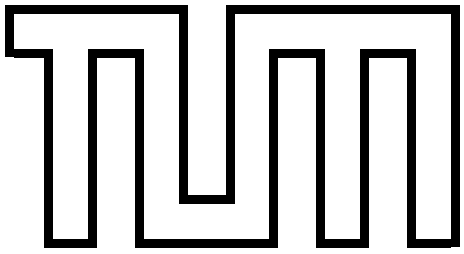


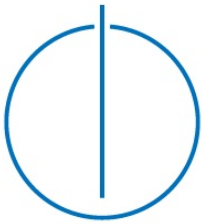
Technische Universität München

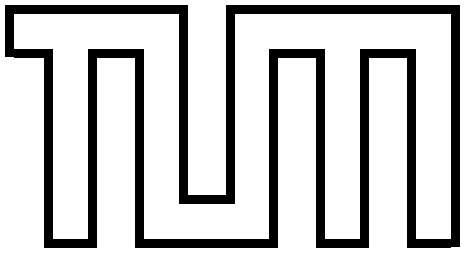
Fakultät für Informatik

Master's Thesis in Informatik

VIBES: Fast Blockchain Simulations for Large-scale Peer-to-Peer Networks

Lyubomir Stoykov





Technische Universität München

Fakultät für Informatik

Master's Thesis in Informatik

VIBES: Fast Blockchain Simulations for Large-scale Peer-to-Peer Networks

VIBES: Schnelle Blockchain Simulationen für große und hochskalierbare Peer-to-Peer Netzwerke

Author: Lyubomir Stoykov

Supervisor: Prof. Dr. Hans-Arno Jacobsen

Advisor: Dr. Kaiwen Zhang

Submission: 14.03.2018

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

München, 14.03.2018

(Lyubomir Stoykov)

Abstract

Following the success of Bitcoin, Ethereum and Hyperledger, blockchains are now gaining widespread adoption in a wide variety of applications, using a diversity of distributed ledger systems with varying characteristics. Yet, properties of such systems are not sufficiently analyzed. To better understand the behavior of these systems, we propose VIBES: a configurable blockchain simulator for large scale peer-to-peer networks. With VIBES users can explore important characteristics and metrics of the network, reason about interactions between nodes, and compare different scenarios in an intuitive way. VIBES differentiates itself from previous works in its ability to simulate not only blocks, but also transactions and run thousands of nodes on a single PC. Also, VIBES is the first simulator designed to be extended to blockchain systems beyond bitcoin.

Inhaltsangabe

Nach dem Erfolg von Bitcoin, Ethereum und Hyperledger finden Blockchains nun breite Anwendung in einer Vielzahl von Anwendungen, denen diverse Distributed-Ledger-Technologien mit unterschiedlichen Eigenschaften zugrunde liegen. Jedoch sind Eigenschaften solcher Systeme nicht ausreichend analysiert worden. Um das Verhalten dieser Systeme besser zu verstehen, schlagen wir VIBES vor: einen konfigurierbaren Blockchain-Simulator für große Peer-to-Peer-Netzwerke. Mit VIBES können Benutzer wichtige Merkmale und Metriken des Netzwerks explorieren, über Interaktionen zwischen Nodes Schlüsse ziehen und verschiedene Szenarien auf intuitive Art und Weise vergleichen. VIBES unterscheidet sich von früheren Werken in seiner Fähigkeit nicht nur Blöcke, sondern auch Transaktionen zu simulieren und Tausende von Nodes auf einem einzigen PC auszuführen. Schließlich ist VIBES der erste Simulator, der auch um Blockchain-Systeme über Bitcoin hinaus erweitbar ist.

Acknowledgment

I would like to thank my advisor Dr. Kaiwen Zhang for actively supporting me with my thesis, providing me with invaluable knowledge and being reachable even at unusual times. Also, I would like to thank my supervisor Prof. Dr. Hans-Arno Jacobsen for giving me the opportunity to write this thesis at the chair of business information systems at TU Munich.

Contents

List of Figures

List of Tables

Abbreviations

P2P	peer-to-peer.
PoS	Proof of Stake.
PoW	Proof of Work.
UI	User Interface.
VIBES	Visualizations of Interactive, Blockchain, Extended Simulations.

Chapter 1

Introduction

Every now and then a wave of innovation forms that redefines the future of technology. Currently the world is riding on the top of another Gartner's hype curve [5] as public attention has finally caught up with the potential of blockchain. Blockchain refers to a decentralized list of records that carry data linked up by cryptography. It is primarily used in peer-to-peer networks to reach consensus between Nodes that do not trust each other.
The exponential increase in popularity recently gained by blockchain has resulted in great variety of new use cases such as Hyperledger (Fabric) [6], IOTA [7], REQ [8] and others.

1.1 Motivation

Unfortunately, due to lack of understanding and appropriate tools, a blockchain network is often seen as a block box where internal processes and relationships between Nodes remain very much unexplored and extremely difficult to reason about. We believe that a configurable simulator can solve this problem. A simulator is a tool that models the behaviour of the real system with minimum information loss, less resources and ideally faster than real-time.

1.2 Problem Statement

At the time of writing this paper the market cap of blockchain-based networks exceeds 500 billion dollars. A huge amount of value circulates in such applications. Yet blockchain's

future remains uncertain due to severe, unsolved problems such as scalability [9].

A simulator would make it easier for developers and the community to resolve those problems, because the implications of an eventual change in the network can be predicted. Therefore, this paper proposes a simulator to help developers, heavy bitcoin users and students gain more insights into a blockchain network, its behaviour and the consequences of a potential change.

There are only few attempts to simulate such applications and most of them fail at large scale. The main challenge lies not so much in the implementation details of a particular blockchain based application, but in the ability to simulate at large scale in a cost-effective, user-friendly manner. As we later explain in Chapter 3, current simulations run on huge clusters consisting of multiple powerful machines and are able to analyze only about 50 Nodes [10]. There is one single simulator that can scale beyond hundred of Nodes, but it does not implement transactions in the network, only blocks [11].

Our goal was to enable fast, scalable and configurable network simulations on a single computer without any additional resources and attention to user experience. User experience refers to the ability to easily understand and efficiently use the end product.

To evaluate our approach we rely on six very important metrics as follows - speed, scalability, flexibility, extensibility, correctness and powerful visuals.

- Speed refers to the ability to simulate hundreds of Nodes on an average laptop and thousands of Nodes on a high-end powerful personal computer.
- Scalability is defined by the curve of speed decrease as we increase number of Nodes, transactions or neighbours. Ideally, the curve should be linear.
- Flexibility marks out the number of parameters a simple simulation takes. We want to be able to include at least important characteristics such as block size, number of neighbours, number of Nodes, number of transactions per block, etc.
- Extensibility describes the ability from developer's perspective to use the simulator as a framework to implement more concrete use cases simply by building his use case on top of what VIBES provides.
- Correctness should be evaluated by empirical analysis of the behaviour of the

simulator, ideally backed up by theoretical proofs.

- Powerful visuals refers to an attempt to design a beautifully crafted user interface that provides insights via visual analysis. Powerful visuals can also be used by beginners as introduction to the difficult to grasp abstractions and internal processes of a blockchain network.

To this end, we envision a generic blockchain simulator that is fast, scalable, configurable and provides the base for developers to implement more concrete use cases on top. The visual side of VIBES is directed as much to heavy users and developers as to students and beginners who want to be introduced to this complex topic.

1.3 Approach

To address the presented objectives, we first and foremost made sure to fast-forward the whole network ahead of time and skip heavy computations such as solving a block as described in Chapter 4. A so called Coordinator took the role of an application-level scheduler to make sure this is possible. Moreover, we used appropriate tools and frameworks to achieve simulation at event level via the actor model. The deployed approach allowed us to span multiple processor cores and threads so that maximum utilization of CPU was achieved. For the visual side of the simulator the pattern of Atomic Design [12] was used to build a high-quality, composable UI.

