Both provide a light-weight messaging system that is capable of propagating message in memory or over the network.
- Both support clustering without requiring significant changes in the code base.
- Both are modular and applications are written in both as sets of actors (or verticles in Vert.x's case) that communicate using message channels.

Based on my personal experience and my thesis advisor's recommendation, Java language and the Akka framework is chosen for the implementation.

4.3 Distributed Ledger Simulation Framework

Distributed Ledger Simulation Framework is the name of the project developed as part of this research to fulfill the previously listed requirements. Ultimate goal of the framework is to support development of simulators for any distributed technology. Unlike previous simulators where everything is designed to simulate a specific distributed ledger, DSLF is designed from ground up to be extendible and scalable without making any assumptions about the simulated distributed ledger technology.

In order to support wide variety of use cases, DLSF adopted a modular architecture. Each module is a maven sub-project and can be added as a dependency into any simulator project as needed. Having a modular design also allows offering dedicated modules for certain features while keeping the main module as unopinionated and lean as possible.

Currently, DLSF project consists of one main and two feature modules. **Core Module** creates the general runtime architecture. It is designed to take care of the common tasks so that developers can focus on their use cases. **Bitcoin Module** includes classes and components that can be used to accelerate development of Bitcoin simulators. The last module called **Boot Module** includes convenience components to simplify application bootstrapping process.

4.4 DLSF Core Module

DLSF Core module is the only mandatory module that needs to be added as a dependency by the simulation projects. It forms the foundation of any simulation project by providing all the necessary components, data structures and extension points. Its primary responsibilities in an application can be listed as follows:

1. Managing the whole life-cycle of a simulation from accepting the initiating HTTP request and parsing the configuration file to the deployment of necessary components and actors across the cluster.
2. Offering a web socket gateway component that acts as an abstraction layer that allows simulation components send real-time updates to the outside world.
3. Coordination of available workers in the cluster.
4. Storing and serving results of previous simulation runs.
5. Providing information about available simulations and worker states.
6. Parsing, validating and propagating simulation run configuration.
7. Instantiating and managing reducer and service instances during a simulation run.

The composition diagram shown in 4.1 illustrates the components that make up a simulation system implemented using DLSF. Each component shown is an Akka actor that only communicate via messages. Components colored in gray are provided by simulation developers and only created for individual simulation runs by the system. They are provided to the system via simulation templates described in Section 4.4.4. Each component has a dedicated section describing their characteristics in detail below.

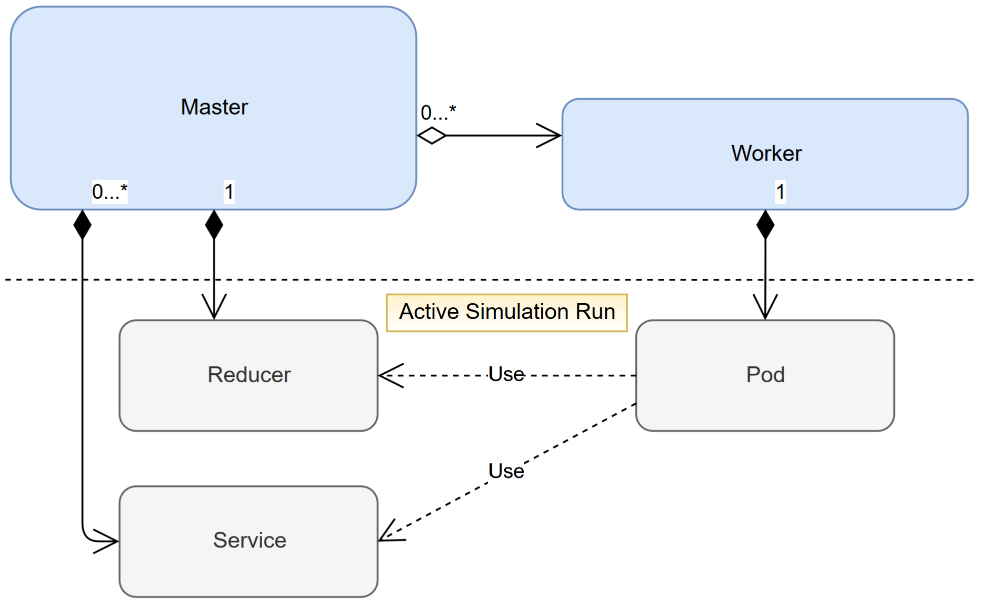
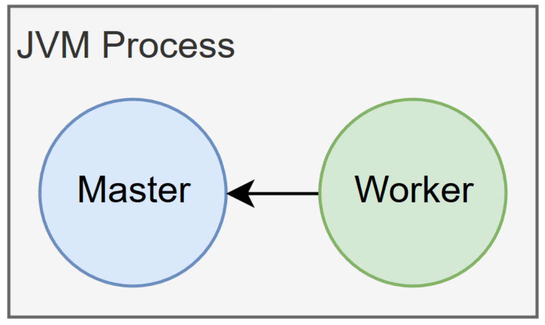


Figure 4.1: DLSF Core composition diagram. Components in gray are extension points that are provided by individual simulation templates.

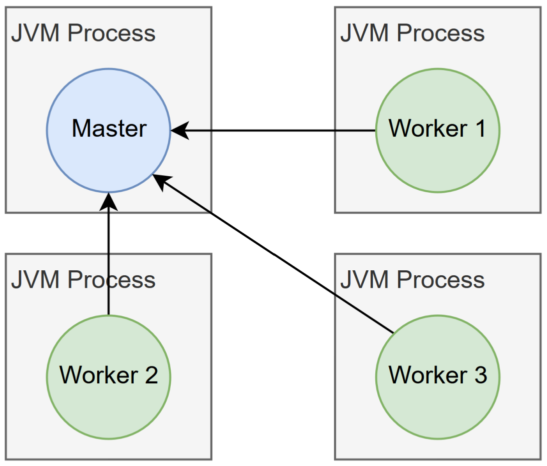
4.4.1 Architecture Modes

DLSF Core allows simulation systems to run in two architecture modes. Akka make this possible with its powerful messaging and actor discovery abstractions. Clustered mode is enabled by compiling the project with a configuration file similar to Akka's example cluster configuration [43]. Besides this compile time configuration, simulations do not require a single line of code change to be able to run on clusters.

Standalone simulation system mode that runs on a single JVM process is shown in Figure 4.2a. This mode is useful and the most efficient when simulations are run on a single computer because both master and worker run on the same JVM process. Cluster simulation system mode that runs on multiple JVM processes is shown in Figure 4.2b. This mode can be used to utilize server clusters where each JVM process communicates via TCP messages.



(a) Standalone mode simulation system that runs on the same JVM process.



(b) Clustered mode simulation system that runs on multiple JVM processes.

Figure 4.2: Comparison of simulation system architecture modes.

4.4.2 Master

Master component is the most complicated and feature rich component of the framework. It has multiple duties and communicates with other components via different protocols. For instance, it acts as a web socket gateway to simulation reducers and services while acting as a coordinator for the workers. The whole system is capable of running one simulation at a time. For this reason, master makes sure all workers are idle before it accepts a new simulation run creation request. Association between the master and other components is shown in Figure 4.3.

HTTP Endpoints

Master component creates a number of HTTP endpoints. These endpoints can be used to manage the system and to access its state. These endpoints are implemented in a way that all the possible use cases expected from a simulator are covered regardless of the specifics of simulations. Upon receiving a request, master takes necessary actions on behalf of the user. Endpoints and their descriptions are listed below. Commonly used data transfer objects are also depicted in the Figure 4.4.

- **GET /workers** - Show all available workers and their state. Response body is a WorkerDto array.
- **GET /simulations** - Show all available simulations. Response body is SimulationDto array.
- **GET /active-run** - Get details of the active simulation run. Response body is a

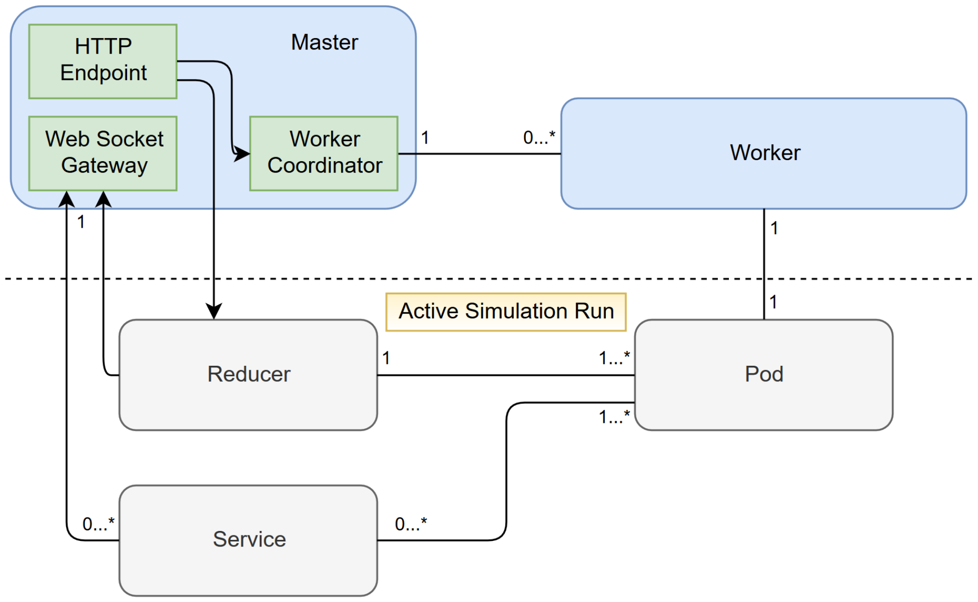


Figure 4.3: DLSF Association Diagram. Components in gray are extension points that are provided by individual simulations.

SimulationRunDto.

- **PUT /active-run** - Create a new active simulation run. Starts a new simulation run and accepts simulation specific configuration in the payload. Request body is CreateSimulationDto.
- **DELETE /active-run** - Stop the active simulation run. Allows manually terminating a simulation run.
- **GET /active-run/updates** - This URI is for subscribing active simulation run updates. Any client connecting to this endpoint is expected to upgrade the connection to the web socket protocol.
- **GET /runs** - Get a list of all previous simulation run details. Response body is a SimulationRunDto array.
- **GET /runs/:runId** - Get a simulation run detail. Request body is a SimulationRunDto.
- **GET /runs/:runId/results** - Get a simulation run result. Result payload is directly fetched from simulation's reducer and can be in any form.
- **DELETE /runs/:runId** - Delete a simulation run record with its results permanently.

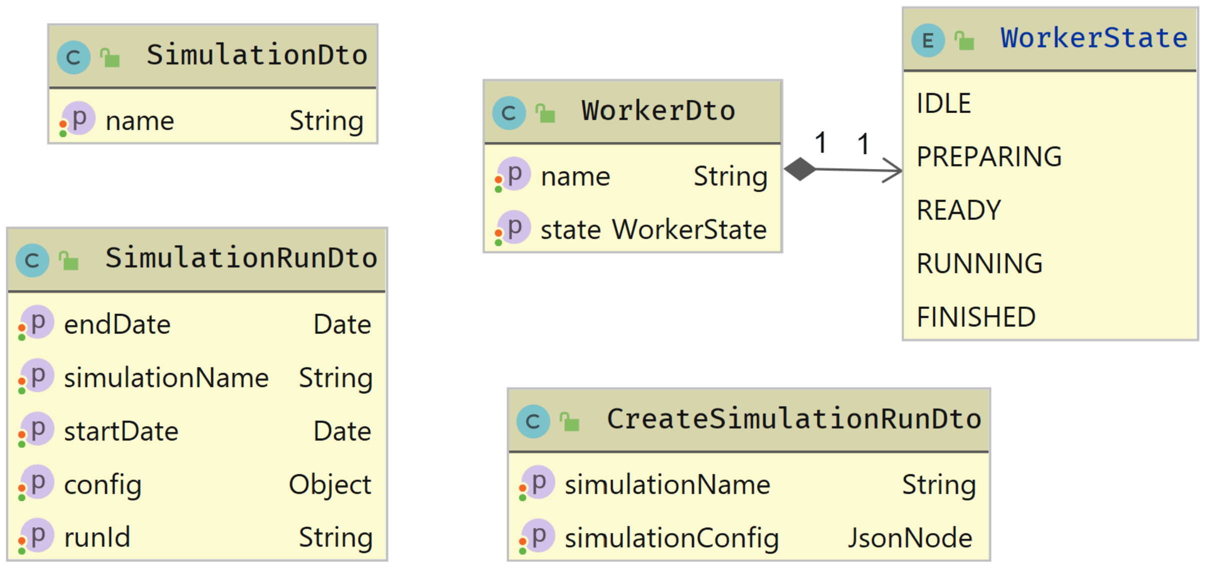


Figure 4.4: DLSF data transfer object class diagrams.

Web Socket Gateway

Web socket gateway implementation in master component accepts web socket connections when there is an active simulation run within the system and at the end of each run, it closes all the connections. It allows clients to subscribe real time updates of the active simulation run. It automatically manages the pool and the lifecycle of connected sockets and is not limited to a single client connection. During simulation runs, components within the system also get a hold of a reference of the gateway protocol.

Having a gateway allows components to send updates any time and in any form during a simulation run. Simulation developers can use this actor's reference within their components to publish updates depending on their simulation scenario. Gateway saves simulator developers from the burden of the tasks such as abstracting the complexity of managing web socket client connection lifecycles and serialization. Simulation developers do not even need to learn how Akka's web socket server implementation works because details such as these are already handled by the framework.

Coordinating Workers

Simulation systems developed with DLSF are capable of running simulations on a cluster and each cluster runs a worker component described in Section 4.4.3. Master component is responsible for coordinating these workers. It tracks workers that join and leave the cluster, tracks their state both during simulation runs and when they are idle, and coordinates simulation runs across the cluster.

Simulation Run Results

Master component is responsible for managing previous simulation run results and allows users to access or delete them when needed via its RESTful API. In a simulation developed by DLSF, the Reducer is responsible for providing the result. Their lifetime exceeds the lifetime of the simulation. Once a simulation completes, every component and resource allocated for the run deleted except the Reducer. This Reducer then used whenever a result is requested for this specific simulation run. Users can delete these Reducers when they no longer need them or when they want to free up some resources. Master manages all these functionalities automatically for simulation developers.

4.4.3 Worker

Workers are responsible for creating and running Pods. These components discover and register themselves to the master when instantiated and afterwards they will be available for running simulation pods. Workers can exist without a master, but they would be constantly seeking a master and would do nothing until they register to one.

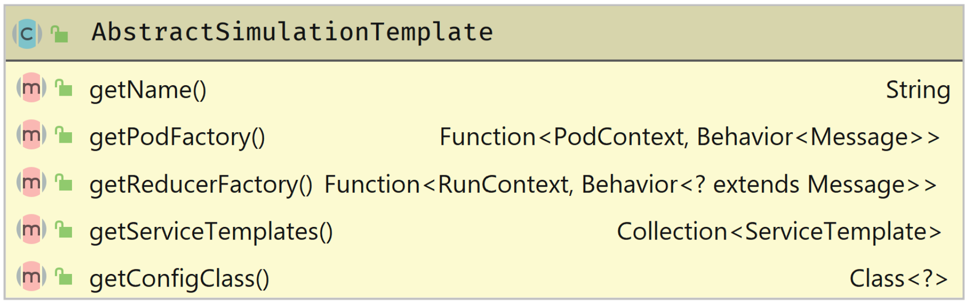
Master coordinates all workers by commanding them to prepare, start or terminate simulations. They communicate their state to the master to let it know when they are idle or when they are prepared to run the current simulation. Each worker creates a single simulation pod when master orders them to prepare for a simulation. They instantiate pods by providing them the run context which includes the configuration object, service and reducer references.

4.4.4 Simulation Template

Every simulation in a DLSF system consists of a reducer, services and pods. These components are defined by extending the AbstractSimulationTemplate class offered by the framework. Simulation templates need to be registered to both master and the worker instances during initialization by simulation developers, but the framework also offers a module that can be used to automate this task.

AbstractSimulationTemplate is an abstract class that serves as an extension point for the framework. Simulations are registered to the system using this template, and then used by master and worker components. Template class diagram is shown in Figure 4.5.

Methods that need to be implemented by simulations are as follows:



AbstractSimulationTemplate	
getName()	String
getPodFactory()	Function<PodContext, Behavior<Message>>
getReducerFactory()	Function<RunContext, Behavior<? extends Message>>
getServiceTemplates()	Collection<ServiceTemplate>
getConfigClass()	Class<?>

Figure 4.5: DLSF AbstractSimulationTemplateDiagram class diagram.

- **getName()** - Returns the unique name of the simulation implemented in this template. This name is used throughout the system to address this simulation.
- **getConfigClass()** - Returns the Java class that will be used to deserialize the simulation configuration received over the network via the HTTP endpoint. Master takes care of validating and converting received JSON object into a Java class instance specified here.
- **getReducerFactory()** - Returns a factory function to create a reducer behavior.
- **getPodFactory()** - Returns a factory function to create pods.
- **getServiceTemplates()** - Returns a collection of service templates. A service template is a simple class that consists of a factory and a service key to be used for registering and discovering service actors.

4.4.5 Pod

Pods are simulation specific components created and managed by workers. As mentioned earlier, there can be multiple workers running multiple instances of a pod in a simulation. A pod is provided all the necessary parameters on initialization by its worker and expected to signal when its setup is complete. Later on it communicates with its worker to signal when the simulation is complete on its instance. Worker can also signal its pod to stop when simulation is terminated manually by user.

4.4.6 Service

Services are actors that are managed by the master and can be used to offer additional functionalities to running pods and to each other. They can be thought of as singletons in a typical application. Only one instance is created per service template, and they are made available to all other components via references. DLSF does not know the implementation details or protocols of these services and only needs to know how to create and register them during an active simulation run. It is up to the simulation developers to design their components best suited to their scenarios.

4.4.7 Reducer

Reducer shares the same characteristics with services with slight differences. Framework expects reducers to be responsible for delivering results whenever asked by the users. For this reason their protocol is expected to include certain message structures to ask and return simulation results. This is enforced by the Java type system. Framework does not interfere with other message types in reducers' protocol, therefore their messaging protocol can be further extended to support communication with other simulation components.

Once a simulation is finished, reducer actor is kept alive by master until results are deleted manually by users. This allows the system to deliver results to users without enforcing any architectural design. Reducers only need to respond to master's requests for results and therefore can be implemented to keep results in memory, in a file or in a database.