1.4 Contribution

The current way of running blockchain simulation is by emulating a whole network with all its implementation details such as block mining, sockets and TCP connections in an unscalable fashion. The most important contribution of this thesis is the novel attempt at an implementation of a blockchain simulator that replicates the network at an event level. This enables the removal of arguably unimportant, but very expensive and complex implementation details such as explicitly solving a block or maintaining a socket connection alive. In this regard, a proposal for an application-level scheduler to forward the network ahead of time has been made and initial implementation provided. The scheduler makes sure Nodes skip heavy computations and the whole network moves faster and with less resources.

1.5 Organization

The thesis is outlined as follows: In the next chapter theoretical knowledge has been covered that provides the needed insights to understand the rest of the thesis. In Chapter 3 related work is discussed to put this paper in perspective. After that, VIBES' architecture and design have been explained. Additionally, in Chapter 4, we dive into implementation details that should serve either as a reference for developers or to reason about non-obvious technical decisions that have been made. In Chapter 5 we evaluate VIBES according to six criteria using combination of empirical and theoretical analysis. In the last section, we present our conclusion and make suggestions for future work.

Chapter 2

Background

The following section lays down the theoretical foundation needed to understand the rest of this thesis. We'll explain what blockchain actually means, what problems it solves and also dive deep into the internals of bitcoin, because a big part of the simulator is highly inspired by bitcoin.

2.1 Emulation

An emulated system reproduces the exact same behaviour as the real system it emulates, but in a different environment. An emulation has the exact same external, observable behaviour as a real system and usually adheres to all rules of its real counterpart.

2.2 Simulation

A simulation mimics the basic behaviour of a real system, but does not comply to all its details. A simulation is an abstraction, a model that captures the most important properties of its real counterpart. Therefore, a simulated system behaves similarly to a real one, but might be implemented differently.

2.3 Peer-to-Peer Network

A definition of a P2P network has been provided in the paper *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications* as follows:

A distributed network architecture may be called a Peer-to-Peer(P-to-P, P2P) network, if the participants share a part of their own hardware resources (processing power, storage capacity, network link capacity, printers). These shared resources are necessary to provide the Service and content offered by the network... [13]

2.3.1 Network Topology

Network topology defines the way Nodes are structured in the network. Physical topology is concerned with how the systems (Nodes) are physically connected through actual cables, work stations and servers that transmit data. Logical topology refers to the way communication (logically) flows between devices (Nodes) [14].

Unstructured Network

In an unstructured network there are no rules imposed about how Nodes are structured. The structure of the network is arbitrary.

Structured Network

In a structured peer-to-peer network Nodes adhere to rules to form a particular structure. For instance, Nodes might be ordered in a way to ensure efficient search in the network.

2.3.2 Gossip Protocol

The term refers to a communication protocol that defines the way messages are broadcast in a peer-to-peer network. The protocol has to be efficient so that information is propagated quickly without flooding the network.

It is beyond the scope of this thesis to describe any gossip protocol in detail, but in Section 2.7.8 we'll explain how gossiping (information propagation) in bitcoin works.

2.4 Cryptography

Cryptography is the discipline of securing information exchange between parties from adversaries (malicious entities). In the context of blockchains, it is used to protect data from modification and prove ownership.

2.4.1 (Cryptographic) Hash Function

A (cryptographic) hash function refers to a deterministic function that compresses a string of arbitrary input to a string of fixed length [15], where it is nearly impossible to infer the initial input from the output. The output of a hash function is called a hash. Different input may result in the same hash, but the probability for that is extremely low in a "collision resistant" cryptographic hash function.

2.4.2 Encryption

The process of converting information from plain text into encoded format that is inaccessible or meaningless to unauthorized parties.

2.4.3 Decryption

The inverse of encryption - the process of converting encrypted information into plain text by authorized parties.

2.4.4 Digital Signature

A digital signature is a mathematical scheme which ensures that a message has not been modified (integrity) and it was created by a known author (authentication). Digital signature also makes sure that the author can not deny his ownership (non-repudiation) [16].

2.4.5 Public and Private Key

Public and private keys are used to encrypt and decrypt information. Information encrypted with public key can only be decrypted by the corresponding private key and vica-versa. Usually, the private key is held secret and used to prove ownership of a message. The public key is then used by other parties to assert the ownership of a message encrypted by the private key. These keys can also be used to secure information exchange. Moreover, public and private keys serve to validate and create digital signatures.

2.5 Distributed Ledger

To explain what blockchain is we would need to refer to the term (distributed) ledger. A distributed ledger is essentially an asset database that can be shared across a network of multiple sites, geographies or institutions. All participants within a network have their own copy of the ledger [17].

2.6 Blockchain

Blockchain is a distributed ledger constructed out of blocks that carry data, linked up by cryptography. Cryptographic techniques are used to assert, among others, ownership and data integrity. Visualization of a blockchain can be seen on Figure 2.1.

Each Node in the network has its own copy of the distributed ledger and the longest chain in the network (the one with most blocks) is the ultimate source of truth.

Nowadays many applications use blockchain in a peer-to-peer network to achieve consensus (with very high probability) between Nodes that do not trust each other.

It is difficult to claim where the term originally came from, but it is believed [18] to have been popularized by Satoshi Nakamoto in his paper *Bitcoin: A Peer-to-Peer Electronic Cash System*. Since a big part of the simulator is inspired by Bitcon's protocol, we'll explain it in detail in the next section.

2.7 Bitcoin

If not specified otherwise, the explanations here are largely based on bitcoin's wiki [2], bitcoin's initial protocol as introduced by Nakamoto [1], the paper *Information Propagation in Bitcoin Network* [19] and bitcoin's developer guide [20].

As defined by Nakamoto, bitcoin is a purely peer-to-peer version of electronic cash that allows online payments to be sent directly from one party to another without going through a financial institution [1]. The digital asset (the currency unit) carried by the network can be referred to as bitcoin as well.

Bitcoin proposes a solution to the double spend problem, where the problem is associated with the ability to spend the same coins (currency units) more than once. The main contribution is that the problem is solved without the involvement of trusted central authorities such as banks.

As already implied, bitcoin is a blockchain-based system. Collectively working peers ensure the trustworthiness of the system by validating transactions and hashing them into an ongoing chain of blocks. The longest chain serves as proof of events witnessed [1].

A large part of VIBES follows bitcoin's protocol, which we will now introduce in more detail.

2.7.1 Block

The blockchain is implemented as a directed tree consisting of blocks. Each block contains transactions and a (hash) reference to the previous block as shown on Figure 2.1. It also contains a unique, difficult to solve mathematical problem.

A new block can only be added to the blockchain if a Node finds solution to this problem. Although the solution is difficult to find, it is easy to verify. While a Node looks for solution to the problem, it can also add new transactions to the block as it receives them from other peers in the network. For clarity, we refer to transactions in a solved block as

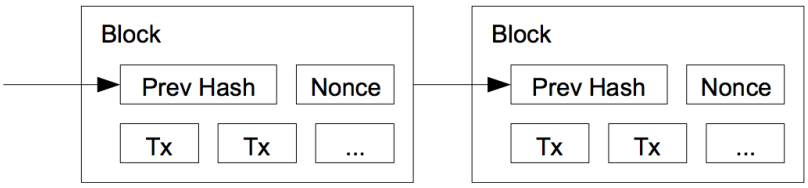


Figure 2.1: Visualization of a blockchain. [1]

committed.

Once a solution has been found, the Node adds the block to its chain and broadcasts it to the network. The block will only be accepted by peers if it comes from a blockchain longer than theirs and carries only valid (not already spent) transactions. When a peer accepts a block, following steps are taken [19]:

- The Node will verify if the block came from the same branch on the blockchain. Remember the blockchain is organized as a directed tree.
  If yes, it will switch the head of its blockchain to the head of the blockchain carried by the received block by retrieving all intermediate blocks on the branch and applying them incrementally.
  If not, the Node will find the first common ancestor shared by both branches. All changes down to this ancestor will be reverted, valid transactions added back to the mempool (introduced later) and then all intermediate blocks from the longer chain applied on top.
- Current uncommitted transactions from the next block (the Node was working on) will be synchronized as well simply by removing the ones included in recently added blocks.