Each simulation reducer is required to extend the BaseReducerProtocol. Class diagram of the protocol is shown in Figure 4.6. SimulationEnded message notifies the Reducer when simulation run ends. ResultRequest message is sent when a simulation run result is requested by users using the HTTP endpoint. Once received, ResultRequest expects a reply to the master with the payload of simulation results. The base protocol class ensures that simulation specific reducers can still respond to master's requests, regardless off their implementation details.

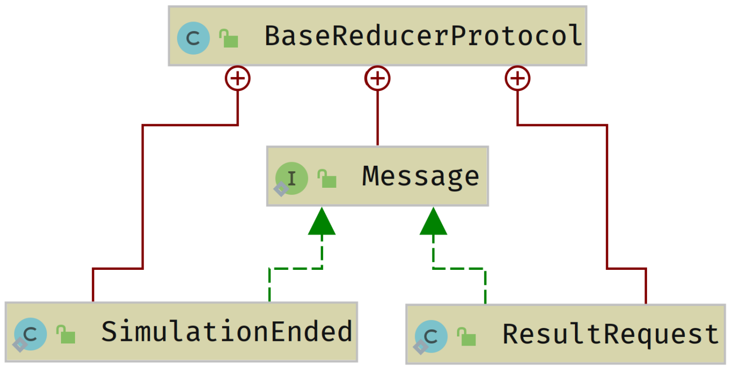


Figure 4.6: DLSF Core package BaseReducerProtocol class diagram.

4.5 DLSF Bitcoin Module

DLSF Bitcoin module contains all the data structures and helper libraries that could be used in Bitcoin simulations. Currently, it contains 3 top-level packages: **data**, **network** and **node**.

4.5.1 Data Package

Data package includes classes that resemble the main data structures in Bitcoin protocol. These classes are also extensively used by other packages in this module and provide interoperability between the module and user developed simulations. Users are encouraged to use these classes in their Bitcoin simulations.

These 4 classes include all the fields specified in Bitcoin protocol with very minor differences. Since the framework ignores the cryptography related parts of the protocol in order to offer better performance, hashes are directly included as a field so that they don't need to be generated by each node. Another difference is the lack of transaction script related fields. DLSF does not support the Bitcoin Script language and transactions can be considered legitimate during simulation. TxIn and TxOut data classes do not include such fields and TxOut directly uses the recipient field.

Diagrams of the main data structures: Block, Tx, TxIn and TxOut are listed in Figure 4.7.

4.5.2 Network Package

Bitcoin's network topology is unstructured and its nodes follow small set of rules while initiating connections. Because of these characteristics, a network coordinator component with enough configurability can cover most of the simulation use cases. Network package makes use of this fact and can be used to accelerate the creation of network topologies similar to that of Bitcoin with less code.

The package provides an Akka actor named BitcoinNetworkCoordinator that aims to take advantage of these characteristics. BitcoinNetworkCoordinator has the following features:

1. It allows the configuration of nodes' number of inbound and outbound connections.
2. It allows nodes to discover other nodes with a single message. Neighbors are distributed automatically in a way that would create a network similar to the Bitcoin network.

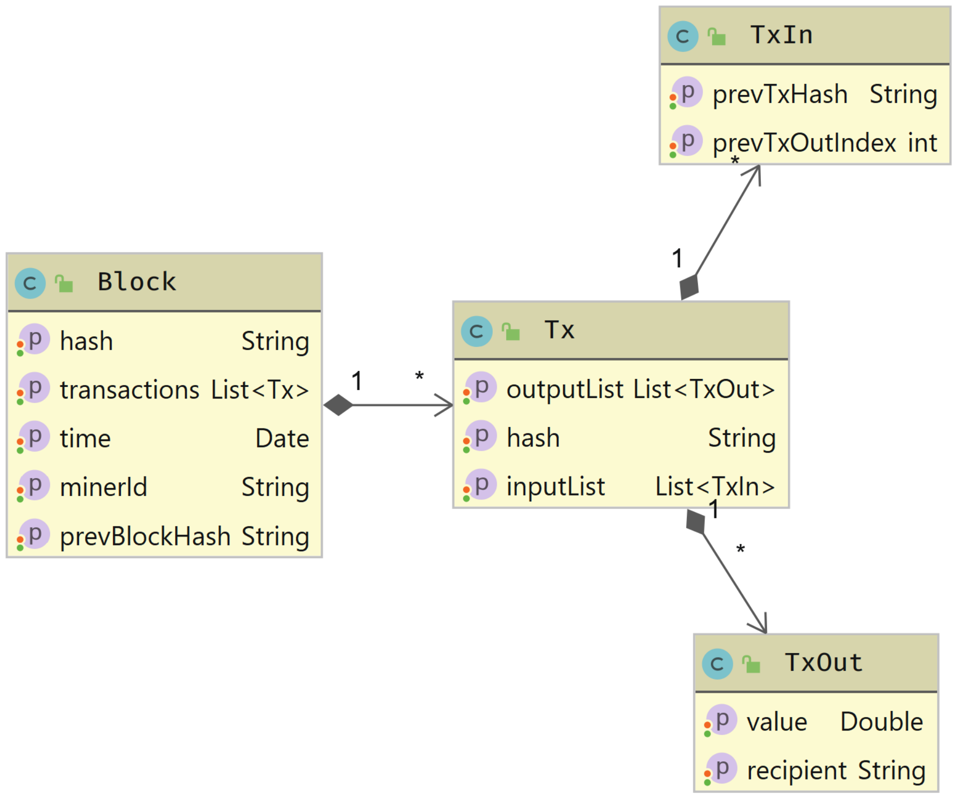


Figure 4.7: DLSF Data package class diagrams.

3. It is able to create random and reproduceable network topologies so that simulations could be run in identical topologies with different parameters when needed.
4. It provides the data representation of created topologies so that they can be used for monitoring and presentation purposes.

Configuration

BitcoinNetworkCoordinator is configured with the following parameters:

- **seed** - Seed is an optional parameter that allows reproduceable topologies. Topologies created with the same seed are always similar. It can be set to 0 to indicate the topology should be completely random.
- **totalNumOfNodes** - Specifies the total number of nodes in the network that are expected use the coordinator service to discover connections.
- **maxOutboundConnections** - Specifies the maximum number of connections initiated by a node.

- **maxInboundConnections** - Specifies the maximum number of connections should a node accept other than the ones initiated by itself.

Neighbor Discovery

BitcoinNetworkCoordinator accepts registration messages from nodes that want to discover peers in the network. Once all the nodes register themselves to the coordinator, it creates a network topology based on its configuration and notifies all registered nodes with a message that includes inbound and outbound connection addresses. From then on, registered nodes can directly communicate with their connections.

Getting Network Topology

BitcoinNetworkCoordinator creates random topologies, but sometimes it can be useful to access the created topology. Because of this reason, BitcoinNetworkCoordinator accepts network topology requests and provides the data representation the network topology as a response. Simulation developers can fetch this data from simulation reducer or any other actor and use it for reporting or debugging purposes.

4.5.3 Node Package

The Node Package includes all the data structures needed to simulate the internal state of a full node. Implementations of offered classes are heavily inspired by Bitcoin Core and follow the protocol rules as close as possible. A full node uses many different sub-components to manage its internal state. Features and implementation details of the classes corresponding to these sub-components are described in the sections below.

BlockTree

This component is used to keep track of the whole tree of blocks that includes the confirmed, stale and orphan blocks. It can be instantiated by providing the hash of the genesis node. Best block chain is referred to as main branch in the implementation of this class. In terms of functionality, it offers two types of methods: one for getting the main branch and another for adding blocks. Methods and properties of BlockTree class can be seen in Figure 4.8.

Getting the main branch is useful for miners because new blocks are mined on top of main branch's highest block. It is also useful for nodes to decide which transactions are part of the main branch, or in other words confirmed.

Discovering a new block can result in 3 different scenarios:

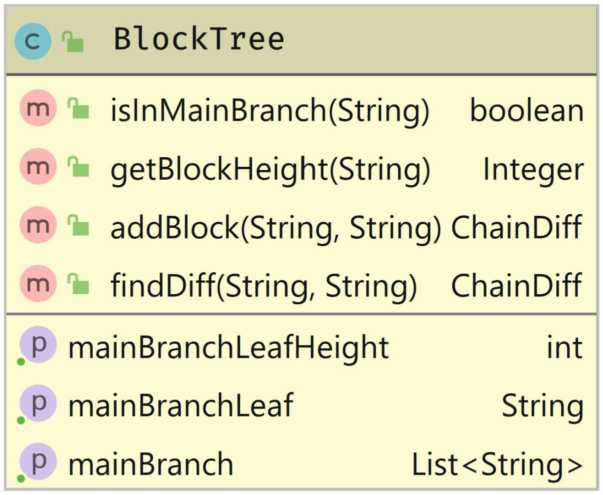

BlockTree
isInMainBranch(String) boolean
getBlockHeight(String) Integer
addBlock(String, String) ChainDiff
findDiff(String, String) ChainDiff
mainBranchLeafHeight int
mainBranchLeaf String
mainBranch List<String>

Figure 4.8: DLSF Data package BlockTree class diagram.

1. If block is built on top of a block that is not known yet, the main chain stays the same and the block is added to the orphan blocks.
2. If block is built on top of the highest block on the main chain, it extends the main chain. There might be orphan blocks that are built on top of this newly added block, in this case those orphan blocks are also added to the chain and the main chain would be extended by more than one block.
3. Similar to blocks added to the main chain, blocks that are added to side chains can also un-orphan some previously orphan blocks. Depending on the new length of the chain:
   (a) If the side chain still shorter than or the same length as the main chain, main chain remains unchanged.
   (b) If the side chain is now longer than the main chain, it becomes the new main chain and triggers a chain reorganization.

To handle all of these scenarios, adding a block to the BlockTree returns a data structure called **ChainDiff** which includes added and removed blocks to the main branch in order. API consumers then decide and act upon the changes happened on the main chain and adjust their internal state.

ChainState

ChainState is a data structure that keeps track of the all confirmed transactions and UTXOs on the best block chain. It shares the name of the Bitcoin Core component that offers same the functionality. This component provides methods that can be used for the following purposes:

- Checking if a transaction with hash is confirmed.
- Getting details of a transaction.
- Finding transaction output instances that are used as an input.
- Checking if a transaction output is already spent.
- Calculating the total fee of a transaction.

FullNode automatically manages and updates this component whenever a block is added to the best block chain or whenever blocks are removed from the best block chain in the event of chain reorganization. ChainState class diagram is shown in Figure 4.9.

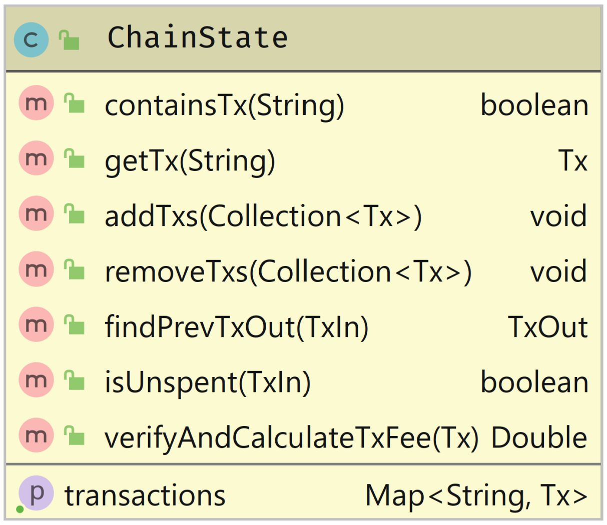

ChainState
containsTx(String) boolean
getTx(String) Tx
addTxs(Collection<Tx>) void
removeTxs(Collection<Tx>) void
findPrevTxOut(TxIn) TxOut
isUnspent(TxIn) boolean
verifyAndCalculateTxFee(Tx) Double
transactions Map<String, Tx>

Figure 4.9: DLSF Data package ChainState class diagram.

MemPool

MemPool is also named after a similar component found in Bitcoin Core. Its main purpose is to keep transactions issued by nodes in the network. Unlike blocks, transactions can be freely issued by any node at any time and because of this reason memory pool component is updated very frequently. Miners use this pool to get transactions that are going to be included in their candidate blocks.

MemPool implementation offers methods to manage verified and orphan transactions. Moreover, it also offers methods to query transactions by any of their used transaction inputs. All of these methods are extensively used by the FullNode, but they can also be used by simulator developers when needed. Complete diagram of the MemPool is shown in Figure 4.10.

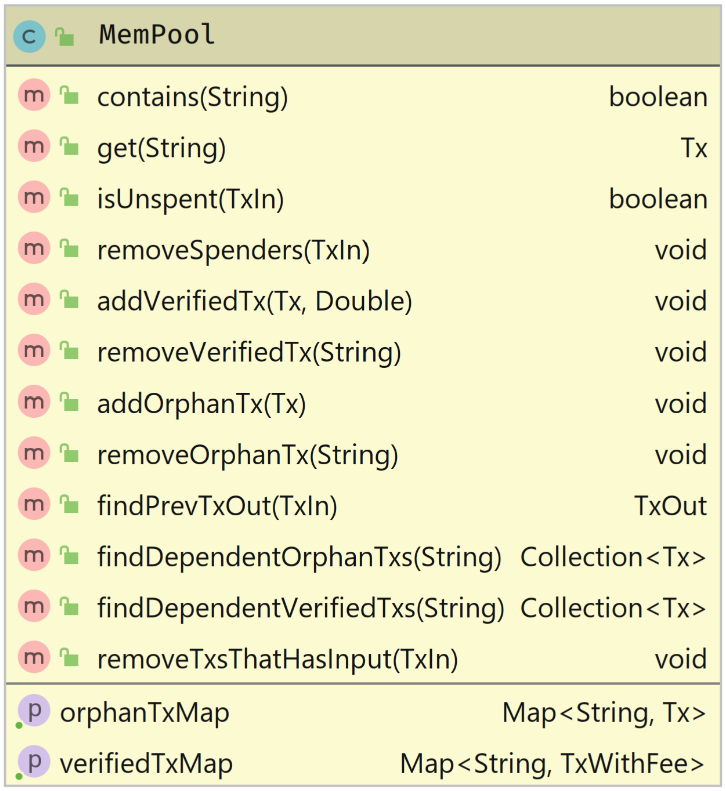

MemPool
contains(String) boolean
get(String) Tx
isUnspent(TxIn) boolean
removeSpenders(TxIn) void
addVerifiedTx(Tx, Double) void
removeVerifiedTx(String) void
addOrphanTx(Tx) void
removeOrphanTx(String) void
findPrevTxOut(TxIn) TxOut
findDependentOrphanTxs(String) Collection<Tx>
findDependentVerifiedTxs(String) Collection<Tx>
removeTxsThatHasInput(TxIn) void
orphanTxMap Map<String, Tx>
verifiedTxMap Map<String, TxWithFee>

Figure 4.10: DLSF Data package MemPool class diagram.

Wallet

Wallet is a very simple data structure. Its intended use is to keep all available UTXOs for an address. These unspent transactions kept as candidate TxIns to be used in future transactions. Wallet class diagram is depicted in Figure 4.11.

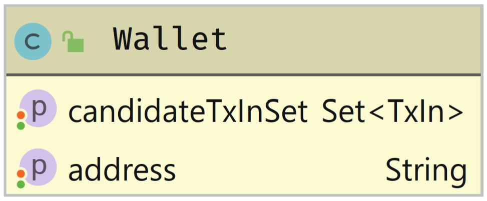


Figure 4.11: DLSF Data package Wallet class diagram.

FullNode

FullNode is the top-level class that resembles a full node in Bitcoin network and it manages its own state with help from all the sub-components introduced earlier. It also offers higher level convenience methods to digest newly discovered blocks and transactions.

Implementing the Bitcoin protocol from scratch based on the protocol rules and the reference implementation is a tedious task and FullNode is created to address this problem. FullNode can be thought as a realistic Java implementation of Bitcoin Core without the networking, mining and cryptographic parts. Simulation developers only need to provide blocks and transactions as they are received from the simulated network, and FullNode instance takes care of validation, verification and updating the state of every sub-component.

FullNode is instantiated with the following two parameters:

- **genesisBlock** - Genesis block is used as a starting point for the chain. This block is also hard-coded in Bitcoin Core implementation. Being able to specify the genesis block instead of hard coding it allows simulation developer to start their simulations with existing funds.
- **walletAddress** - Wallet address specifies the wallet address that will be tracked by this node. This allows each FullNode to track UTXOs of a single address very efficiently. Developers are free to ignore this parameter if they want to implement more sophisticated UTXO tracking logic.

FullNode allows access to all of its sub-components so that simulation developers can use them according to their needs. In addition to these, it has two public methods that offers higher level functionality to simulation developers. These methods first validates provided parameters and then updates all the sub-components according to the Bitcoin protocol. Class diagram of FullNode is depicted in Figure 4.12.

receiveBlock method can be used to add new blocks to the blockchain and returns a Boolean indicating whether the provided block should be propagated to the neighbors. Algorithm followed by this method is depicted in the flowchart in Figure 4.13.

receiveTx method can be used to validate, verify and add transactions to the **MemPool**. This method throws exceptions if the provided cannot be validated or rejected. Algorithm followed by this method is depicted in the flowchart in Figure 4.14.

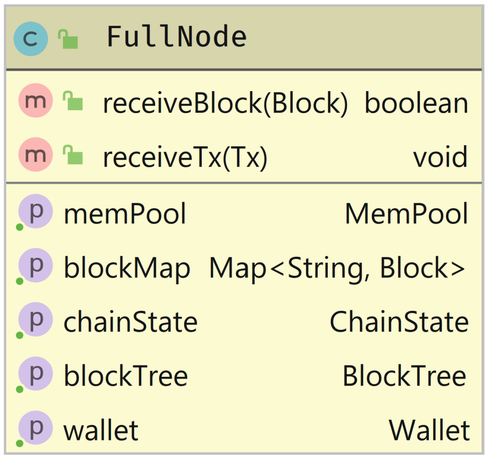


Figure 4.12: DLSF Data package FullNode class diagram.