There are many more block-related details, but they are not relevant for understanding the rest of this paper.

Genesis Block

The genesis block refers to the first block included in the blockchain.

Fork

As already mentioned, a blockchain is represented by a directed tree. It is very well possible that different Nodes independently find solution to a block at the same height of the chain within a very short period of time. In this case, the chain will branch out - Nodes will add different blocks on top of a common ancestor, and propagate their block to the network.

Nodes in the network will accept the first received block and, as they keep on extending their chain, eventually one branch will become longer again, carrier of ultimate truth.

Orphan Block

Orphan blocks are valid blocks which are not part of the main chain. In other words, a Node with the longest chain has not seen the previous block an orphan points to. Usually they develop as a result of forks.

2.7.2 Transactions

A transaction transfers value (bitcoins) in the network. Each transaction contains multiple inputs of value from a payer and multiple outputs to payees. The process is visualized on Figure 2.2

A payer includes their digital signature in a transaction to prove it came from him. The network could easily validate the claim by using his public key.

The payer also includes the recipient's public key in the transaction to transfer ownership in a way that is verifiable by the network. The process is visualized on Figure 2.3.

Mempool / Memory Pool / Transactions Pool

The memory pool contains unconfirmed, but validated transactions, some of them candidates for the next block.

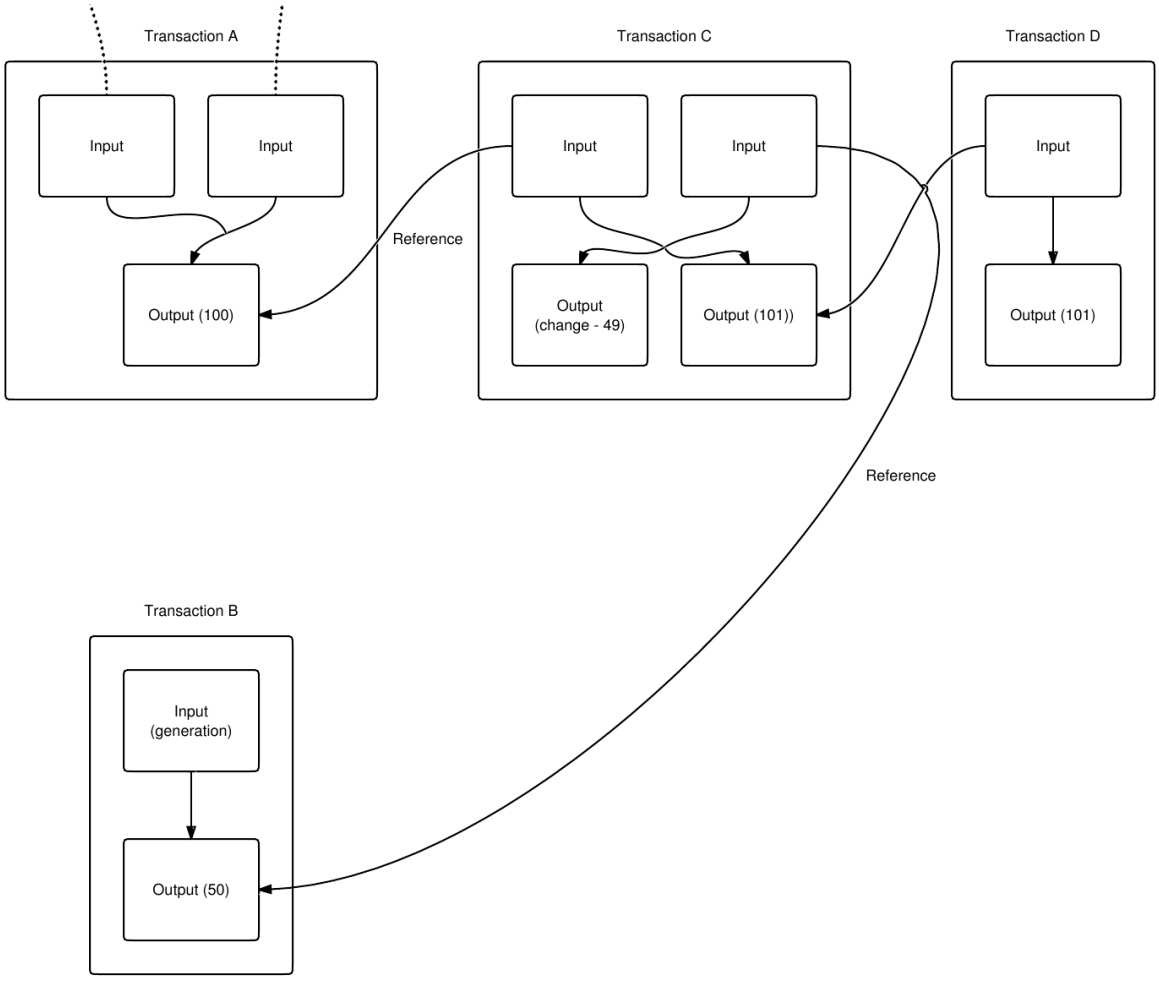


Figure 2.2: A sends 100 BTC to C and C generates 50 BTC. C sends 101 BTC to D, and he needs to send himself some change. D sends the 101 BTC to someone else, but they haven't redeemed it yet. Only D's output and C's change are capable of being spent in the current state. [2]

Confirmation

A transaction is said to be confirmed if it has been committed as part of a block in the current longest blockchain.

Note that a transaction is never "indefinitely" confirmed as valid, because another chain could outgrow the current one and revert it. However, if a transaction was included in the longest chain at least 6 blocks ago, it is believed to have been confirmed (or irreversible) with very high probability.

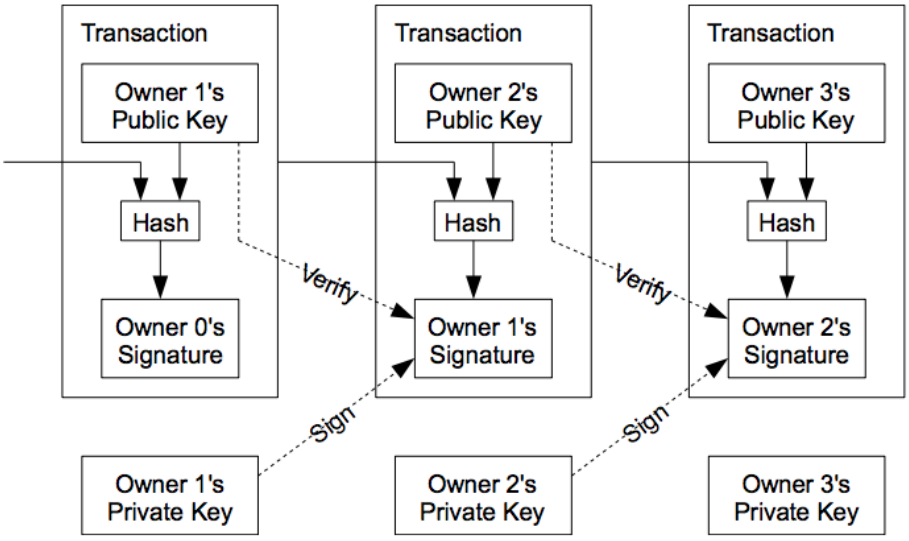


Figure 2.3: Visualization of transactions. [1]

2.7.3 Consensus and Proof of Work

To reach consensus, certain Nodes in the network work on a difficult to solve problem. The process of solving a block is called *mining*. Nodes are trying to find a number that, combined with the block header, results in a hash with given number of leading zeros. This number is called *nonce* and whoever finds it has the right to add its block to the chain. The rest of the network can easily verify the correctness of the solution and extend its chain with the block after it has been broadcast.

The protocol of reaching consensus in this way is called Proof of Work.