4.6 DLSF Boot Module

Simulation systems created with DLSF can be really complex. They could include many simulations, or they can be run on a cluster which consist of multiple servers. This complexity can make assembling and running the application a tedious task. For instance, master node needs to know all the available simulation templates so that it can offer them as an option via its REST API and validate simulation configurations before initiating the run. Simulators need to know all the available simulation templates so that they can create simulation instances and manage their lifecycle when a simulation run triggered. Moreover, readable node names can be given to each component so that they can introduce themselves to the master node with meaningful names.

DLSF Boot module is created just to address this problem. It offers classes and a main method with arguments in order to make the start-up of the system more convenient. Start method is the only static method provided by the Boot class and it accepts two optional arguments:

- **Node Type** - Node type specifies whether a master or a worker instance should be created. Users can also choose to instantiate both to run non-clustered, standalone simulations.
- **Node Name** - Node name specifies the name to be used to register the node to the Akka system.

On application start, boot module searches the whole class-path using Java Reflection API to find all the simulation templates automatically. Then it creates desired instances with given names and provides found simulation templates to these components. Simulation

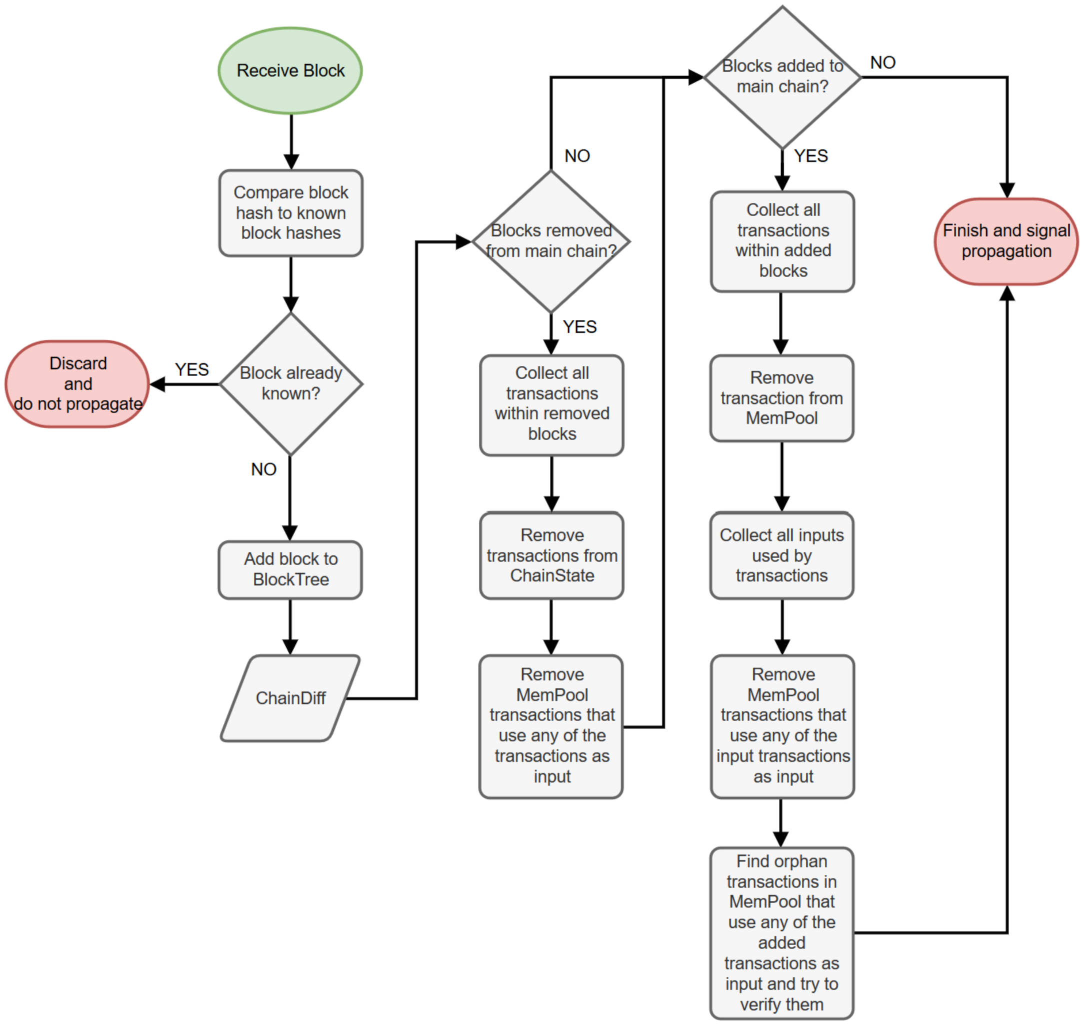

Receive Block
Compare block hash to known block hashes
Block already known?
YES
Discard and do not propagate
NO
Add block to BlockTree
ChainDiff
Blocks removed from main chain?
NO
YES
Collect all transactions within removed blocks
Remove transactions from ChainState
Remove MemPool transactions that use any of the transactions as input
Blocks added to main chain?
NO
YES
Collect all transactions within added blocks
Remove transaction from MemPool
Collect all inputs used by transactions
Remove MemPool transactions that use any of the input transactions as input
Find orphan transactions in MemPool that use any of the added transactions as input and try to verify them
Finish and signal propagation

Figure 4.13: FullNode#receiveBlock method algorithm flowchart.

developers can create as many templates as they want in their modules and boot their system by a single method call as shown in Listing 4.1.

```
public static void main(String[] args) {
    Boot.start(NodeType.STANDALONE, "master+worker");
}
```

Listing 4.1: Booting a standalone simulation system with DLSF Boot module.

Boot module also offers a main method for convenience. Users can utilize this by creating executable jars and providing Boot as a main class. Node type and the name can be provided to the program via **–type** and **–name** parameters.

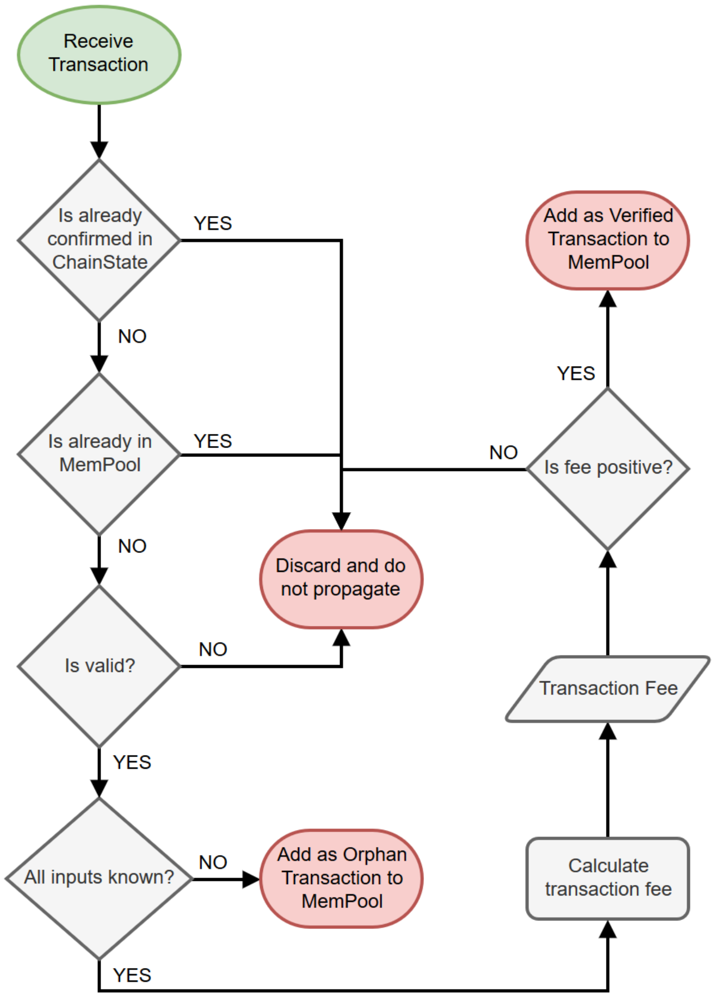

Receive Transaction
Is already confirmed in ChainState
YES
NO
Is already in MemPool
YES
NO
Is valid?
NO
YES
All inputs known?
NO
YES
Add as Orphan Transaction to MemPool
Discard and do not propagate
Calculate transaction fee
Transaction Fee
Is fee positive?
NO
YES
Add as Verified Transaction to MemPool

Figure 4.14: FullNode#receiveTx method algorithm flowchart.

4.7 Example Simulations

In addition to the development of Distributed Ledger Simulation Framework, two blockchain simulations are created to asses both the developer experience and the usefulness of the framework. Each simulation is discussed in their respective sections in detail. Furthermore, a web client is also developed to showcase the capabilities of these simulations better.

4.8 Blockchain Simulation System Web Client

Simulations developed with DLSF provide all the functionality via HTTP endpoints and web sockets. This architectural design decouples the simulations from any client side implementation. Users and developers could interact with the system using web clients, mobile applications and create even more complex simulation systems by programmatically interacting with the endpoints.

To be able to demonstrate these abilities, a web application is developed in TypeScript using the Angular framework. This client is capable of running and managing both of the example simulations. Even though it can be extended further in the future, it is recommended to create dedicated clients for different simulations. This would give simulation developers even more freedom and a space to demonstrate their simulations' capabilities without forcing them to use any client-side framework or architecture.

Web client consists of 4 main views: Dashboard, New Simulation Run, Active Simulation Run and Simulation Run List. Users can navigate these views using the side navigation menu shown in Figure 4.15.

4.8.1 Dashboard View

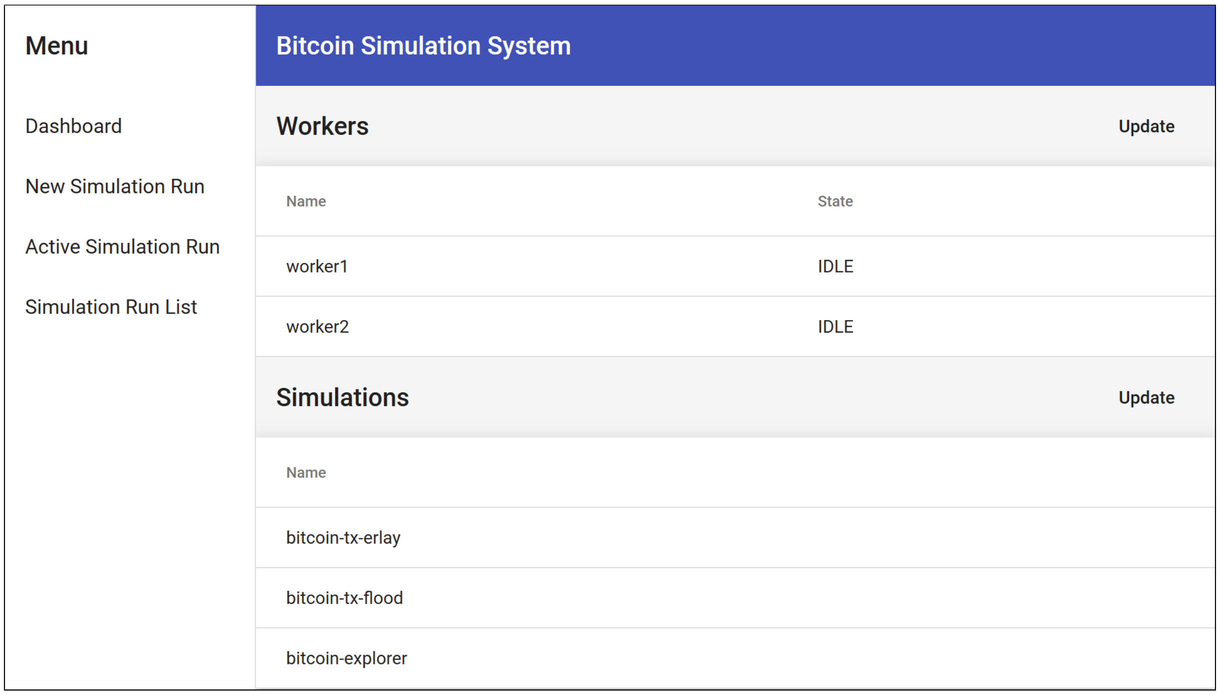
This view is the default page shown on the index page. It allows users to see the general state of the system. Workers section on the top shows all the workers registered to the system as well as their current state. Simulations section at the bottom shows which simulations are available in the system. There is also update buttons that can be used to update the information on the screen without reloading the page.

4.8.2 New Simulation Run View

New Simulation Run view allows users to run new simulations with specified configuration. This view only provides JSON configuration templates with defaults for each simulation and does not contain client-side validation or input fields. Since web client is only implemented to demonstrate the capabilities of the framework and simulations, not much effort is spent on the examples' end user experience. Users can pick the templates offered on top right and modify them as they would like. Clicking run button at the bottom sends a request to the server and initiates a simulation run. Upon receiving successful creation response, users are automatically navigated to the Active Simulation Run view to follow the real time updates of the run they just created.

4.8.3 Active Simulation Run View

Active Simulation Run view shows real-time statistics about the currently running simulation. This view subscribes to the framework's web socket gateway and directly



Workers

Name	State
worker1	IDLE
worker2	IDLE

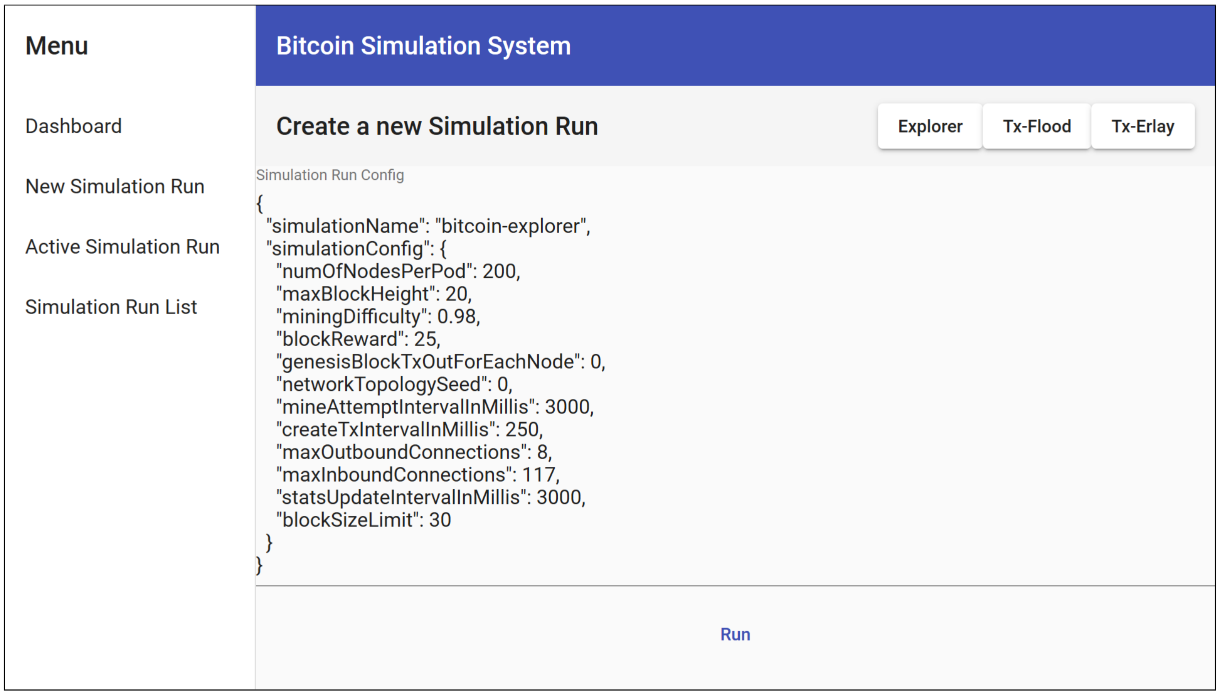
Simulations

Name
bitcoin-tx-erlay
bitcoin-tx-flood
bitcoin-explorer

Figure 4.15: Web client's Dashboard view.

receives updates from components of the active simulation run without needing to poll any endpoint. When the active run finishes, users are navigated to the simulation run result page automatically. Fields shown in this view depicted in Figure 4.17 and descriptions of each field are as follows:

- **Tx per Second** - Confirmed transactions per second.
- **Blocks per Second** - Confirmed blocks per second.
- **Blocks in Main Chain** - Number of blocks in the best block chain, excluding the genesis block. In other words, this is the current length of the highest block chain.
- **Created Blocks** - Number of created valid blocks. This includes blocks in both the main and orphan chains.
- **Txs In Main Chain** - Number of confirmed transactions in the best block chain.
- **Txs In MemPool** - Number of verified and orphan transactions in the transaction pool that are waiting to be confirmed in the next block.



```
{
 "simulationName": "bitcoin-explorer",
 "simulationConfig": {
  "numOfNodesPerPod": 200,
  "maxBlockHeight": 20,
  "miningDifficulty": 0.98,
  "blockReward": 25,
  "genesisBlockTxOutForEachNode": 0,
  "networkTopologySeed": 0,
  "mineAttemptIntervalInMillis": 3000,
  "createTxIntervalInMillis": 250,
  "maxOutboundConnections": 8,
  "maxInboundConnections": 117,
  "statsUpdateIntervalInMillis": 3000,
  "blockSizeLimit": 30
 }
}
```

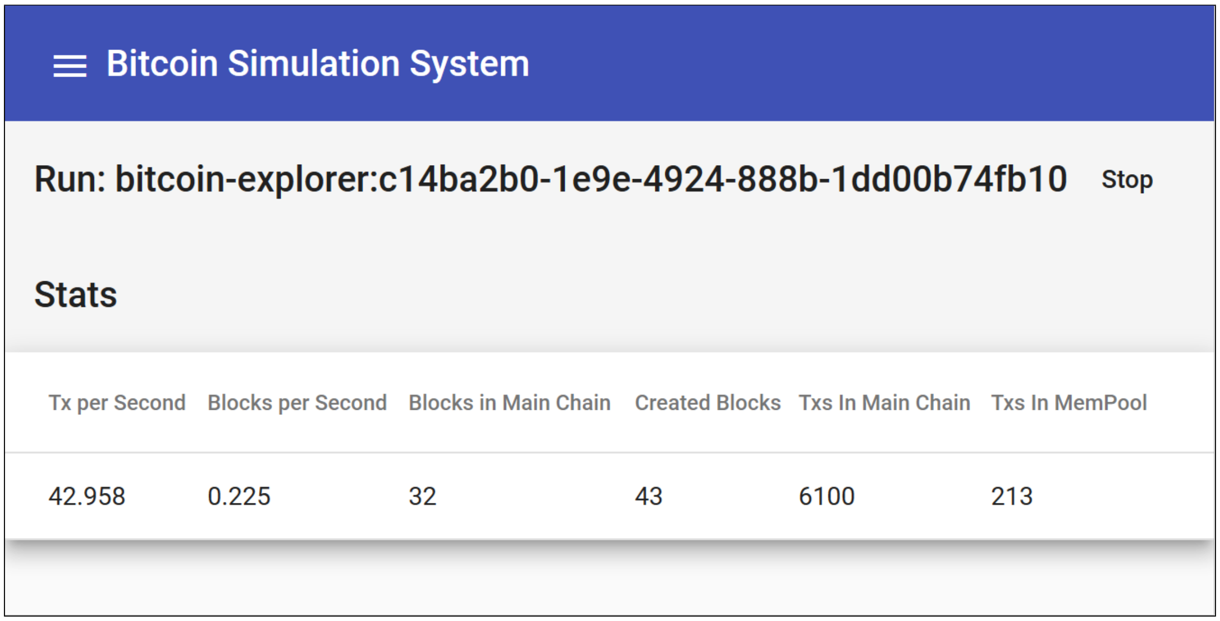
Figure 4.16: Web client's New Simulation Run view.

4.8.4 Simulation Run List View

Simulation Run List view shows all the previous simulation runs. This view is depicted in Figure 4.18 and has the following fields:

- **Run ID** - A unique ID for this simulation run that is assigned by the framework.
- **Simulation Name** - Name of this simulation that this run belongs to.
- **Start** - Start date and time of this simulation run.
- **End** - End date and time of this simulation run.

In addition to the metadata provided above, this view allows users to take two actions. First button is the show results button that navigates users to the dedicated results page of the run. Simulation specific result pages are described in detail under each simulation's related sections. Second button is the delete button. When clicked, it removes the run results from the system permanently.



Run: bitcoin-explorer:c14ba2b0-1e9e-4924-888b-1dd00b74fb10

Stats

Tx per Second	Blocks per Second	Blocks in Main Chain	Created Blocks	Txs In Main Chain	Txs In MemPool
42.958	0.225	32	43	6100	213

Figure 4.17: Web client's Active Simulation Run view.

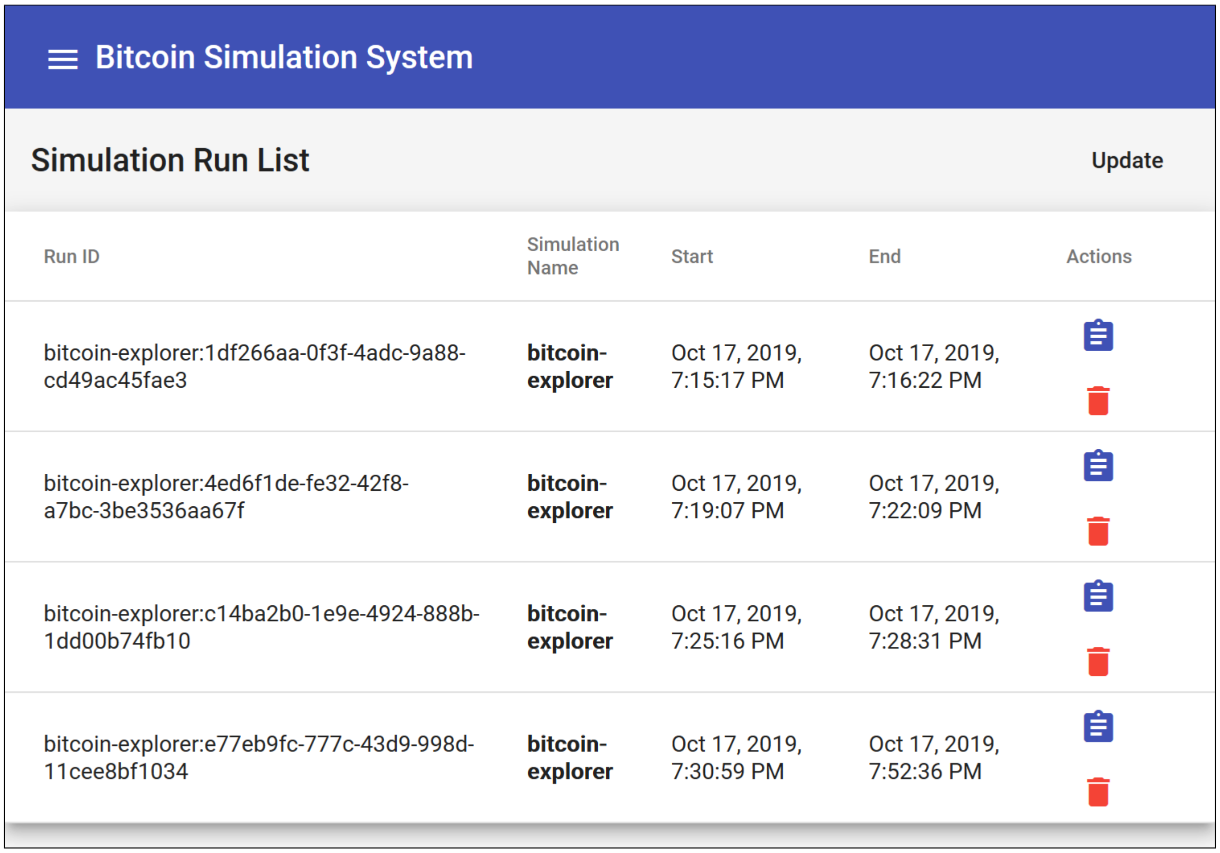
4.9 Simulation 1: Bitcoin Block Explorer

First simulation created as part of this thesis is the Block Explorer. It is inspired by the popular blockchain block explorers and simulators for education purposes. As described in the Section 3.3, block explorers give useful insights about the ledger being constructed. Having a Bitcoin block explorer simulation would allow students and newcomers understand how the protocol and the network behaves under different configurations. It would also let researchers tweak certain characteristics of the protocol via simulation configuration and study the results. Lastly, having a block explorer would allow users to asses if the framework is able to simulate the Bitcoin network as close to the real network as possible.

This simulation is also serves as an example for simulation developers who are considering to use DLSF. Following sections discusses the implementation details and capabilities of this simulation. Furthermore, it demonstrates how a simulation similar to this would be developed with DLSF.

4.9.1 Configuration Parameters

Bitcoin Explorer simulation is highly configurable. It allows tweaking many aspects of the Bitcoin network that will be simulated. The mining part of the protocol is simulated using random numbers and explained in detail in Section 4.9.5. Simulation users can



Run ID	Simulation Name	Start	End	Actions
bitcoin-explorer:1df266aa-0f3f-4adc-9a88-cd49ac45fae3	**bitcoin-explorer**	Oct 17, 2019, 7:15:17 PM	Oct 17, 2019, 7:16:22 PM	
bitcoin-explorer:4ed6f1de-fe32-42f8-a7bc-3be3536aa67f	**bitcoin-explorer**	Oct 17, 2019, 7:19:07 PM	Oct 17, 2019, 7:22:09 PM	
bitcoin-explorer:c14ba2b0-1e9e-4924-888b-1dd00b74fb10	**bitcoin-explorer**	Oct 17, 2019, 7:25:16 PM	Oct 17, 2019, 7:28:31 PM	
bitcoin-explorer:e77eb9fc-777c-43d9-998d-11cee8bf1034	**bitcoin-explorer**	Oct 17, 2019, 7:30:59 PM	Oct 17, 2019, 7:52:36 PM	

Figure 4.18: Web client's Simulation Run List view.

also adjust these parameters in order to get the most performance out of the available hardware. Configuration includes the following parameters:

- **numOfNodesPerPod** - Number of nodes that will be created on each pod.
- **maxBlockHeight** - Determines at which block height the simulation should complete.
- **miningDifficulty** - Specifies the difficulty of mining blocks. Section 4.9.5 explains how this parameter is used in simulated mining process.
- **mineAttemptIntervalInMillis** - Specifies at which intervals each node should attempt to mine. Section 4.9.5 explains how interval parameters are used.
- **createTxIntervalInMillis** - Specifies at which intervals each node should create new transactions. Section 4.9.5 explains how interval parameters are used. Parameter can be set to 0 to run simulations with no transactions.
- **blockReward** - Specifies how much a node should be awarded for mining a new

block.

- **genesisBlockTxOutForEachNode** - Specifies how many bitcoins worth of transaction outputs should be put into the genesis block for each node. This allows each node to start with some bitcoin to spend so that they can immediately start issuing transactions.
- **networkTopologySeed** - Specifies the random number generator seed used by the Bitcoin network coordinator. Same seed values generate identical network topologies. This is an optional parameter and when set to 0, completely random topologies are created at each run.
- **maxInboundConnections** - Specifies maximum number of inbound connections should a simulated node accept.
- **maxOutboundConnections** - Specifies maximum number of outbound connections should a simulated node initiate.
- **statsUpdateIntervalInMillis** - Specifies how often nodes should send statistics updates to the web socket subscribers. Statistics updates are the real time updates of an active simulation run.
- **blockSizeLimit** - Specifies the maximum number of transactions that can be put into a block.

4.9.2 Network Coordinator Service

Block explorer simulation creates only one service to allow simulated nodes to discover other nodes. To achieve this, simulation uses an instance of DLSF Bitcoin Module's BitcoinNetworkCoordinator actor described in Section 4.5.2. Service is configured based on user provided simulation run configuration. The design of DLSF simulation templates allow configuring each component on individual simulation runs and the block explorer simulation makes use of this design. BitcoinNetworkCoordinator is configured on each run based on the user provided simulation run configuration. The code snippet in Listing 4.2 shows how this logic is implemented.

```
@Override
public Collection<ServiceTemplate> getServiceTemplates() {
  ServiceTemplate bitcoinNetworkService = new ServiceTemplate();
  bitcoinNetworkService
    .setServiceKey(BitcoinNetworkCoordinator.SERVICE_KEY);
  bitcoinNetworkService.setFactory(
      runContext -> {
        SimulationConfig config = runContext
          .getSimulationConfig(SimulationConfig.class);
        BitcoinNetworkCoordinatorConfig coordinatorConfig =
          new BitcoinNetworkCoordinatorConfig();
        coordinatorConfig.setTotalNumOfNodes(
            runContext.getWorkerNames().size() *
            config.getNumOfNodesPerPod());
        coordinatorConfig.setMaxInboundConnections(
          config.getMaxInboundConnections());
        coordinatorConfig.setMaxOutboundConnections(
          config.getMaxOutboundConnections());
        if (config.getNetworkTopologySeed() != 0L) {
          coordinatorConfig.setSeed(config.getNetworkTopologySeed());
        }
        return BitcoinNetworkCoordinator
          .createBehavior(coordinatorConfig);
      });
    return List.of(bitcoinNetworkService);
  }
```

Listing 4.2: Block Explorer simulation's SimulationTemplate#getServiceTemplates() implementation.

4.9.3 Simulation Reducer

Block Explorer reducer is responsible for receiving data from simulation components and delivering it to the master when requested. Its message protocol consists of 4 types of messages that are shown in Figure 4.19. Two of these messages are implementations of the required BaseReducerProtocol messages explained in Section 4.4.7.

Both BlocksUpdate and StatsUpdate messages are sent by a limited number of simulated Bitcoin nodes selected randomly among all the nodes in the network. BlocksUpdate message is used to send the current state of the blockchain to the reducer, while StatsUpdate is used to communicate stats about the overall network such as transaction per second or the block time.

When received, StatsUpdate message is also sent to the users subscribed to updates via the web socket gateway. Finally, ResultMessage is the data transfer object used by the Reducer to communicate the simulation results. Reducer responds to the ResultRequest

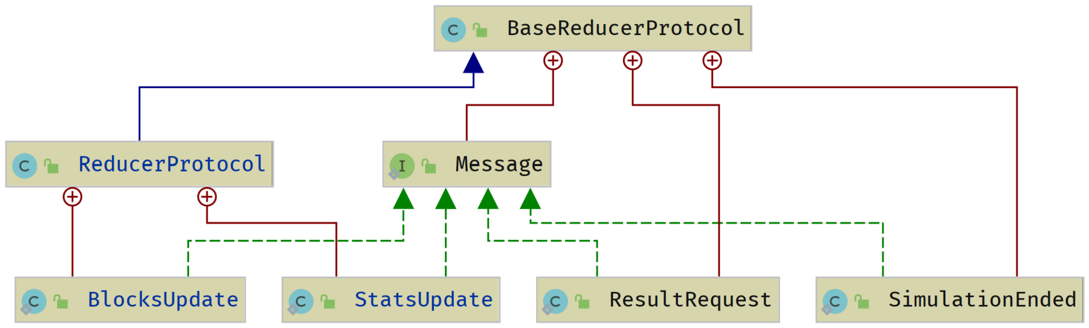


Figure 4.19: Block Explorer simulation reducer protocol class diagram.

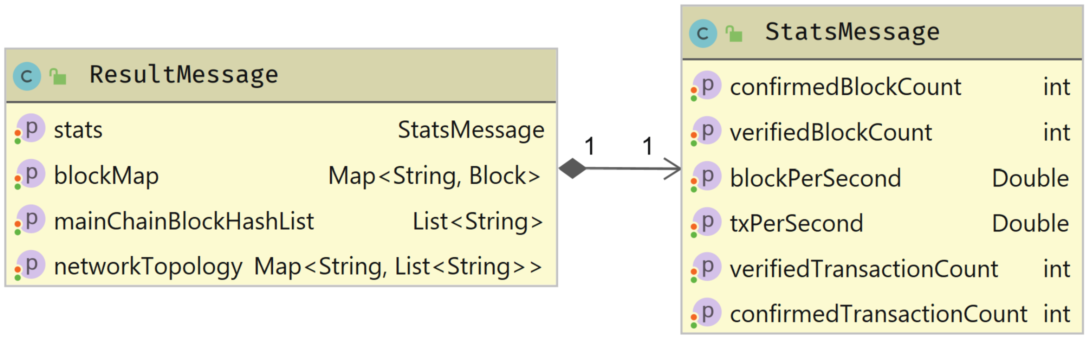

ResultMessage
stats StatsMessage
blockMap Map<String, Block>
mainChainBlockHashList List<String>
networkTopology Map<String, List<String>>
1
1
StatsMessage
confirmedBlockCount int
verifiedBlockCount int
blockPerSecond Double
txPerSecond Double
verifiedTransactionCount int
confirmedTransactionCount int

Figure 4.20: Block Explorer simulation reducer ResultMessage class diagram.

message with an instance of this class. Class diagrams of both messages are shown in Figure 4.20.

4.9.4 Simulation Pod

Pods are the main entry point for each simulation. In block explorer simulation, each pod is responsible for creating simulated Bitcoin nodes by providing their dependencies and coordinating with its worker throughout the simulation run. Simulated nodes use the network discovery service to find other nodes to connect. To be able to provide a reference to the service, each pod discovers the network discovery service via Akka receptionist using the service key. Then they start waiting for a prepare message from their workers. Upon receiving a prepare message, pods create number of nodes specified in the simulation run configuration.

Once created, nodes carry on with their own tasks for the simulation and the pod only handles coordinating the simulation stop phase. Nodes send stopped messages when their blockchain length becomes equal to the specified number in the configuration. When every node stops, the pod tells its worker that the simulation is ended on this instance.

4.9.5 Simulation Nodes

Nodes are instantiated by the pods with every dependency and configuration they need to start participating in the simulation run. They first register to the network discovery service actor and wait for coordinator to send their connection references. Once connections are received, they start behaving like a Bitcoin full node in the network. Each node uses DLSF Bitcoin module's full node implementation to manage its internal blockchain state. Nodes also to handle message propagation, block mining and transaction creation based on their configuration. In addition to participating in the network, some nodes are selected by their pods to send network statistics to the reducer so that users can be notified about the current state of the whole network. These network statistics include transaction per second, block time and other useful metrics. Finally, when the maximum blockchain length is reached, some selected nodes send their whole blockchain state to the reducer in order to be used as a simulation run result.

Mining and Transaction Issuing Intervals with Randomized Offset

Simulation configurations include interval parameters such as *mineAttemptIntervalInMillis* and *createTxIntervalInMillis* that are used to specify how often nodes should try to mine a block or issue new transactions to the network. But this approach alone would result in an unrealistic network where every node tries to mine or issues transactions at the same time. To align these processes with the real world scenarios, an offset time is introduced. Each node first generates a random offset that is between 0 and the specified interval, then it waits for the generated amount of offset time before it starts mining and issuing transactions in specified intervals. Using a randomized offset results in evenly distributed, random transaction messages and the block times within the boundaries of specified intervals.

Simulating the Mining Process

Compared to VIBES and BBSS, Bitcoin nodes in block explorer simulation do not require a central component for mining. Each node tries to mine a block at the specified intervals with a randomized offset explained in 4.9.5. At each interval, a floating-point random number is generated between 0 and 1, and if this number is greater than the specified difficulty node is deemed to find a valid block hash. Both the time interval and the difficulty is specified by users on the simulation run start. Compared to having a centralized component, mining based on random numbers scales a lot better and does not create a performance bottleneck. This approach is very similar to how blocks are mined in real world because results of cryptographic hashing functions also cannot be predicted and therefore random, resulting in block times that are fluctuating and not fixed.

Simulation of Transactions

Nodes in the simulation create transactions based on intervals specified in simulation run configuration. Similar to the real Bitcoin protocol implementation, origin of every transaction input can be tracked back to a block reward on the longest chain or the genesis block. In each interval, nodes check their wallets to see if they have any bitcoin to spend and if so they send some percentage of it to a randomly selected subset of nodes on the network.

4.9.6 Results Page

Block Explorer simulation result page is designed to be similar to the real world block explorers. Its main goal is to present the internal state of the simulated network and results to users. The results page consists of 4 main parts: stats, network topology, block graph and main chain explorer.

Stats View

Stats view shows general statistics about the network. It is identical to what is shown on the active simulation run Figure 4.17. Only difference is that results shown here are final results.

Node Connection Distribution Chart View

This view shows the distribution of total number of connections of the nodes. It distributes each node into buckets of connection numbers to give an overview of connectivity within the network. The view is shown in Figure 4.21.

Network Topology View

This view shows the network topology generated for this simulation run. With this view, users can analyze how parameters specified in the configuration would affect the generated network. The view is depicted in Figure 4.22 and only shows outbound connections initiated by each node.