Difficulty

The difficulty of the problem, or leading number of zeros, is adjusted dynamically by the network so that a Node finds a solution every 10 minutes on average.

2.7.4 Full Node

Full Node refers to a Node that has a full copy of the distributed ledger.

2.7.5 Miner

Miner is a Node that tries to solve the Proof of Work, or mine a block. Note that a full Node does not have to be a miner, but could be.

Rewards and Incentives

A Node has incentive to mine a block, because it receives a fixed amount of coins for the solution itself and also collects fees from all transactions that are included in the block. The fees are predetermined by the sender of a transaction.

The bigger the fee, the higher the probability a miner will select this transaction as a candidate for the next block.

Mining pool

Mining pool is formed by a group of Nodes that are trying to mine a block collectively.

2.7.6 Smart Contracts

Smart contracts refer to transactions that use the decentralized nature of the network to enforce agreements between Nodes in order to minimize dependency on outside agents, such as the court system [20].

2.7.7 Network Topology

There is not any particular topology behind bitcoin, by construction the Nodes in the network form a random graph [19].

When Nodes bootstrap for the first time, they do not know the addresses of any active full Nodes, so they query hardcoded DNS servers. The DNS servers then respond in return with IP addresses of full Nodes that may accept new connections. Once a Node has discovered several peers, they can also start exchanging information about location of other Nodes in the network, providing a decentralized way of Node discovery [20].

2.7.8 Information Propagation

Block Broadcast

There are three methods to broadcast a block in the network [20].

- Unsolicited Block Push: If a miner knows none of its peers have a block (because it has just been mined), it can directly push it to all of them.
- Standard Block Relay: The relay Node sends an *inv* message to all its peers with a reference to the new block. Depending on the case, the synchronization might require several back-and-forth exchanges, described in bitcoin's developer guide [20], but in its simplest form it looks like this: If any of the recipients wants the block, they send a *getdata* message in response and the relay Node forwards them the block. The *getdata* request might be preceded by a *getheaders* message to avoid any orphan blocks and synchronize the chain faster.
- Direct Headers Announcement: The overhead of an *inv* and *getheaders* message is skipped by simply sending *headers* message containing most recent (up to 2000) block headers. The recipient might then request via *getdata* any missing blocks.

The default way to broadcast a block nowadays is via Standard Block Relay. Note that VIBES uses Unsolicited Block Push for simplicity (and because there is technically no network overhead during simulation).

Transaction Broadcast

Transaction broadcasting is much simpler. A simple *inv* message is sent to neighbours and if any of them wants the transaction, they respond with a *getdata* message. In return, the relay Node forwards them the transaction.

2.7.9 Double Spend Attack

Double spend attack is the most famous vulnerability of bitcoin and refers to a situation where an attacker works simultaneously on two copies of the distributed ledger and is able to spend the same coins twice as a result.

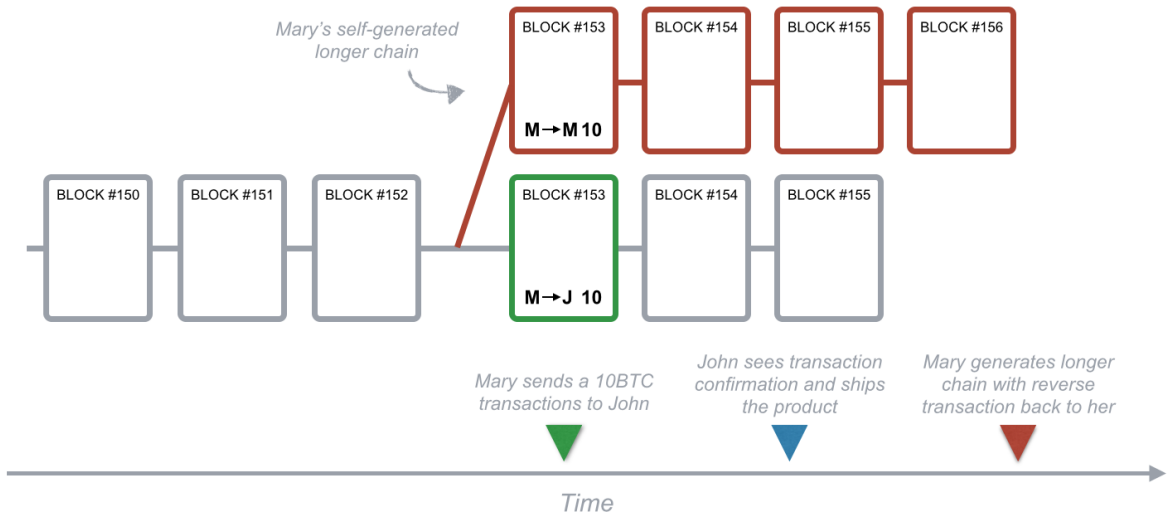


Figure 2.4: Visualization of double spend attack. [3]

Assume an attacker has two copies of the chain, spends money from transaction $T1$ to purchase a car, solves a block, and broadcasts the first copy in the network.

Meanwhile, the attacker keeps working on extending his second copy, where he spends the same money from $T1$ to purchase a boat, but does not propagate the second chain to the network yet.

Once $T1$ from the first chain has been labeled as confirmed with very high probability (for instance, after 6 blocks), the attacker can receive his car.

At this point, the attacker decides to broadcast the second chain to the network, where he spent money from $T1$ for the second time.

If the second chain now is longer than the first one, the transaction where he spent the money for a car will be rendered invalid and he'll be able to use the same money for the second time to purchase the boat.

In order to attack the network in this way, the attacker must have been faster at solving blocks than all other Nodes combined. This would require about 51% of the computational power in the network. Another example of double spend attack is shown on Figure 2.4.

2.8 Proof of Stake

Proof of Stake (PoS), similarly to PoW, is a consensus algorithm to reach consensus between Nodes in a blockchain-based network.

Proof of Stake's consensus is based on a validator's stake in the network. Each Node could become a validator by sending a special type of transaction that locks up their currency into a deposit. In comparison to Proof of Work, where Nodes conceptually "vote" with their CPU power, in Proof of Stake validators vote on the next block based on their current stake (deposit) in the network, which, among others, reduces energy consumption [21].

Chapter 3

Related Work

In efforts to better position this paper and put our simulator in perspective, in this section we will have a look at related work.

3.1 Bitcoin-Simulator

To my best knowledge, the first ever blockchain simulator, called Bitcoin-Simulator, has been proposed in the paper *On the Security and Performance of Proof of Work Blockchains* [11]. However, there are several crucial differences between VIBES and Bitcoin-Simulator as follows:

- First and foremost, Bitcoin-Simulator only simulates the network at block level. This means that, in contrast to VIBES, no transactions are considered there.
- Bitcoin-Simulator is based on NS-3, a discrete event-driven simulator [22], to emulate a network and span nodes that communicate between each other. VIBES is based on Akka and does not emulate a network, each Node is presented as an Actor. Conceptually, both simulators have completely different architecture and design. In Section 3.2 we'll explain in more detail how NS-3 compares to VIBES.
- Bitcoin-Simulator does not provide UI. We put a lot of effort into VIBES to provide a good user interface and enable visual analysis.
- Although we have strived to create a more generic version of a blockchain simulator,

it is clear that both simulators are largely inspired by bitcoin's protocol.

Unfortunately, it is difficult to claim anything about speed or scalability of the Bitcoin-Simulator since I was not able to install it. However, the paper makes clear that the number of simulated nodes is capped at 6000, something that VIBES does not restrict.

I'm not aware of any other simulator that comes close to Bitcoin-Simulator or VIBES.

3.2 P2P Network Emulators

In this section a comparison between P2P network emulators and VIBES is done. But before we can do that, more context is needed.

Introduction and Context

Although a P2P Network emulator does not simulate a blockchain network, it can be used as a solid basis to build upon. There are many different P2P emulators such as NS-3 [22], PeerSim [23], OMNeT++ [24], OPNET [25] and others.