Block Graph View

Block graph shows the final state of the block chain at the end of the simulation. It includes blocks in the longest chain and the stale blocks on the chains with smaller height.

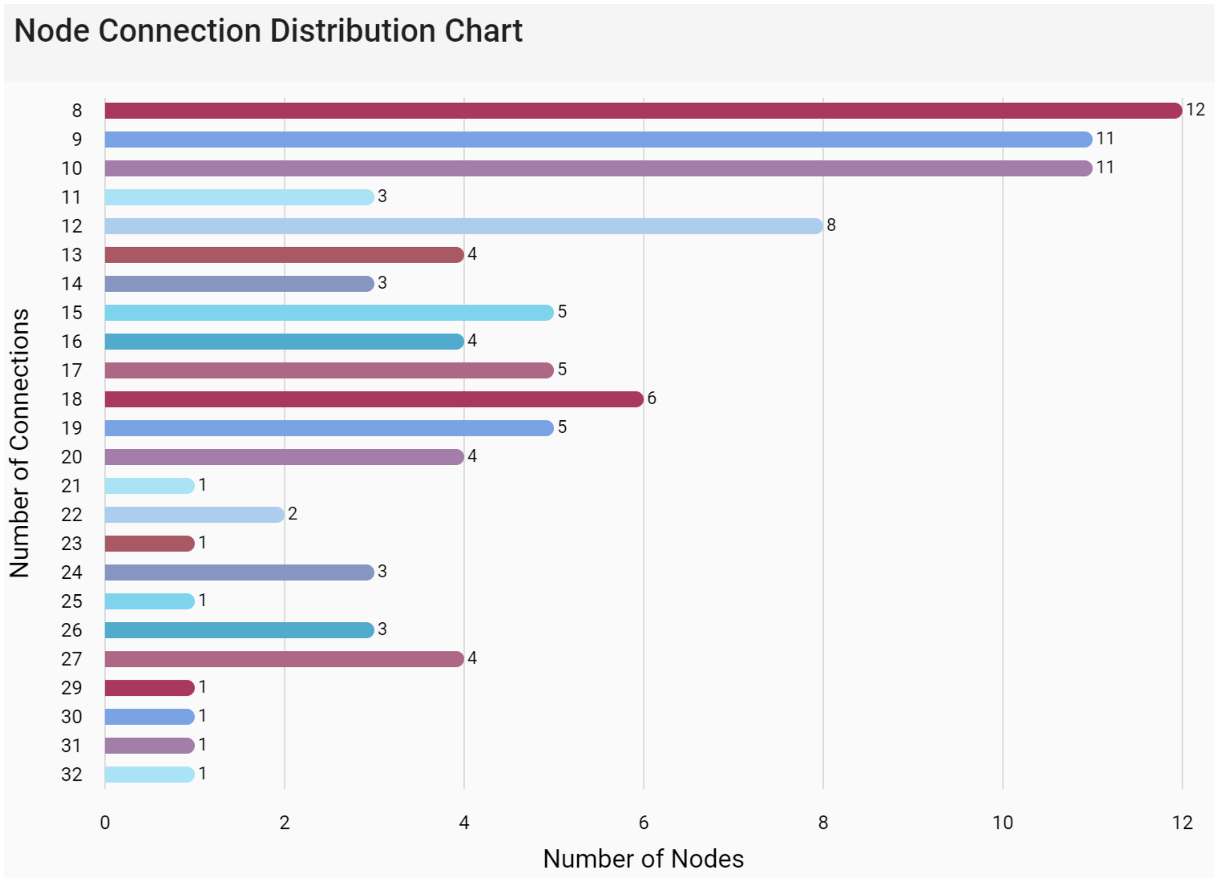


Figure 4.21: Block Explorer simulation results, connection distribution chart view.

Main Chain Explorer View

Main chain explorer view shows all blocks, transactions, transaction inputs and transaction outputs on the main chain. Table includes clickable links for navigating to the different parts of the chain. For instance when a transaction input is clicked, the table switches to the transaction that yielded the clicked output. It also includes all the information for the blocks and transactions such as mine time, previous block, miner's ID and transaction hashes. The view can be seen in Figure 4.24.

4.10 Simulation 2: Bitcoin Transaction Protocols

Erlay is a work in progress Bitcoin improvement proposal that is explained in detail in Section 2.5.9. It aims to improve over the existing transaction propagation protocol by reducing the amount of redundant messages. Running simulations to see how Erlay protocol compares against the existing message flooding protocol is a solid use case for a

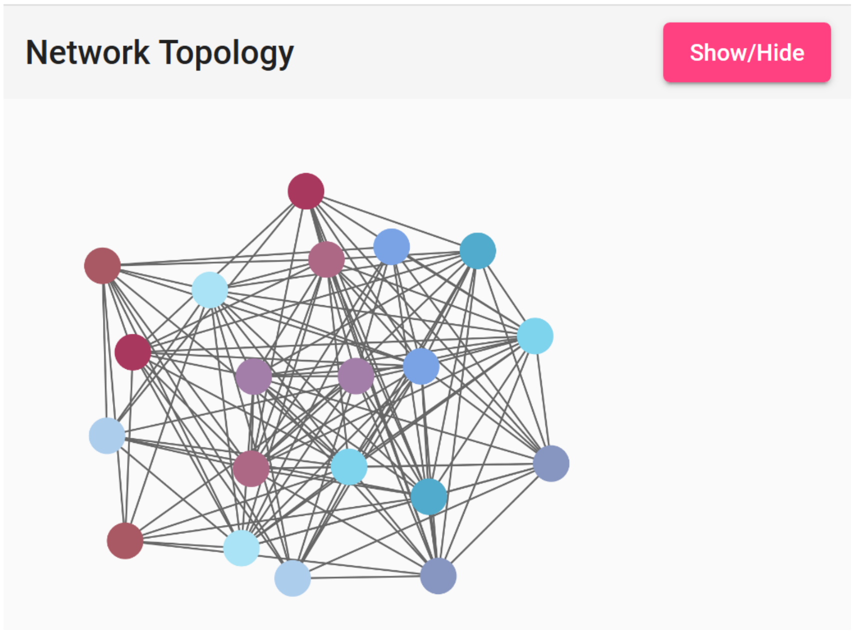


Figure 4.22: Block Explorer simulation results, network topology view.

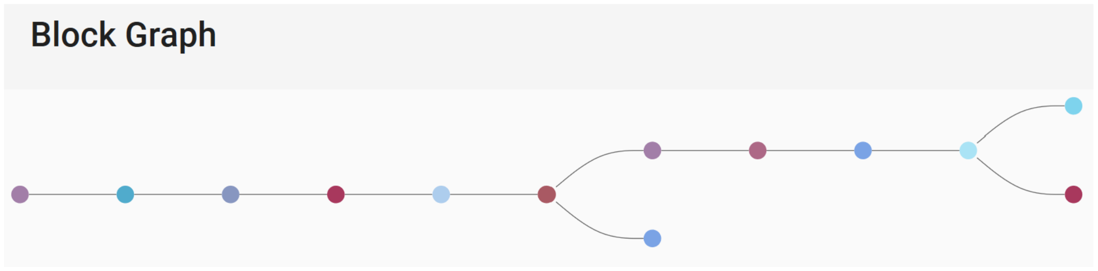
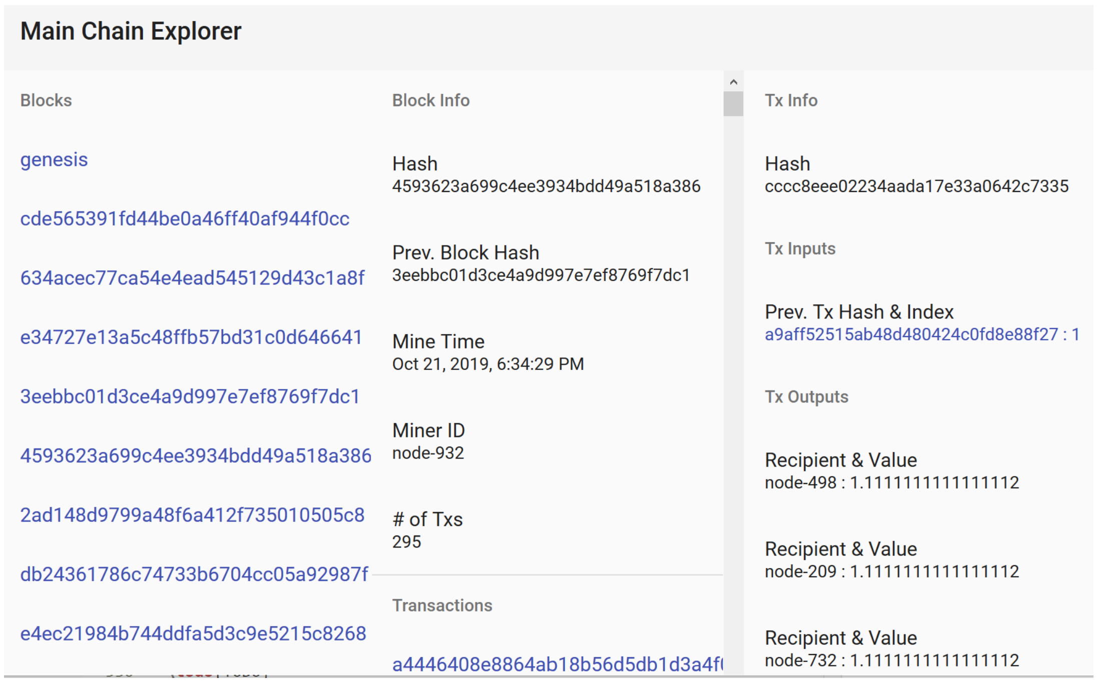


Figure 4.23: Block Explorer simulation results, block graph view.

simulation framework such as DLSF.

Assessing the performance of the new protocol would need two similar simulations that only differs in the way they propagate transactions. Two protocol simulations with the following names developed to achieve this: *bitcoin-tx-flood* and *bitcoin-tx-erlay*. Former uses the existing protocol while the latter implements and uses the proposed protocol.

Since Erlay only changes how transactions propagated, rest of the system should behave exactly the same. Because of this reason, simulations created for this purpose has mostly identical properties to the block explorer simulation discussed in Section 4.9. Both simulations uses the same network coordinator service and the reducer. Pods of each simulation is also explained in the following chapters.



Main Chain Explorer

Blocks

genesis

cde565391fd44be0a46ff40af944f0cc

634acec77ca54e4ead545129d43c1a8f

e34727e13a5c48ffb57bd31c0d646641

3eebbc01d3ce4a9d997e7ef8769f7dc1

4593623a699c4ee3934bdd49a518a386

2ad148d9799a48f6a412f735010505c8

db24361786c74733b6704cc05a92987f

e4ec21984b744ddfa5d3c9e5215c8268

Block Info

Hash
4593623a699c4ee3934bdd49a518a386

Prev. Block Hash
3eebbc01d3ce4a9d997e7ef8769f7dc1

Mine Time
Oct 21, 2019, 6:34:29 PM

Miner ID
node-932

# of Txs
295

Transactions

a4446408e8864ab18b56d5db1d3a4f

Tx Info

Hash
cccc8eee02234aada17e33a0642c7335

Tx Inputs

Prev. Tx Hash & Index
a9aff52515ab48d480424c0fd8e88f27 : 1

Tx Outputs

Recipient & Value
node-498 : 1.1111111111111112

Recipient & Value
node-209 : 1.1111111111111112

Recipient & Value
node-732 : 1.1111111111111112

Figure 4.24: Block Explorer simulation results, main chain explorer view.

4.10.1 Configuration Parameters

Both transaction protocol simulations use the same configuration parameters that are explained in the block explorer simulation. Only addition to the parameters is for the erlay proposal simulation that needs to run reconciliation. *erlayReconciliationInterval* parameter specifies how often nodes should run the reconciliation algorithm.

4.10.2 Network Coordinator Service

Both transaction protocol simulations uses the BitcoinNetworkCoordinator offered by the DLSF Bitcoin module. Its configuration is already explained in the block explorer simulation section.

4.10.3 Simulation Reducer

Transaction protocol simulations need to send updates and provide results about the general statistics of the simulated system. The reducer used by both simulations gather this information from the nodes in a data structure identical to block explorer's **StatsMessage**. Additionally, transaction protocol simulations need to collect data about all kinds of transaction messages that are received by nodes throughout simulation runs. For this purpose, the data class named NetworkStatsMessage is used. The result message class diagram returned from the reducer can be seen in Figure 4.25.

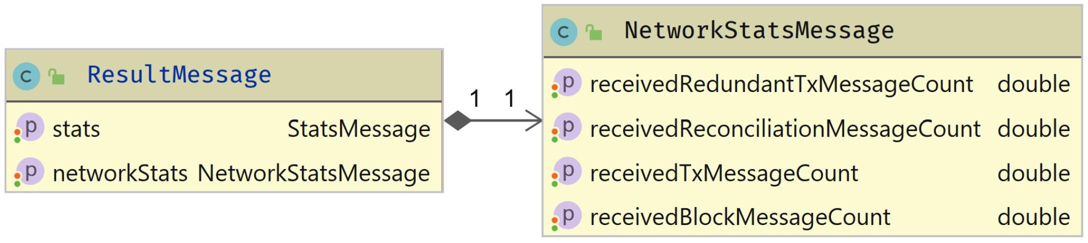

ResultMessage
stats StatsMessage
networkStats NetworkStatsMessage
1
1
NetworkStatsMessage
receivedRedundantTxMessageCount double
receivedReconciliationMessageCount double
receivedTxMessageCount double
receivedBlockMessageCount double

Figure 4.25: Transaction Protocol simulations, reducer ResultMessage class diagram.

4.10.4 Simulation Pod

Since both the block explorer and the transaction protocol simulations use the same service and reducer, their pod implementations are also very similar. The only difference is the factory function parameter that is capable of creating nodes using either the flood or erlay transaction propagation protocol. This factory function is provided via their simulation templates. Both Flood and Erlay nodes described in the following section extend a base Node class so that the same pod implementation can be used in both transaction protocol simulations. Excerpts from the source code in Listing 4.4 and Listing 4.4 show how this is achieved in simulation templates.

```
@Override
public Function<PodContext, Behavior<AbstractPodProtocol.Message>>
getPodFactory() {
  return simulationContext -> {
    return Pod.createBehavior(simulationContext,
      FloodNode::createBehavior);
  };
}
```

Listing 4.3: Pod factory implementation of Flood transaction protocol simulation template.

```
@Override
public Function<PodContext, Behavior<Message>> getPodFactory() {
  return simulationContext -> {
    return Pod.createBehavior(simulationContext,
      ErlayNode::createBehavior);
  };
}
```

Listing 4.4: Pod factory implementation of Erlay transaction protocol simulation template.

4.10.5 Simulation Nodes

As mentioned earlier, the behavior of both transaction protocol simulations are identical besides their transaction propagation algorithms. Because of this reason, inheritance is used to reduce the amount of code that needed to be written. Both FloodNode and ErlayNode implementations extend the abstract Node class as shown in Figure 4.26. Node class includes implementations of the common tasks such as block creation, transaction creation and block propagation. Subclasses implement their own propagation logic on top of this base class.

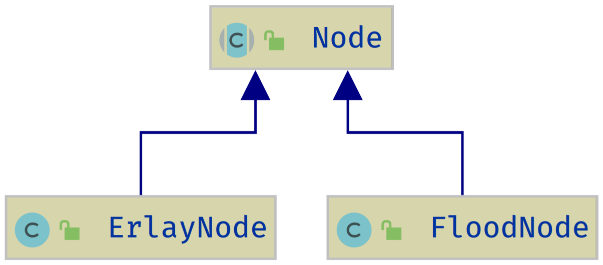


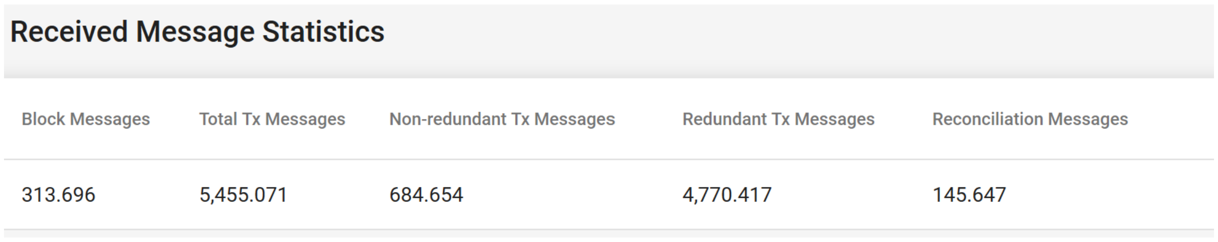
Figure 4.26: Transaction Protocol simulations, Node class diagrams.

FloodNode uses existing transaction propagation protocol used in Bitcoin implementations. It propagates newly discovered transactions to all of its neighbors except the sender. Erlay node on the other hand uses the new protocol proposed in Erlay protocol's paper. The only difference from proposal's implementation written in C++ is the reconciliation algorithm.

Erlay's real world implementation uses a custom library named Minisketch [31] for set reconciliation as described in Section 2.5.9. Unfortunately, there is no equivalent library written in Java for the same purpose. Because of this reason, ErlayNode does reconciliation by sending the whole reconciliation sets. Even though this would use a very high bandwidth in a real world scenario, it is not the case for the simulation. Messages within the same process passed via references and do not go over the wire, therefore this approach does not affect the overall simulation performance and still manages to reconciliate the differences between transaction sets of two nodes. Reconciliation algorithm runs as described in the paper on intervals specified in simulation run configurations by ErlayNode instances.

4.10.6 Results Page

Transaction simulation results page shows average message statistics of a node in the network. Similar to the block explorer results page, This page also includes general simulation run information such as start/end times and the blockchain state statistics. In addition to these, message statics are displayed both in numbers and on a distribution graph. Figure 4.27 shows the received messages statistics and Figure 4.28 shows the distribution graph. Users can test and examine how transaction propagation protocols behave under different configurations using this information.



Received Message Statistics

Block Messages	Total Tx Messages	Non-redundant Tx Messages	Redundant Tx Messages	Reconciliation Messages
313.696	5,455.071	684.654	4,770.417	145.647

Figure 4.27: DLSF transaction simulations result page, received message statistics table.