Since I had limited time, I decided to research which one is preferred by the community and present it here. Based on my research [26], NS-3 was the one.

NS-3 is a very powerful emulator, able to simulate a peer-to-peer system at the network layer of the OSI Model. NS-3 is written in C++. Being so powerful and fine-grained, NS-3 seemed too complex for our use case. A blockchain simulator, as VIBES has shown, does not have to emulate the system at the network layer to achieve reasonable accuracy.

NS-3 seemed very sophisticated, with difficult to navigate documentation and enterprise spirit. It is a viable option only for people that have a prior experience with it, because of its complexity. Moreover, a simulator build on top of NS-3, would be hardly accessible for the day-to-day developer that does not have domain specific knowledge in NS-3.

At the time I was exploring NS-3, I already had the initial concept of VIBES in mind and decided to implement it.

From this point on, I started working on VIBES and forgot about NS-3 until later when I asked myself how scalable and fast NS-3 actually is. How does it deal with problems I came across while implementing VIBES. Does NS-3 work in real time? If simulating an event in the future, would NS-3 wait until time passes? Would it do something different?

Since I do not have the knowledge to span a network in NS-3 and analyze its performance, I started reading the documentation and manual to find answers to my questions. Then I came across the following paragraph:

ns-3 is a discrete-event network simulator. Conceptually, the simulator keeps track of a number of events that are scheduled to execute at a specified simulation time. The job of the simulator is to execute the events in sequential time order. Once the completion of an event occurs, the simulator will move to the next event (or will exit if there are no more events in the event queue). If, for example, an event scheduled for simulation time "100 seconds" is executed, and the next event is not scheduled until "200 seconds", the simulator will immediately jump from 100 seconds to 200 seconds (of simulation time) to execute the next event. This is what is meant by "discrete-event" simulator. [27]

As you will later find out, VIBES proposes a very similar concept that we call fast-forward computing and the event execution (in VIBES an event is represented by an executable) is managed by a so called Coordinator, which plays the role of an application-level scheduler.

VIBES and P2P Emulators - differences

Now that enough context has been provided, we can highlight the differences between VIBES and NS-3.

As already mentioned, the initial implementation of our simulator, explained in Section 4.3.6, works very similarly to NS-3's discrete-event approach. After we have implemented this solution, we started testing and came across a bottleneck. The problem was that only a single scheduled event could be executed at a time before we move onto the next one. To cite NS-3 once again:

Once the completion of an event occurs, the simulator will move to the next event [27]

This behaviour is blocking. While one event is being processed, every other event has to wait. In a blockchain based network this becomes problematic, because we have

many scheduled events within an extremely short period of time (think about Nodes propagating transactions).

Assume we have $event1$ scheduled at $t1$ and $event2$ scheduled at time $t2$. In this scenario, it might even be the case that the execution of $event1$ is slower than the difference in time $|t1 - t2|$. This means that the simulator will move slower than real time, because events occur with extremely short period of time between them and are executed one by one, sequentially.

In NS-3 there is nothing we can do about it, because the scheduler is provided by the framework itself. In VIBES, however, we have control over the scheduler (Coordinator). Moreover, we have application specific knowledge. As it is explained later in Section 4.3.7, we can use that to our advantage. The Coordinator can execute multiple events simultaneously whenever appropriate and thus improve the blocking behaviour.

Bitcoin-Simulator, NS-3 Scheduler and VIBES

I can only speculate here, but it now makes sense that Bitcoin-Simulator does not implement transactions in the network. Transactions in the network lead to multiple events with very short period of time between them. Combined with the blocking scheduler of NS-3, the simulation will be rather slow. VIBES improves on that and can simulate fast transactions in the network. A good opportunity for future work is to directly compare the amount of events NS-3 and VIBES are able to process within certain amount of time.

3.3 Testnet

Although not directly comparable to VIBES, it is worth mentioning other way to explore a blockchain network.

As for bitcoin specifically, an obvious solution would be to use Testnet [28] - an alternative bitcoin block chain used for testing. Nevertheless, this approach does not scale, it is very expensive and also impractical since it bootstraps real Nodes in a test environment.

A Testnet simulations have been used in *BLOCKBENCH: A Framework for Analyzing Private Blockchains* [10] to analyze blockchains. The setup there was as follows:

The experiments were run on a 48-node commodity cluster. Each node has an E5-1650 3.5GHz CPU, 32GB RAM, 2TB hard drive, running Ubuntu 14.04 Trusty, and connected to the other nodes via 1GB switch. [10]

Nevertheless, only 32 Nodes in the network have been simulated. On the other hand, VIBES can easily simulate over 1000 Nodes on a single PC with similar capacity.

3.4 Back of the Envelope Approach

The last way to explore a blockchain network would be to perform a back of the envelope approach, using empirical data to infer certain properties of the network as presented in *On scaling decentralized blockchains* [29] or *Information Propagation in the Bitcoin Network* [19].

This approach deviates significantly from the idea of a configurable simulator, but is worth mentioning for the fact that one can at least analyse properties of the network. Needles to say, it is a very expensive, non-configurable and time consuming way to explore properties of blockchain.

Chapter 4

Approach

Before we present the architecture of VIBES, let's look at some of the requirements we have. The simulator has to be able to run hundreds of Nodes concurrently, constantly process data and be very responsive. To address concurrency, we need multi-threaded environment. Although the simulator can run on a single computer, Nodes should ideally make no assumptions about shared memory and behave as if in a distributed system. Thus, a node's state should be completely isolated from the rest of the world and only the node itself is allowed to alter its state. In a distributed type of setting, we're looking to minimize side effects.

Also, we want to simulate the network at event level, which means VIBES should be message-driven. In order to meet those requirements the simulator is build as a reactive system - responsive, message-driven and capable of producing a real-time feel as introduced in the Reactive Manifesto [30].

4.1 Prerequisites

4.1.1 The Actor Model

VIBES makes use of the Actor Model, originally described by Carl Hewitt [31]. An Actor is a computational entity and the primary unit of concurrency in the model. Actors, just as like any distributed system, communicate via messages, concurrently and without affecting each other's internal state. No shared state means no locks and message-first

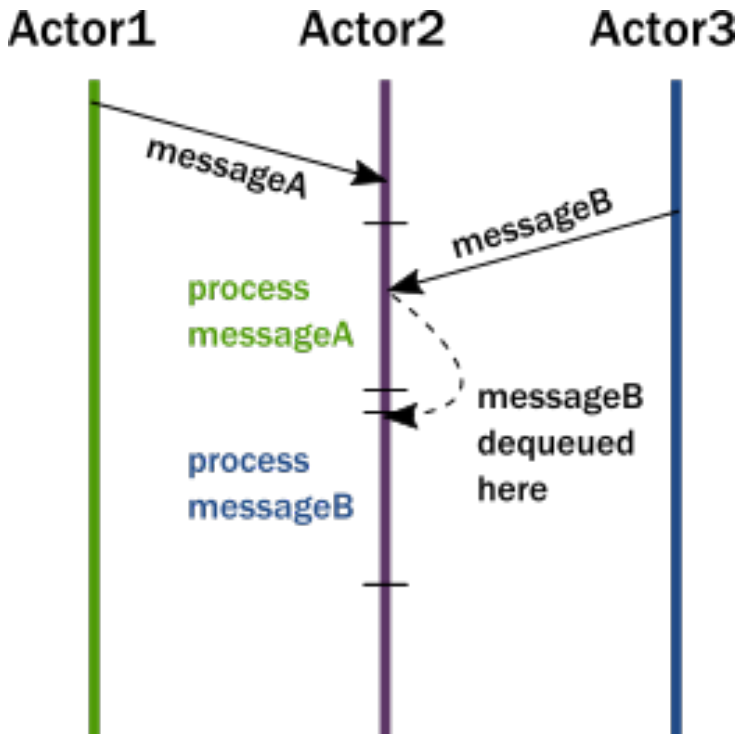


Figure 4.1: The Actor Model - Message Diagram [4]

communication means no blocking. A simple communication diagram between actors is shown on Figures 4.1 and 4.2.