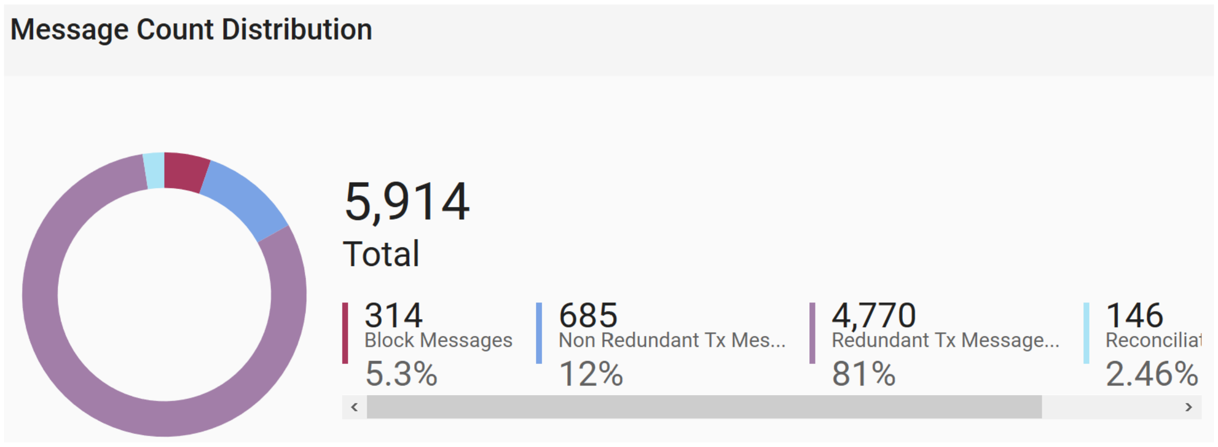


Figure 4.28: DLSF transaction simulations result page, message count distribution graph.

Chapter 5

Evaluation

This chapter evaluates both the framework and simulations created with it using 6 different metrics: Usefulness, Correctness, Scalability, Efficiency, Flexibility and Extensibility. All test simulations are run on a personal computer with an i7-6700HQ processor and 16 gigabytes of RAM.

5.1 Usefulness

Like any other good tool, DLSF should also satisfy some need and has to serve a purpose. In the previous chapters, it is already discussed how DLSF allows developers to create efficient simulations for different kinds of distributed ledger technologies. To demonstrate some use cases, two types of example simulations are developed. First of these examples is the block explorer and the second one is the transaction protocol simulations. Following sections discusses two of the possible use cases for using these simulations.

5.1.1 Simulations as an Education Tool

Bitcoin is a relatively new technology and it might be challenging for newcomers to grasp the internals of the protocol. Simulations such as block explorer 4.9 can be used as a learning tool to understand the inner workings of the protocol. Users can learn how Bitcoin works by examining how the network is structured and how blocks are formed using the graphical tools offered by the user interface.

Another use case would be changing certain properties of Bitcoin protocol and observing how they would affect the whole network. Bitcoin protocol has concrete set of rules about how nodes should operate in the network. For instance, things like block limit and the difficulty adjustments are hard coded in the protocol. There is also configurations that affect the network structure such as number of inbound and outbound connections. By using a simulation such as the block explorer, users can easily change these configurations and inspect the results in no time without needing to run real nodes on clusters of servers.

5.1.2 Testing Improvement Proposals

Bitcoin improvement proposals are very important for the ongoing development of the protocol and its official client, Bitcoin Core. Contributors and researchers test their ideas by either implementing their changes on a Bitcoin Core fork or by developing and running simulations. Especially in the early stages of these proposals, prototyping with simulations can bring great benefits. Thus, this is one of the important use cases for frameworks such as DLSF, where users can rapidly prototype their ideas.

As mentioned in Section 2.5.9, Erlay proposal is tested by the authors using a Bitcoin specific simulator. Transaction protocol simulations developed as part of this thesis could also be used to asses how Erlay compares with the existing Flood protocol. Results obtained from these simulations are discussed in Section 5.2.4, and they are very similar to the authors' findings. Given that DLSF is not tightly coupled with any distributed ledger implementation, it can be used to test other distributed ledger technology proposals as well.

5.2 Correctness

Simulations are expected to have similar characteristics that of a real world system they simulate and the DLSF simulations developed to simulate Bitcoin networks are no exception. Block explorer simulation allows users to investigate all aspects of the blockchain and the network. It demonstrates how simulations closely follow the real world protocol rules. This section discusses certain characteristics of Bitcoin networks and evaluates how simulation results compare to them. Finally, results of the transaction protocol simulations are discussed and their correctness are validated by comparing them to the Erlay protocol proposal paper's results.

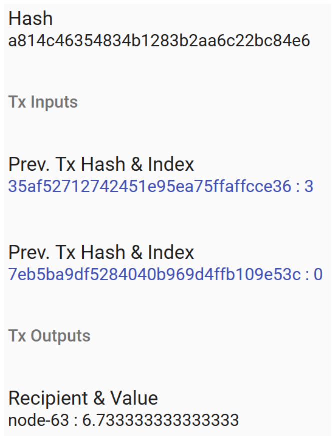
5.2.1 Correctness of Transactions

Transaction data structure in DLSF Bitcoin module is modeled after the real protocol, therefore it includes all the information that is needed for confirmation. Each transaction uses UTXOs as inputs and generates outputs that can be used by other transactions.

This allows one to track each input all the way to its origins similar to the real Bitcoin transaction.

Each received transaction goes through validation and verification steps that are modeled based on the Bitcoin Core implementation. Flowchart of these steps are shown in Figure 4.14. This means that each and every transaction is checked for validity and double spending before they are accepted by nodes. Since each transaction is validated and verified by nodes, only legitimate coins are spent during simulations. Simulations also ensure that output of any transaction is not greater than the sum of its input values.

DLSF bitcoin explorer simulation's block explorer view can be used to check and validate every confirmed transaction on the chain. The view is shown in Figure 4.24. It can also be used to track each transaction input all the way to its origin. Our assessments show that each confirmed transaction complies with the aforementioned protocol rules. Excerpt from an example simulation result is shown in Figure 5.1. Transaction **a814** uses transaction outputs of **35af** and **7eb5** as inputs. As seen on transaction details, total amount of UTXOs used in **a814** is equal to its output.



Hash
a814c46354834b1283b2aa6c22bc84e6

Tx Inputs

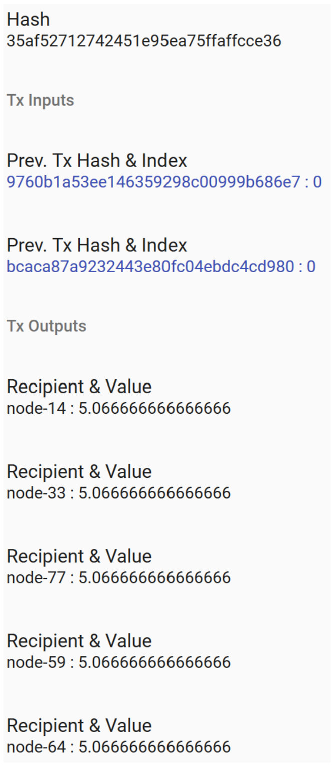
Prev. Tx Hash & Index
35af52712742451e95ea75ffaffcce36 : 3

Prev. Tx Hash & Index
7eb5ba9df5284040b969d4ffb109e53c : 0

Tx Outputs

Recipient & Value
node-63 : 6.733333333333333

(a) Details of example simulation transaction **a814**.



Hash
35af52712742451e95ea75ffaffcce36

Tx Inputs

Prev. Tx Hash & Index
9760b1a53ee146359298c00999b686e7 : 0

Prev. Tx Hash & Index
bcaca87a9232443e80fc04ebdc4cd980 : 0

Tx Outputs

Recipient & Value
node-14 : 5.066666666666666

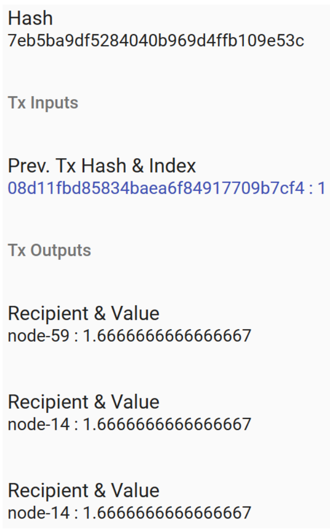
Recipient & Value
node-33 : 5.066666666666666

Recipient & Value
node-77 : 5.066666666666666

Recipient & Value
node-59 : 5.066666666666666

Recipient & Value
node-64 : 5.066666666666666

(b) Details of example simulation transaction **35af**.



Hash
7eb5ba9df5284040b969d4ffb109e53c

Tx Inputs

Prev. Tx Hash & Index
08d11fbd85834baea6f84917709b7cf4 : 1

Tx Outputs

Recipient & Value
node-59 : 1.6666666666666667

Recipient & Value
node-14 : 1.6666666666666667

Recipient & Value
node-14 : 1.6666666666666667

(c) Details of example simulation transaction **7eb5**.

Figure 5.1: Details of example simulation run transactions.

5.2.2 Correctness of Blocks

Unlike VIBES and BBSS where the direction of the chain is determined by a central component, block creation process is truly distributed in DLSF simulations. This resembles the real world events that can occur in the network. Full node implementation provided by DLSF Bitcoin module also follows the set of protocol rules in order to make simulations as realistic as possible. The algorithm followed after receiving of a block is shown in Figure 4.13 and it is very similar to the Bitcoin Core implementation.

Correctness of block chain

Block explorer simulation's block graph view shows all the chain blocks linked to their previous blocks. An example simulation is run to check correctness of constructed block chain. Figure 5.2 shows the whole block tree is constructed with confirmed and stale blocks as described in the protocol. Graph also shows that consensus is achieved among nodes as expected by following the longest chain algorithm. Figure 5.3 shows the details of 3 main chain blocks which are correctly linked to their previous block.

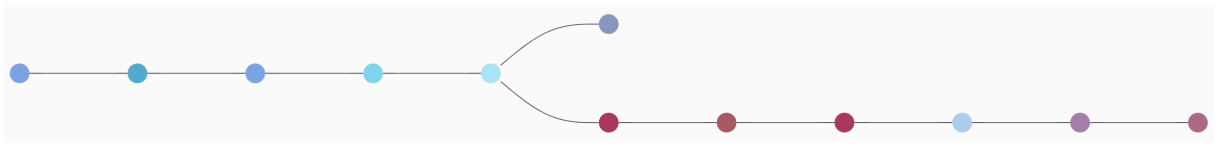
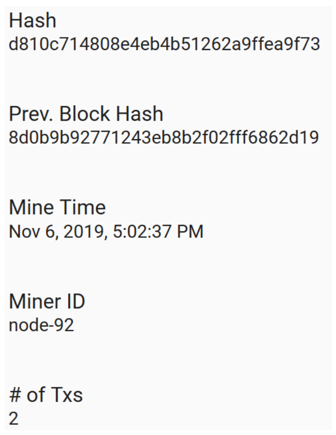


Figure 5.2: Block explorer simulation test results, block graph view.

.



Hash
d810c714808e4eb4b51262a9ffea9f73

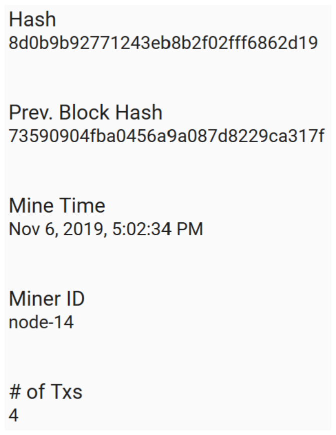
Prev. Block Hash
8d0b9b92771243eb8b2f02fff6862d19

Mine Time
Nov 6, 2019, 5:02:37 PM

Miner ID
node-92

# of Txs
2

(a) Details of example simulation block **d810**.



Hash
8d0b9b92771243eb8b2f02fff6862d19

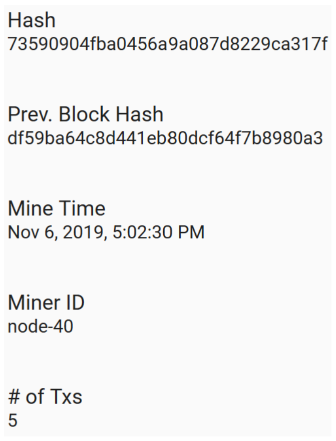
Prev. Block Hash
73590904fba0456a9a087d8229ca317f

Mine Time
Nov 6, 2019, 5:02:34 PM

Miner ID
node-14

# of Txs
4

(b) Details of example simulation block **8d0b**.



Hash
73590904fba0456a9a087d8229ca317f

Prev. Block Hash
df59ba64c8d441eb80dcf64f7b8980a3

Mine Time
Nov 6, 2019, 5:02:30 PM

Miner ID
node-40

# of Txs
5

(c) Details of example simulation block **7359**.

Figure 5.3: Details of example simulation run blocks.

Correctness of block times

Simulation nodes try to mine blocks in user specified intervals by generating random numbers and comparing it to the specified difficulty. This process is explained in Section 4.9.5. It is very similar to the hashing process in Bitcoin because results of hashing is also random and cannot be determined beforehand. Similar to the block times of Bitcoin, block generation times are also not fixed in simulations. Using random number generation allows block times to fluctuate similar to the real world.

A Bitcoin explorer simulation is run to compare block times of simulated networks with the Bitcoin mainnet. Simulation is configured to have 1000 nodes and its block time parameter is set to 6 seconds, which is 1/100 of the mainnet's block time. Complete set of configuration parameters are as follows:

- **numOfNodesPerPod:** 1000
- **maxBlockHeight:** 20
- **miningDifficulty:** 0.999
- **mineAttemptIntervalInMillis:** 6000
- **createTxIntervalInMillis:** 0
- **blockReward:** 25
- **genesisBlockTxOutForEachNode:** 100
- **networkTopologySeed:** 0
- **maxInboundConnections:** 117
- **maxOutboundConnections:** 8
- **statsUpdateIntervalInMillis:** 5000
- **blockSizeLimit:** 2038

Table 5.1 shows the comparison of block timestamps between mainnet and the simulated network. Random distribution can be seen on both timestamps. Mainnet's block times range from 5 seconds to 48 minutes and simulated network's block times range from less than a second to 14 seconds. This proves that DLSF Bitcoin simulation block times resemble the random distribution aspect of mainnet's block times.

Mainnet Blocks Blocktime: 10 minutes		Simulated Blocks Blocktime: 0.1 minutes	
Height	Timestamp	Timestamp	Height
602313	Nov 4, 2019 1:31:42 PM	Nov 4, 2019 4:29:34 PM	1
602314	Nov 4, 2019 1:35:07 PM	Nov 4, 2019 4:29:43 PM	2
602315	Nov 4, 2019 1:46:02 PM	Nov 4, 2019 4:29:47 PM	3
602316	Nov 4, 2019 1:50:21 PM	Nov 4, 2019 4:29:47 PM	4
602317	Nov 4, 2019 2:09:34 PM	Nov 4, 2019 4:30:01 PM	5
602318	Nov 4, 2019 2:09:39 PM	Nov 4, 2019 4:30:06 PM	6
602319	Nov 4, 2019 2:12:36 PM	Nov 4, 2019 4:30:11 PM	7
602320	Nov 4, 2019 3:00:56 PM	Nov 4, 2019 4:30:26 PM	8
602321	Nov 4, 2019 3:04:56 PM	Nov 4, 2019 4:30:29 PM	9
602322	Nov 4, 2019 3:06:34 PM	Nov 4, 2019 4:30:40 PM	10
602323	Nov 4, 2019 3:08:15 PM	Nov 4, 2019 4:30:43 PM	11
602324	Nov 4, 2019 3:10:38 PM	Nov 4, 2019 4:30:44 PM	12
602325	Nov 4, 2019 3:23:08 PM	Nov 4, 2019 4:30:46 PM	13
602326	Nov 4, 2019 3:52:30 PM	Nov 4, 2019 4:30:49 PM	14
602327	Nov 4, 2019 3:57:18 PM	Nov 4, 2019 4:30:54 PM	15
602328	Nov 4, 2019 4:03:57 PM	Nov 4, 2019 4:31:06 PM	16
602329	Nov 4, 2019 4:08:49 PM	Nov 4, 2019 4:31:20 PM	17

Table 5.1: Comparison of block times between Bitcoin mainnet and DLSF simulated network. All timestamps are in UTC+1 time zone. Bitcoin mainnet data taken from Bitcoinchain [8].

5.2.3 Correctness of Network

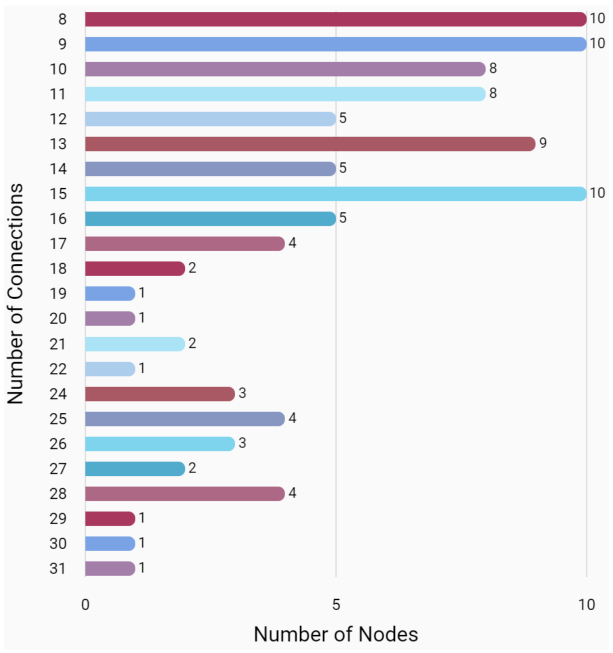
Bitcoin nodes usually connect up to 8 nodes and can accept connections from 117 other nodes. They form an unstructured network topology by following this simple rule. Topologies created by simulations also follow this rule and it can be configured using parameters similar to the Bitcoin Core. This results in a simulation network topology that resembles the network of mainnet nodes. Moreover, simulation developers can set the *networkTopologySeed* parameter to have identical network topologies in different simulation runs.

DLSF simulations have minor differences in a way the messages are propagated. Bitcoin nodes first announce the newly discovered data and then send it to their neighbors only when requested. This saves the bandwidth and prevents unnecessary usage. On the other hand, DLSF simulations mostly run within the same process where messages are passed using references and there is no bandwidth concerns. Because of this reason, simulation nodes propagate all the information at once instead of using the two-step approach used in Bitcoin. Besides this difference, rest of the message propagation resembles the real world behavior of the network. Nodes only receive messages from their neighbors via message propagation and do not share state by other means among themselves.

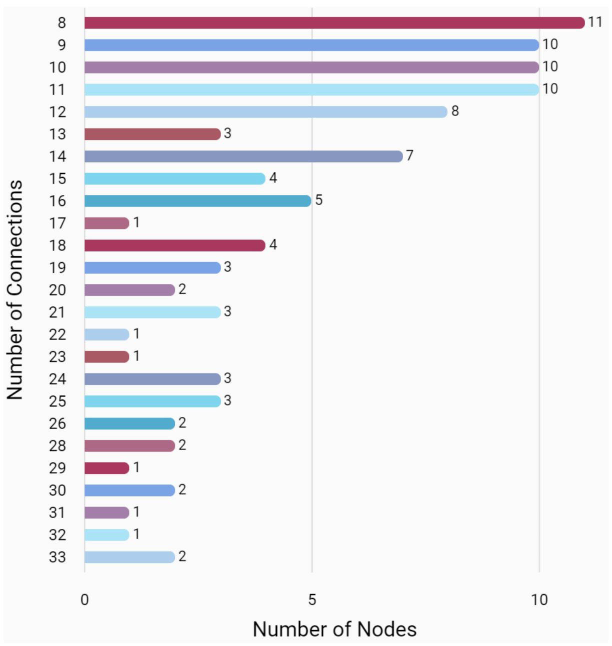
To assess the correctness of the created network topologies and to test if *networkTopologySeed* parameter works as expected, 4 different simulations are run with the parameters shown in Table 5.2. Node connection count distribution chart of each run are taken from the result page of the block explorer simulation and shown in 5.4. Each node has at least 8 connections and no node has more than 125 total connections as expected. Identical distributions of connection counts for Run 3 and Run 4 can be seen in Figure 5.4c and Figure 5.4d. Figures indicate that the framework respects the *networkTopologySeed* parameter and generates identical topologies when the same value is provided.

Run	Node Count	Max Outbound Con.	Max Inbound Con.	Topology Seed
1	100	8	117	0 (Random)
2	100	8	117	9999
3	100	8	117	1234
4	100	8	117	1234

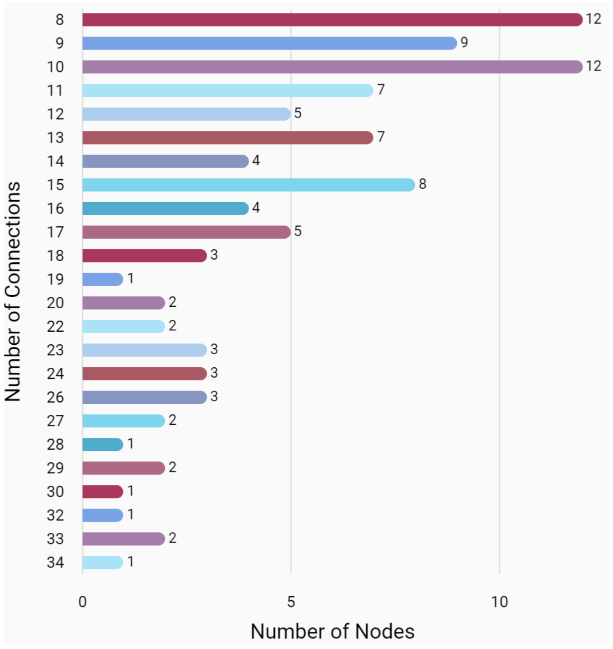
Table 5.2: Simulation configuration parameters of 4 simulation test runs.



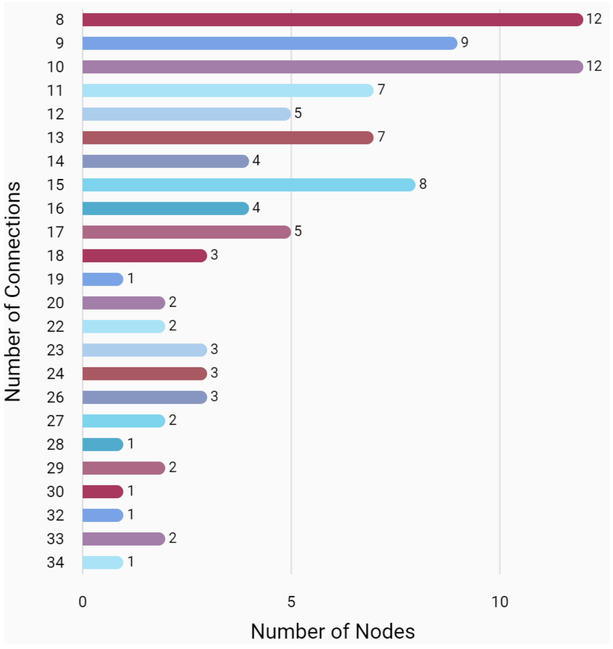
(a) Node connection count distribution of Simulation Run 1.



(b) Node connection count distribution of Simulation Run 2.



(c) Node connection count distribution of Simulation Run 3.



(d) Node connection count distribution of Simulation Run 4.

Figure 5.4: Node connection count distribution charts of simulation runs. Parameters of each run is shown in Table 5.2.

5.2.4 Correctness of Transaction Protocol Simulations

As mentioned in Section 2.5.9, better connectivity among nodes means better security and currently the biggest factor limiting the connectivity of the Bitcoin network is the bandwidth usage of the existing transaction propagation protocol referred to as Flood. Newly proposed protocol named Erlay claims to scale better with increasing connectivity and the authors already backed their claim with simulation results that are shown in Figure 2.3.

DLSF Bitcoin transaction protocol simulations are developed to obtain network massage statistics of networks that are using these two protocols. By comparing findings from these simulations with the results of Erlay's authors, one can assess the correctness of these simulations. Each protocol is tested with increasing connectivity from 8 outbound connections up to 32 outbound connections. Each simulation is run for approximately 120 seconds, and rest of the simulation configuration parameters are identical for each run. Table 5.3 shows the run results for each combination. DLSF simulations do not measure the size of the transaction and block messages because size of data objects used in its Java implementation drastically differs from Bitcoin Core's C++ implementation. Results shown on each column is as follows: transaction protocol, number of outbound connections, confirmed transactions per second and the average number of transaction messages, non-redundant transaction messages, redundant transaction messages and reconciliation messages received by each node.

Protocol	Out. C.	Tx/S	Total Tx M.	Non R. Tx M.	Redun. Tx M.	Recon. M.
Flood	8	24.13	42228.84	2836.48	39392.36	0
Flood	16	24.51	86988.04	2850.28	84137.76	0
Flood	24	24.67	131124.58	2850.28	128267.45	0
Flood	32	24.29	173702.13	2849.05	170853.08	0
Erlay	8	23.77	6959.25	865.78	6093.47	116.21
Erlay	16	22.47	8421.96	930.45	7491.51	112.51
Erlay	24	23.33	8576.74	948.07	7628.67	112.06
Erlay	32	23.41	8576.25	947.3	7628.95	111.47

Table 5.3: Transaction protocol simulation run results with increasing connectivity.

Similar to the findings of the Erlay proposal paper, DLSF transaction protocol simulation results also prove that with increasing connectivity, Erlay protocol scales much better than the existing Flood protocol and maintains the same rate of confirmed transactions per second. Erlay uses less transaction messages and significantly reduce redundant message count in the network. Only additional messages used in Erlay protocol are the set reconciliation messages. As discussed in Section 2.5.9, reconciliation messages are designed to be efficient and small.

Unlike Erlay proposal's results that compare the bandwidth usage of each protocol, DLSF simulation compares the number of different messages. Since each message type is indicated in the results, one can use average block and transaction message sizes to calculate the bandwidth difference if needed. Figure 5.5 show the bar chart representations of the results presented in Table 5.3 for easy comparison with the result graph of the Erlay protocol shown in Figure 2.3.

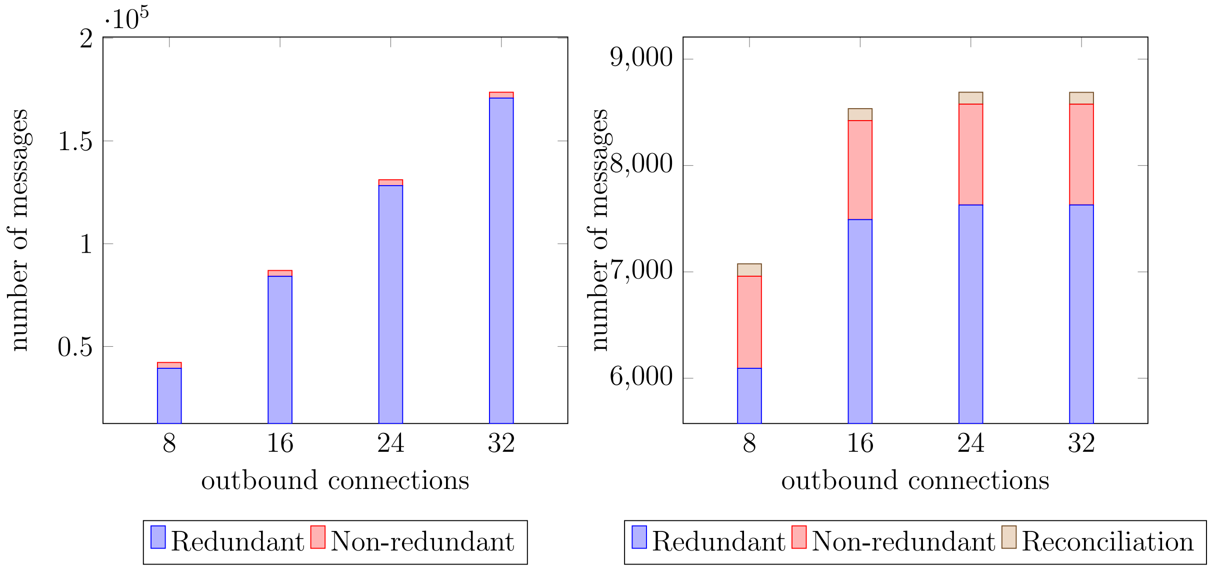


Figure 5.5: Transaction protocol simulation run results with increasing connectivity. Chart on the left shows Flood protocol results and chart on the right shows Erlay protocol results.

5.3 Speed and Scalability

Simulations developed with DLSF are very efficient and can scale both horizontally and vertically. Following sections discuss the speed and scalability limits of DLSF simulations on the given hardware.

5.3.1 Number of Nodes

In order to test how many Bitcoin nodes can be simulated by DLSF simulations on the given hardware, a number of simulations are run with disabled transaction messages. The system was able to simulate up to 6000 nodes. Since each node keeps its own state isolated from other nodes, running 6000 nodes means maintaining 6000 blockchain states. Figure 5.6 shows the memory usage increase with increasing number of nodes.

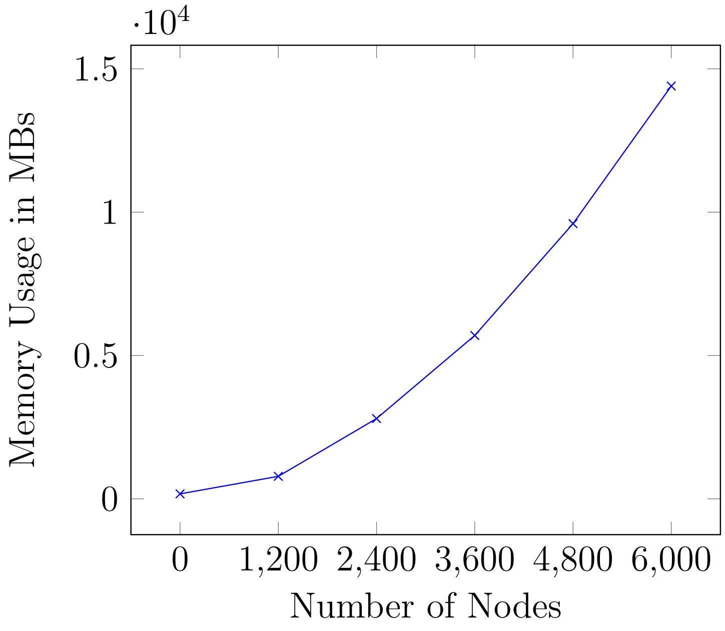


Figure 5.6: DLSF block explorer simulation's memory usage with increasing number of blocks.

5.3.2 Block Mining Rate

As explained in Section 4.9.5, DLSF simulations simulate the block mining process without needing a central component that could limit the overall performance of the simulation. A set of simulations are run with disabled transactions to test the speed of this simulated mining process. Even though a network without transaction messages does not completely resemble the real world scenario, obtained results still serve as an important scaling metric. Each simulation test run was configured with the following parameters:

- **numOfNodesPerPod:** 1000
- **maxBlockHeight:** 5000
- **miningDifficulty:** 0.999
- **mineAttemptIntervalInMillis:** 10

- **createTxIntervalInMillis:** 0
- **blockReward:** 25
- **genesisBlockTxOutForEachNode:** 0
- **networkTopologySeed:** 0
- **maxInboundConnections:** 117
- **maxOutboundConnections:** 8
- **statsUpdateIntervalInMillis:** 5000
- **blockSizeLimit:** 2038

Simulation run results are shown in Table 5.4. Simulated framework was able to reach block times of 0.023 seconds. In comparison to the block times of Bitcoin mainnet (600 seconds), DLSF was able to simulate block mining 26,087 times faster on average.

Run	Blocks per Second	Block Time (seconds)	Confirmed Blocks	Stale Blocks
1	44.248	0.0226	5000	138
2	43.478	0.0230	5000	142
3	44.248	0.0226	5000	161
4	45.045	0.0222	5000	142
5	42.735	0.0234	5000	163

Table 5.4: Block time test simulation run results.

5.3.3 Number of Transactions

Transaction messages are propagated through the whole network via flooding in Bitcoin. In order to test the transaction simulation performance of DLSF Bitcoin simulations, block explorer simulation is run 8 times with the following configuration parameters:

- **numOfNodesPerPod:** 1000
- **maxBlockHeight:** 10
- **miningDifficulty:** 0.999
- **mineAttemptIntervalInMillis:** 5000

- **createTxIntervalInMillis:** 25000
- **blockReward:** 25
- **genesisBlockTxOutForEachNode:** 100
- **networkTopologySeed:** 0
- **maxInboundConnections:** 117
- **maxOutboundConnections:** 8
- **statsUpdateIntervalInMillis:** 5000
- **blockSizeLimit:** 2038

Simulation results are shown in Table 5.5. 1000 nodes were able to simulate a network with confirmed transaction rates close to 30 transactions per second, which is 10 times faster than real network.

Run	1	2	3	4	5	6	7	8
Tx per Second	27.542	37.250	25.411	31.640	30.773	33.952	20.154	28.000

Table 5.5: Transaction rate test simulation run results.

5.3.4 Cluster Mode

As discussed in Section 4.4.1, DLSF simulations can be run on multiple processes that span across server clusters. An example simulation is run on the given hardware to showcase the cluster mode and to assess if it works as expected. The system had 1 master and 3 worker nodes, similar to the architecture shown in Figure 4.2b. Simulation is run with the following configuration parameters and Akka application configuration file for clustering is shown in 5.1.

- **numOfNodesPerPod:** 100
- **maxBlockHeight:** 10
- **miningDifficulty:** 0.993
- **mineAttemptIntervalInMillis:** 3000
- **createTxIntervalInMillis:** 30000
- **blockReward:** 25

- **genesisBlockTxOutForEachNode:** 100
- **networkTopologySeed:** 0
- **maxInboundConnections:** 117
- **maxOutboundConnections:** 8
- **statsUpdateIntervalInMillis:** 5000
- **blockSizeLimit:** 2038

```
akka {
    actor {
        provider = "cluster"
        allow-java-serialization = on
        serializers {
        java = "akka.serialization.JavaSerializer"
        }
    }
    remote {
        artery {
        enabled = on
        transport = tcp
        canonical.hostname = "127.0.0.1"
        canonical.port = 2552 // port for master
// canonical.port = 2553 // port for worker1
// canonical.port = 2554 // port for worker2
// canonical.port = 2555 // port for worker3
        }
    }
    cluster {
        seed-nodes = [
        "akka://DLSFSystem@127.0.0.1:2552"
        ]
        auto-down-unreachable-after = 10s
    }
}
```

Listing 5.1: Clustered simulation run application.conf file.

In this example run each worker pod was simulating 100 nodes, which makes total count of nodes on the network 300. Figure 5.7 shows the constructed chain and Table 5.6 shows the number of main chain blocks mined by each worker's nodes. Results show that even though the nodes were running in different JVM processes, they successfully collaborated and constructed the blockchain. This proves that simulated nodes in clustered mode were able to communicate and work as if they were running in the same process.

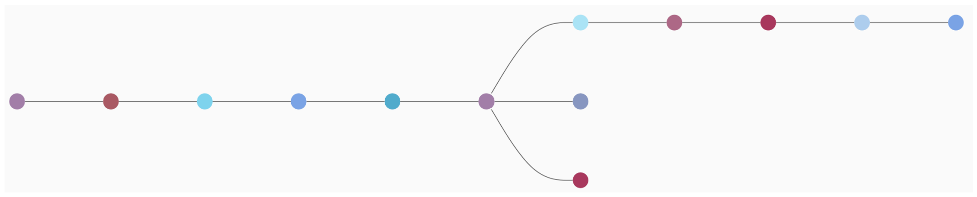


Figure 5.7: Block chain graph constructed in a clustered simulation run.

Mined by nodes in Worker 1	Mined by nodes in Worker 2	Mined by nodes in Worker 3
3	4	3

Table 5.6: Clustered simulation run: number of main chain blocks mined by each worker's nodes.

5.4 Efficiency

DLSF simulations are capable of utilizing available resources efficiently by using the actor model combined with Akka's powerful event loop and messaging implementations. DLSF runtime does not introduce any performance penalties during simulation runs. Unlike other Bitcoin simulations such as VIBES and BBSS, simulations developed as part of this thesis does not have a central component that hinders the performance. But since mining and transaction creation are totally random and distributed, configuration parameters play a great role to make most of the available resources. Parameters such as number of nodes, mine interval, transaction creation interval and number of connections determine how demanding a simulation run is going to be. Users are encouraged to test different combinations of these parameters in order to find combinations that work best for their hardware and use cases.