4.1.2 Executables and Work Requests

In this paper, we use the terms an executable and a work request interchangeably. They refer to a piece of computational unit that a Node wants to execute, but needs permission to do so. The permission is granted by a later introduced Coordinator.

4.1.3 Best Guess

A best guess is a timestamped executable that a Node wants to execute at the time carried by the timestamp. For instance, if an *executable* is to mine a block and the timestamp ts, a best guess is the tuple $(ts, executable)$.

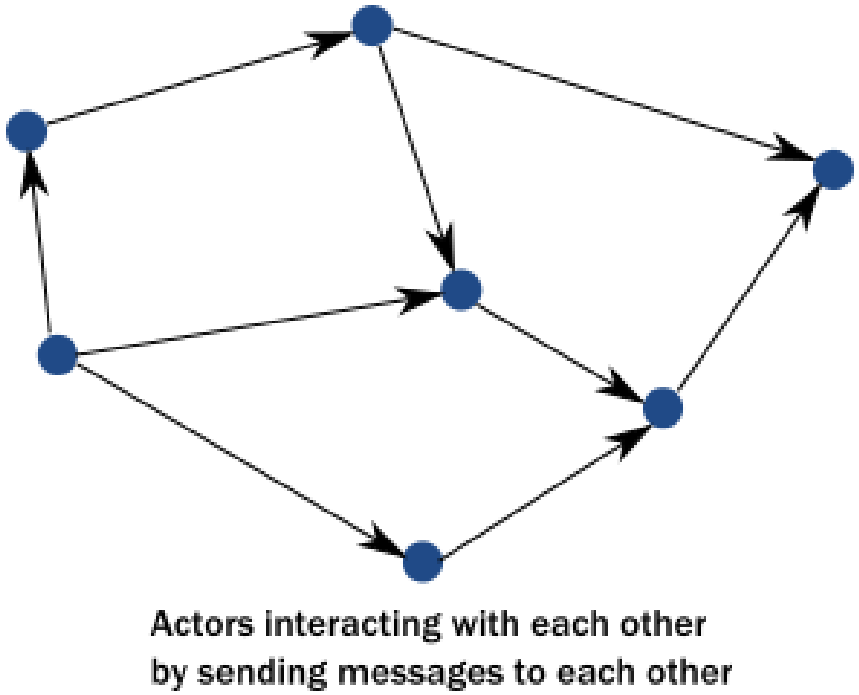


Figure 4.2: The Actor Model - Interaction [4]

4.1.4 Fast-forward

Fast-forward refers to the term of executing a work request in the future with timestamp ts where $ts > now$. At this point, the whole system moves forward to the point in time ts.

4.1.5 Priority Queue

A Node builds a Priority Queue of best guesses - timestamped work requests sorted by the timestamp, meaning that the earliest executable is always on top. We briefly mentioned the Coordinator - it also has a Priority Queue. The Priority Queue of the Coordinator contains the top (first) elements from the Priority Queues of all Nodes. This implies that the top element in the Priority Queue of the Coordinator is the best guess with the earliest timestamp across all Nodes.

4.1.6 Votes

A Node "votes" with the best guess from the top of its priority queue (the executable with earliest timestamp). The "vote" is received by the Coordinator and the best guess added in the Coordinator's priority queue. The Coordinator's priority queue contains only a single vote / best guess from each Node in the network.

4.1.7 Executable Types

Each executable has a type, defined as follows:

- MineBlock - a Node's executable (work request) that will solve the current block and add it to the blockchain.
- IssueTransaction - a Node's executable that will create a transaction.
- PropagateTransaction - a Node's executable that will propagate transaction to its neighbours.
- PropagateOwnBlock / PropagateExternalBlock - a Node's executable that will propagate a block to its neighbours.

```
object ExecutionType extends Enumeration {
    val MineBlock, PropagateOwnBlock, PropagateExternalBlock,
     ↪  IssueTransaction, PropagateTransaction = Value
}
```

4.1.8 Configuration parameters

- $numberOfNodes$: number of nodes
- $blockTime$: expected time to solve a block
- $neighbourDiscoveryInterval$: the interval at which nodes update their neighbour tables.
- $numberOfNeighbours$: number of open connections from a node to other peers in the network
- $throughput$ - expected number of transactions in a block

- *transactionSize* - size of transaction in KB
- *propagationDelay* - transmission + verification time of a block
- *simulateUntil* - the moment in the future where simulation stops

These parameters can be improved as described in Chapter 6. For instance, *propagationDelay* should ideally be dependant and computed based on block's size, distance and network bandwidth. Currently it is used as means to summarize those three parameters in one and therefore we have sacrificed accuracy for simplicity. Moreover, there is no parameter to capture maximal block size.

4.2 Design and Architecture of the Solution

The solution implements each Node in the network as an Actor. Each Actor follows a very simple protocol that replicates the behaviour in a blockchain network, as described in Chapter 2. Once the simulation has come to an end, another actor, called Reducer, takes the network's state as an input and returns an output in a format convenient for the client to process. The output is delivered to the client via an http module. An architecture of VIBES is presented on Figure 4.3.

Now there is one missing piece to the architecture we haven't yet explained, called the Coordinator. The Coordinator is an Actor as well and helps to address one of the most important aspects of VIBES - scalability and speed. To this end, we propose the concept of fast-forward computing via an application-level scheduler called Coordinator. At a higher level, the concept works like this:

Firstly, Nodes build their priority queue of executables by making best guesses about how much time a potential executable would take. In the next step, Nodes vote with the first executable in their queue and ask the Coordinator to fast-forward the whole network at this point in time. Once the Coordinator has collected all votes, it issues timestamped permissions to some of the Nodes so they can label their executable as complete and fast-forward ahead of time carried by the timestamp. In this way, Nodes are able to skip heavy computations and the whole network moves much faster forward. Moreover, the Coordinator has an overview of the global state of the system and is therefore able to prevent conflicts between Nodes and guarantee the order of execution between processes.

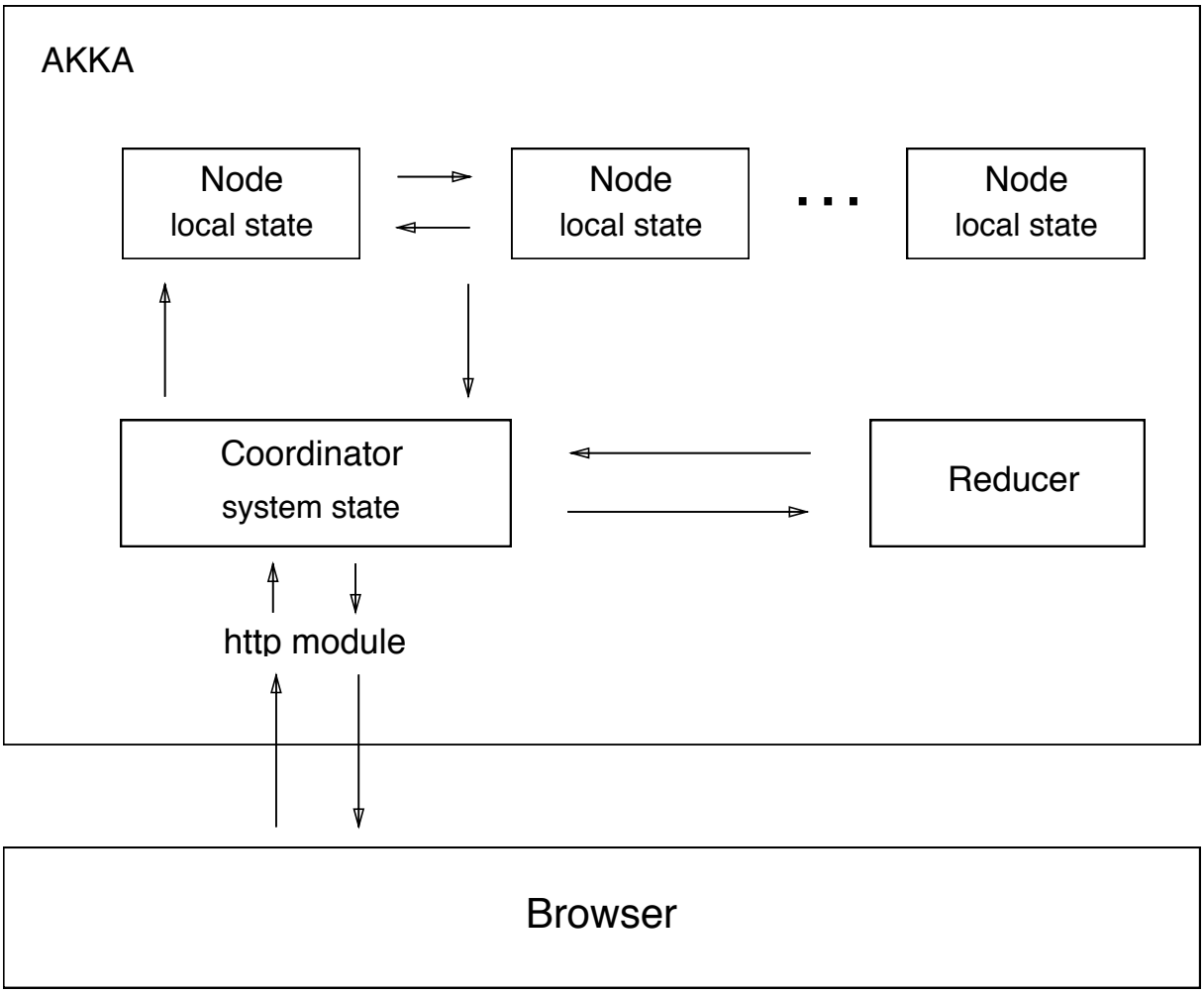


Figure 4.3: VIBES' Architecture

The process has been visualized on Figure 4.4.

Here is a high-level example to make things clearer. Suppose we wish to simulate a blockchain system where miners resolve consensus using Proof-of-Work and a block time of 10 minutes similar to bitcoin. Without the Coordinator, the simulation would require on average 10 minutes until a solution of the next block has been found. Instead, VIBES fast-forwards in the following manner: each Node makes a best guess on the time to complete the operation, then sends this information to the Coordinator, and asks for permission to skip ahead and label the block as mined. The Coordinator then waits until it receives the information from every node and retains the earliest timestamp ts (to finish Proof-of-Work) received. The Coordinator then gives permission to the node with the earliest timestamp ts to mine the block, assigns ts as the time of the mining operation, and fast-forwards the entire network to time ts. All Nodes make then their

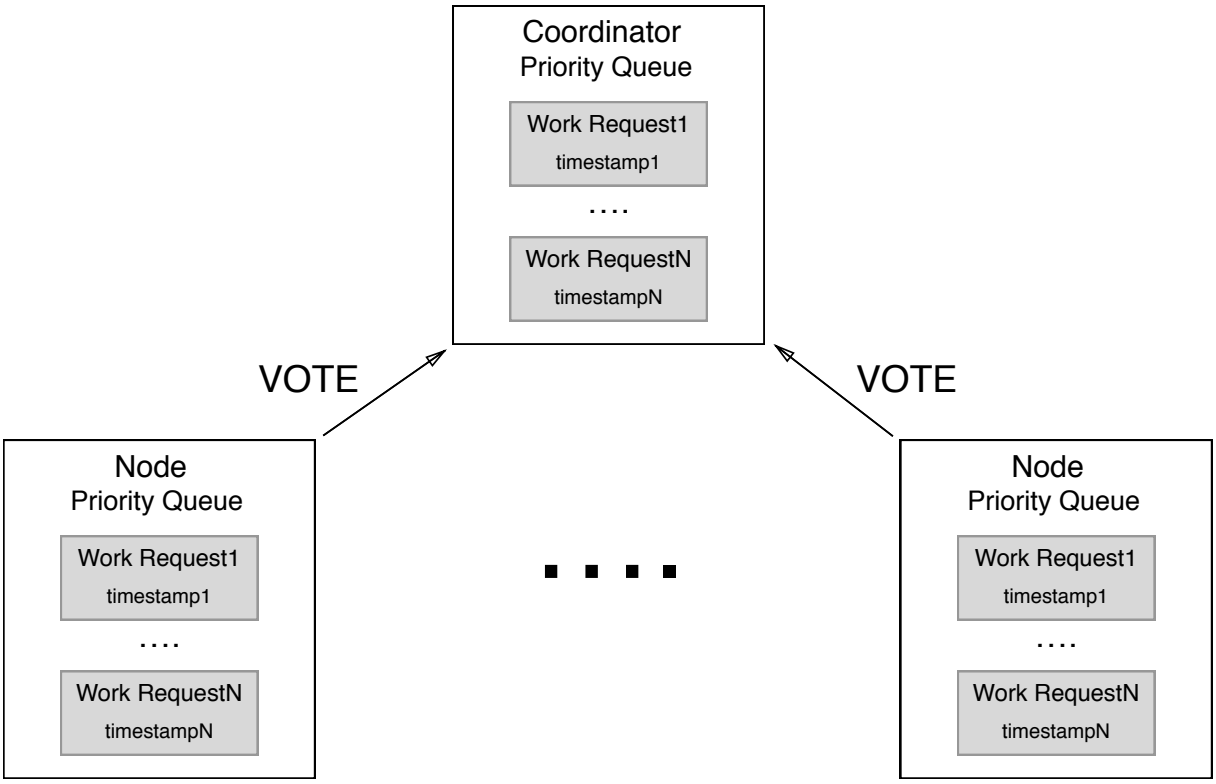


Figure 4.4: Nodes voting to fast-forward

new best guesses and from this point on, the system works recursively until Nodes have finished their work. We encourage the reader to have a look at the Appendix of this paper where the source code of the Coordinator has been presented with a lot of comments.

4.3 Implementation

In this section, we'll explain the solution in more detail. First, let's introduce the Actors in our system.

4.3.1 Coordinator (MasterActor)

This actor builds the core of VIBES. The Coordinator controls the execution of events in the network and is the only Actor that could give Node a permission to execute a work request. The Coordinator is an application-level scheduler that ensures correct order of

execution of work requests in the network based on their timestamps.

During a simulation, Nodes in the network vote with their executables. The votes are collected by the Coordinator. After each Node has voted with its best guess, the Coordinator retains the best guess with earliest timestamp and gives the respective Node (where the guess has came from) a permission to execute it. The execution itself is what we refer to as fast-forward computing. Namely, a Node executes a work request in the future based on a best guess how much time it would take.

Where it gets complicated is that the Coordinator might decide to issue permissions for multiple executables from its queue at the same time in order to improve performance.

At this point multiple Nodes would get the right to execute their work request simultaneously. In other words, the Coordinator has granted / assigned work to multiple Nodes, each represented by a Java thread. As a result, Akka's own thread scheduler takes over and the order of execution between the threads is not guaranteed anymore. Since each thread is responsible for each Node's executable and the threads' executions order is not guaranteed, the executables' execution order is not guaranteed either.

To sum up, if the Coordinator issues permissions to Nodes to execute their work requests simultaneously and one work request is represented by the timestamp $t1$ and another by $t2$ with $t1 < t2$, it still may happen that the executable with $t2$ gets priority by Akka to be the first one to run, although $t1$ is the earlier timestamp.

However, the Coordinator makes sure to only issue multiple permissions when the order of execution does not matter, preventing conflicts and being able to parallelise work whenever possible. The concept will become clearer in Subsection 4.3.7.