To demonstrate the varying CPU usage, two simulation runs are created with same parameters except different transaction creation intervals. Each node generates a transaction at this interval and every new transaction propagates through the whole network, thus directly effecting the resource usage. Both runs simulate 1000 nodes that try to mine a block every 3 seconds. CPU usages of the first simulation run simulating 20 confirmed transactions per second and the second simulation run simulating 30 confirmed transactions per second is shown in Figure 5.8 and Figure 5.9 respectively. CPU usage graphs prove that DLSF managed to utilize all the available CPU resources when needed.

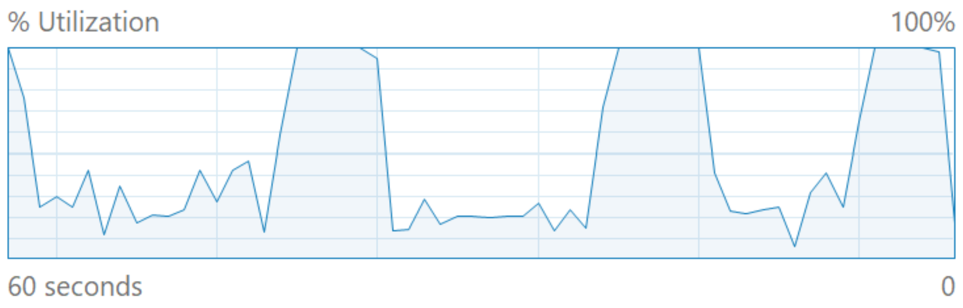


Figure 5.8: CPU utilization of a simulation run with 20 transaction confirmations per second.

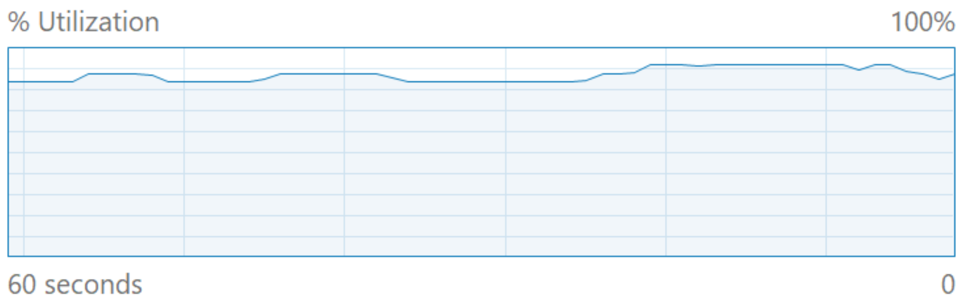


Figure 5.9: CPU utilization of a simulation run with 30 transaction confirmations per second.

5.5 Flexibility

Having many configuration parameters allows users to run simulations under different conditions without changing the source code. DLSF allows simulations to have arbitrary number of configuration parameters. Block explorer and transaction protocol simulations provide maximum flexibility to their users by offering the following configuration parameters:

- **numOfNodesPerPod** - Number of nodes that will be created on each pod.
- **maxBlockHeight** - Determines at which block height the simulation should complete.
- **miningDifficulty** - Specifies the difficulty of mining blocks. Section 4.9.5 explains how this parameter is used in simulated mining process.

- **mineAttemptIntervalInMillis** - Specifies at which intervals each node should attempt to mine. Section 4.9.5 explains how interval parameters are used.
- **createTxIntervalInMillis** - Specifies at which intervals each node should create new transactions. Section 4.9.5 explains how interval parameters are used. Parameter can be set to 0 to run simulations with no transactions.
- **blockReward** - Specifies how much a node should be awarded for mining a new block.
- **genesisBlockTxOutForEachNode** - Specifies how many bitcoins worth of transaction outputs should be put into the genesis block for each node. This allows each node to start with some bitcoin to spend so that they can immediately start issuing transactions.
- **networkTopologySeed** - Specifies the random number generator seed used by the Bitcoin network coordinator. Same seed values generate identical network topologies. This is an optional parameter and when set to 0, completely random topologies are created at each run.
- **maxInboundConnections** - Specifies maximum number of inbound connections should a simulated node accept.
- **maxOutboundConnections** - Specifies maximum number of outbound connections should a simulated node initiate.
- **statsUpdateIntervalInMillis** - Specifies how often nodes should send statistics updates to the web socket subscribers. Statistics updates are the real time updates of an active simulation run.
- **blockSizeLimit** - Specifies the maximum number of transactions that can be put into a block.
- **erlayReconciliationInterval** - Specifies how often nodes should run the reconciliation algorithm. It is only used in Erlay protocol simulation.

5.6 Extensibility

DLSF simulations are not expected to be extensible and the reason behind this is explained in previous chapters. Building a simulator for one use case and trying to extend it to support others proved to be a very hard undertaking. This was also one of the main reasons why the previous simulators failed to be truly extensible. Instead, DLSF offers a different approach by providing a modular framework architecture and encouraging developers to use these modules. In DLSF, framework itself is extensible and each use case is a different simulation developed on top of it. Three simulations that are able to test two different scenarios developed as part of this thesis. Being able to complete such task within couple of months proves how extensible the framework is.

Another unique aspect of DLSF is the concept of a simulation system. Simulations created with DLSF, regardless of their purpose or implementation details can be deployed as a single system. Users can decide which simulation to run and access individual results from different simulations. This is another level of extensibility for the framework that was not found in previous simulators.

Chapter 6

Summary

This thesis takes extensible blockchain simulator research done in TU Munich as a foundation, and proposes a different approach for developing extensible, large scale blockchain simulations. First shortcomings and findings of the previous research work is discussed and requirements of an extensible simulation systems are listed. Based on these requirements, a modular simulation framework that aims to accelerate building such systems is proposed and implemented. In addition to the development of the framework, two types of simulations are built to demonstrate the extensibility, correctness and usability of the framework.

First of these simulations is called Bitcoin block explorer simulation. It is inspired by mainnet block explorers and is developed to demonstrate correctness of the simulated systems. It also serves as an education tool to teach newcomers how Bitcoin works. Evaluation of the Bitcoin block explorer simulation has shown that simulated networks resemble the characteristics of the real Bitcoin network.

Second type of simulations implemented in this thesis are Bitcoin transaction protocol simulations. These simulations are implemented to compare a candidate transaction propagation protocol called Erlay to the existing Flood protocol. They also serve as an example of how DLSF can be used to prototype and test new proposals in a very short time. Finally, results produced by these simulations are compared to the similar results found in Erlay protocol's proposal paper [3].

6.1 Status

Distributed Ledger Simulation Framework currently consists of one main and two feature modules, and it currently fulfills many the requirements that are listed in Section 4.1. Some missing features that are anticipated but could not be implemented during the timespan of this thesis are discussed in Section 6.3. In addition to the framework, example simulations are also completed and ready to be used by users.

6.2 Conclusions

There are many possible use cases for efficient and extensible distributed ledger simulation systems. But currently there is no such widely adopted simulation system that is capable of simulating more than one distributed ledger technology. This was the main motivation behind the research of DLSF and other simulators developed at TU Munich. Unlike previous simulators, DLSF is designed from ground up to be extensible and approaches the same problem with a different mindset. Evaluation of the results show that DLSF lives up to its promise, and simulations built with it have the ability to simulate wide variety of use cases.

6.3 Future Work

Following sections discuss the possible extensions and improvements for the DLSF framework.

6.3.1 Faster Message Serializers

DLSF uses object references for in-memory message propagation but simulations run in cluster mode require serialization for inter-process messages. DLSF currently utilizes Java serializers for this task. Java serializers are easy to use, but they are also slower and generate bigger messages. Akka provides configurations for custom serializers and it can be utilized in future to improve the message throughput and overall performance of clustered simulations.

6.3.2 Network propagation delay

DLSF does not simulate the network propagation delays that occur in real networks. This was one of the requirements that are listed in this paper but due to the time limitation it could not be implemented. Simulation of a network delay on the messaging layer would be an important improvement to the framework and it would enable even more realistic simulations.

6.3.3 Simulations for Other Distributed Ledger Technologies

Example simulations implemented for DLSF target two different use cases for the framework, but they are both based on Bitcoin protocol. Implementing simulations for other distributed ledger technologies would be a great opportunity to test extensibility of the framework.

6.3.4 Handling Simulation Exceptions

DLSF currently does not have a central exception and error handling mechanism in place for simulation runs. Even though happy paths of the application works as intended, it would be nice to have a central exception handling and error reporting component that could communicate errors during simulation runs to the users. This would also help simulation developers during implementation because they would not need to dig into application log messages to identify errors in their code.

6.3.5 Scheduling Simulation Runs

DLSF is only able to run one simulation at a time and there is no way to schedule the next simulation run before the current one finishes. Considering the fact that some simulations might run for hours, users constantly need to monitor if the current run is finished before they start another. Implementing a scheduling mechanism would greatly improve the user experience and allow users to schedule multiple simulations runs once.

Appendices

Appendix A

Appendix

A.1 Project Structure

DLSF is a multi module maven project and it currently includes 6 Java packages and a web client project. Example simulations and related packages are prefixed with an *x* and framework related packages are prefixed with *dlsf*. Figure A.1 shows the dependency diagram of Java packages.

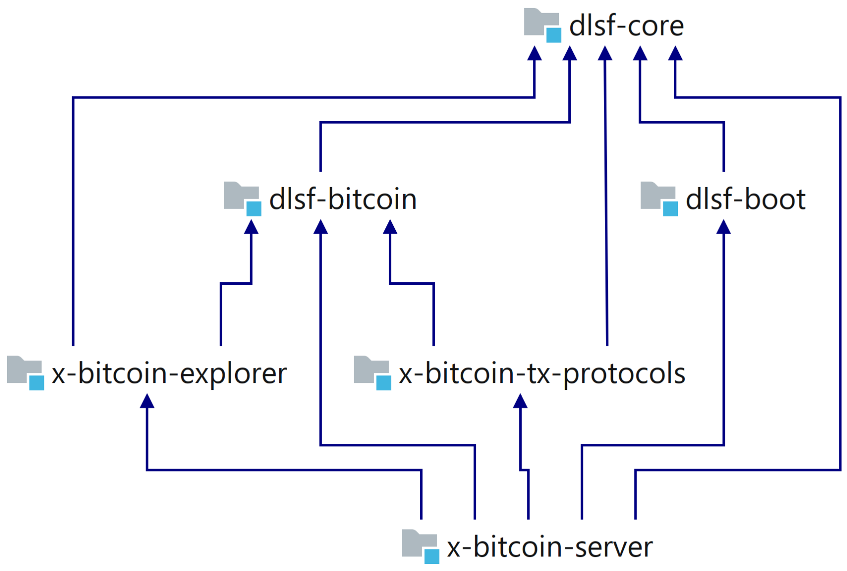


Figure A.1: DLSF project package dependency diagram.

Short descriptions of each module are as follows:

- **dlsf-core:** DLSF Core module.
- **dlsf-bitcoin:** DLSF Bitcoin module.

- **dlsf-boot:** DLSF Boot module.
- **x-bitcoin-explorer:** Bitcoin explorer simulation's module.
- **x-bitcoin-tx-protocols:** Bitcoin transaction protocol simulations' module. It includes both Flood and Erlay simulations.
- **x-bitcoin-server:** This module is used to bootstrap the whole application. It includes all the example simulations and utilizes DLSF boot module.
- **x-bitcoin-client:** This module folder contains the web client's code. Since it is independent of all the Java packages, it is not shown on the dependency diagram.

Bibliography

[1] M. Wander. (2013) Data structure of blocks in the ledger. Accessed: 2019-09-30. [Online]. Available: https://commons.wikimedia.org/wiki/File:Bitcoin_Block_Data.png

[2] Bitcoin improvement proposal 2. Accessed: 2019-09-25. [Online]. Available: https://github.com/bitcoin/bips/tree/ec8f4b003b7b65797f3eec6488dccde9d88a4bd6/bip-0002

[3] G. Naumenko, G. Maxwell, P. Wuille, S. Fedorova, and I. Beschastnikh, "Bandwidth-efficient transaction relay for bitcoin," *CoRR*, vol. abs/1905.10518, 2019. [Online]. Available: http://arxiv.org/abs/1905.10518

[4] Implementierung simulationssoftware: Schwierigkeiten und learnings bei der implementierung der simulationssoftware. Accessed: 2019-10-05. [Online]. Available: https://www.netidee.at/simulation-different-selfish-mining-strategies-bitcoin/implementierung-simulationssoftware

[5] L. Stoykov, "Vibes: Fast blockchain simulations for large-scale peer-to-peer networks," Master's thesis, Technical University Munich, Munich/Germany, 3 2018.

[6] G. Aylov, "Simulations of scalable autonomous smart contracts on ethereum plasma blockchain," Master's thesis, Technical University Munich, Munich/Germany, 1 2019.

[7] M. R. A. Lathif, "Design and implementation of a simulation framework for blockchain alternatives: A directed acyclic graph based distributed ledger," Master's thesis, Technical University Munich, Munich/Germany, 1 2019.

[8] Bitcoinchain - bitcoin block explorer. Accessed: 2019-11-07. [Online]. Available: https://bitcoinchain.com/block_explorer

[9] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. Accessed: 2019-09-30. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[10] Bitcoin testnet. Accessed: 2019-09-30. [Online]. Available: https://en.bitcoin.it/wiki/Testnet

[11] Ethereum ropsten testnet. Accessed: 2019-09-30. [Online]. Available: https://docs.ethhub.io/using-ethereum/test-networks/

[12] Kovan - stable ethereum public testnet. Accessed: 2019-09-30. [Online]. Available: https://github.com/kovan-testnet/proposal

[13] Truffle suite. ganache - ethereum blockchain emulator. Accessed: 2019-09-30. [Online]. Available: https://truffleframework.com/ganache

[14] Hive - ethereum end-to-end test harness. Accessed: 2019-09-30. [Online]. Available: https://github.com/ethereum/hive

[15] S. Research. Simcoin - a blockchain simulation framework. Accessed: 2019-09-30. [Online]. Available: https://github.com/sbaresearch/simcoin

[16] Bitcoin mainnet. Accessed: 2019-09-30. [Online]. Available: https://bitcoin.org/en/glossary/mainnet

[17] F. Schuessler, "Bitcoin-like blockchain simulation system," Master's thesis, Technical University Munich, Munich/Germany, 9 2018.

[18] A. S. Deshpande, "Design and implementation of an ethereum-like blockchain simulation framework," Master's thesis, Technical University Munich, Munich/Germany, 11 2018.

[19] Akka toolkit. Accessed: 2019-09-30. [Online]. Available: https://akka.io/

[20] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," *IACR Cryptology ePrint Archive*, vol. 2016, p. 555, 2016. [Online]. Available: http://eprint.iacr.org/2016/555

[21] J. Banks, J. Carson, B. Nelson, and D. Nicol, "Discrete-event system simulation," 2001.

[22] R. L. Rivest, "Handbook of theoretical computer science," 1990.

[23] A. Menezes, P. van Oorschot, and S. Vanstone, "Handbook of applied cryptography," 1997.

[24] B. Schneier. (2004) Cryptanalysis of md5 and sha: Time for a new standard. Accessed: 2019-09-22. [Online]. Available: https://www.schneier.com/essays/archives/2004/08/cryptanalysis_of_md5.html

[25] D. Turner. (2016) Major standards and compliance of digital signatures - a world-wide consideration. Accessed: 2019-09-22. [Online]. Available: https://www.cryptomathic.com/news-events/blog/major-standards-and-compliance-of-digital-signatures-a-world-wide-consideration

[26] W. Stallings, *Cryptography and Network Security: Principles and Practice.* Prentice Hall, 1990.

[27] UK Government Chief Scientific Advisor. (2016) Distributed ledger technology: beyond block chain. Accessed: 2019-09-22. [Online]. Available: https://www.gov.uk/government/news/distributed-ledger-technology-beyond-block-chain

[28] S. Popov. (2018) The tangle, iota whitepaper. Accessed: 2019-09-30. [Online]. Available: https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf

[29] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

[30] G. Kamiya. (2019) Bitcoin energy use - mined the gap. Accessed: 2019-09-24. [Online]. Available: https://www.iea.org/newsroom/news/2019/july/bitcoin-energy-use-mined-the-gap.html

[31] Minisketch: an optimized library for bch-based set reconciliation. Accessed: 2019-09-30. [Online]. Available: https://github.com/sipa/minisketch

[32] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. D. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," *SIAM J. Comput.*, vol. 38, no. 1, pp. 97–139, 2008. [Online]. Available: https://doi.org/10.1137/060651380

[33] ns-3 — a discrete-event network simulator for internet systems. Accessed: 2019-09-30. [Online]. Available: https://www.nsnam.org/

[34] Bitcoin simulator - source-code repository. Accessed: 2019-11-05. [Online]. Available: https://github.com/arthurgervais/Bitcoin-Simulator

[35] Litecoin - open source p2p digital currency. Accessed: 2019-09-30. [Online]. Available: https://litecoin.org

[36] Dogecoin. Accessed: 2019-09-30. [Online]. Available: https://dogecoin.com/

[37] Bitcoin simulator - usage configuration parameters and usage instructions. Accessed: 2019-09-30. [Online]. Available: https://arthurgervais.github.io/Bitcoin-Simulator/use_it.html

[38] A. Khare, Y. Huang, H. Doan, M. S. Kanwal, and D. C. R. (Ed.), *Chapter 6: Scaling. In A Fresh Graduate's Guide to Software Development Tools and Technologies.* School of Computing, National University of Singapore, 2011, ch. 6.

[39] Angular framework. Accessed: 2019-09-30. [Online]. Available: https://angular.io/

[40] React - a javascript library for building user interfaces. Accessed: 2019-09-30. [Online]. Available: https://reactjs.org

[41] Stack overflow: 2019 developer survey results. Accessed: 2019-10-05. [Online]. Available: https://insights.stackoverflow.com/survey/2019#technology-_-programming-scripting-and-markup-languages

[42] Eclipse vert.x. Accessed: 2019-09-30. [Online]. Available: https://vertx.io

[43] Akka cluster configuration example. Accessed: 2019-11-07. [Online]. Available: https://doc.akka.io/docs/akka/2.5.14/cluster-usage.html#a-simple-cluster-example