Also, since the Coordinator is aware of the global clock and the configuration parameters of the network, it takes care to ensure correct throughput (average number of transactions per block). It randomly chooses Nodes and asks them to create a transaction in the network - Nodes only create transactions on request by the Coordinator. In hindsight, I acknowledge that Nodes could create transactions on random based principle themselves without the Coordinator being involved, and would probably have been a better choice. However, the outcome is the same as throughput is still correct and randomly distributed

between actors. The code looks like this:

```
currentNodeActors = Random.shuffle(currentNodeActors)
val actorsVector = currentNodeActors.toVector
// distribute randomly requests to NodeActors to create throughput
 ↪  number of transactions within blockTime
(1 to VConf.throughPut).foreach { index =>
  val randomActorFrom =
   ↪  actorsVector(Random.nextInt(actorsVector.size))
  val randomActorTo = actorsVector(Random.nextInt(actorsVector.size))
  val now = priorityWorkRequest.timestamp
  randomActorFrom ! NodeActions.IssueTransaction(
    randomActorTo,
    // this means that within the blockTime the throughPut number of
     ↪  transactions will be issued
    // by the nodes. Note that block could be mined in less or more
     ↪  than blockTime, therefore this only
    // ensures average number of transactions per mined block and
     ↪  exact number of transactions per
    // blockTime
    now.plusMillis(VConf.blockTime * 1000 / (index + 1))
  )
}
```

To fully understand the Coordinator's job, we'll give an example and go through every single step it takes in Subsections 4.3.6.

4.3.2 Node (NodeActor)

The NodeActor represents both a full Node and a miner in the blockchain network. As described in Chapter 6, it is a good opportunity for future work to differentiate between both.

In VIBES, each NodeActor has its own blockchain, pool of pending transactions (candidates for the next block) and neighbours. It also works to solve the next block in the chain. A Node has its own priority queue of executables and the next solution of

a block by this Node is represented by the first executable of type MineBlock in the queue.

As a reminder, a NodeActor's executables types are either MineBlock, IssueTransaction, PropagateTransaction or PropagateBlock.

A vote is represented by a best guess (timestamped work request) sent to the Coordinator. The NodeActor sends votes to the Coordinator and is only allowed to execute a piece of work once the Coordinator has granted a permission based on the votes.

Moreover, this Actor receives and propagates blocks from other Nodes in the network. If a received block comes from a longer chain, the Actor takes care to follow synchronization steps described in Section 2.7. The Node synchronizes its blockchain, pool of transactions, rolls back any orphan blocks and adds any valid transactions from orphan blocks back to the transaction pool if they are not already included in the chain. Besides blocks, the NodeActor also takes care to propagate and create transactions in the network.

Here is the definition of a Node:

```
class VNode(
  val id: String,
  val actor: ActorRef,
  val blockchain: List[VBlock],
  val transactionPool: Set[VTransaction],
  val neighbourActors: Set[ActorRef],
  val lat: Double,
  val long: Double
)
```

4.3.3 NodeRepo (NodeRepoActor)

This actors is simply a helper to offload work from the Coordinator. The NodeRepoActor serves as a repository for all Nodes in the network and occasionally broadcasts messages addressed to all of them such as: AnnounceSimulationStart and AnnounceSimulationEnd.

More importantly, this Actor takes care to register or instantiate all Nodes in the network with coordinates on land surface only. To achieve land-only instantiation, we have created a list of rectangles represented by lat, lng coordinates on the world's map that are only within land surface and whenever we generate a Node, we pick a rectangle and randomly instantiate the node within the coordinates of the rectangle, as shown:

```
def createCoordinatesOnLand(): (Double, Double) = {
  val coordinate = coordinateLimits(randomBetween(0,
  ↪  coordinateLimits.size - 1))
  (randomBetween(coordinate.latStart, coordinate.latEnd),
  ↪  randomBetween(coordinate.lngStart, coordinate.lngEnd))
}
```

A good opportunity for future work is to enable configuration of Nodes' distribution on the world map, as explained in Chapter 6.

4.3.4 Discovery (DiscoveryActor)

The DiscoveryActor updates Nodes' neigbhour tables on regular intervals, based on the configuration parameter - *neighbourDiscoveryInterval*. The DiscoveryActor simply dispatches a message to each Node with a new set of randomly chosen neighbours every *neighbourDiscoveryInterval* seconds. I believe there is no specific algorithm or heuristic behind the neighbour discovery in a blockchain based network, but even if there is - it is easy to exchange the current randomized one since the DiscoveryActor is the central authority there. Why don't Nodes update their own tables instead of being managed by another Actor? Because I wanted to offload work from them firstly, and secondly, it is now very easy to exchange discovery implementation. The code looks like this:

```
def announceNeighbours(currentNodes: List[VNode], numberOfNeighbours:
↪  Int): Unit = {
    println("Update neighbours table...")
    currentNodes.foreach(node =>
      node.actor ! NodeActions
        .ReceiveNeighbours(discoverNeighbours(currentNodes, node,
        ↪  numberOfNeighbours)))
}
```

```
def discoverNeighbours(currentNodes: List[VNode], node: VNode,
↪ numberOfNeighbours: Int): Set[ActorRef] = {
    val nodes = currentNodes.filter(_ != node)
    Random
        .shuffle(nodes)
        .take(numberOfNeighbours)
        .map(_.actor).toSet
}
```

4.3.5 Reducer (ReducerActor)

Once the simulation is over, the Reducer takes a set of all nodes as input and finds the longest chain in the network (every Node holds a blockchain). The longest chain contains a list of blocks with arrival times and recipients. Each block contains a list of transaction with arrival times and recipients as well. Based on this information it is trivial to compute the final output of the simulator.

The lists of recipient and arrival times are an implementation detail of VIBES that helps us backtrack events in the network, they do not necessarily correspond to any entity in a real blockchain network.

4.3.6 Naive Algorithm

This subsection explains the initial, naive algorithm and goes through an example step by step. We'll use the terms Node and NodeActor interchangeably.

Iteration

We'll refer to an iteration as the timeframe in which all Nodes have cast their votes and the Coordinator has granted permission(s) for the next executable(s).

Initial stage

Once the simulation kicks in, the MasterActor (Coordinator) instantiates its three helpers: The DiscoveryActor, the NodeRepoActor and the ReducerActor. After that, the

NodeRepoActor creates NodeActors with coordinates on land surface only.

Upon instantiation, the NodeActors subscribe to the DiscoveryActor. At this point, NodeRepoActor, MasterActor and DiscoveryActor are aware of each NodeActor and each NodeActor is aware of them as well. Nodes are not yet aware of each other.

Neighbours Discovery

Once every Node has subscribed to the DiscoveryActor and the system is bootstrapped, the DiscoveryActor pushes neighbours to the neighbour tables of each Node at regular intervals every $neighboursDiscoveryInterval$ seconds. The DiscoveryActor simply dispatches a message to Nodes with their new set of neighbours.

The neighbour selection is randomized, but each Node receives as many neighbours as specified by the $numberOfNeighbours$ configuration parameter of the network.

First iteration

At this point, priority queues are empty and an iteration can start. Firstly, Nodes make their best guesses with executables of type MineBlock and add them to their priority queue. The timestamp is based on a probability function with expected minimum value $blockTime$ after $numberOfNodes$ guesses. In other words, after each Node has voted to mine a block, the earliest timestamp (the winner) will have expected value $now + blockTime$.

After that, each Node votes with the first executable in its priority queue and the Coordinator collects the votes. Once the Coordinator has received all votes, it retains the executable with earliest timestamp. This executable would be of type $MineBlock$.

Executable - MineBlock

We'll now explore what happens in the case when executable is of type $MineBlock$.

Remember that due to an unfortunate design decision the Coordinator also takes care to distribute requests to some of the Nodes in the network to create transactions. The number of requests is based on *throughput* - average number of transactions per block. Those requests are dispatched after a block has been mined - there is no particular reason for that timing apart from the fact that it fit well into the current codebase.

In this sense, after the Coordinator has retrieved an executble of type $MineBlock$, it first distributes requests to Nodes to create *throughput* number of transactions with timestamps $ts+t_i$ where $t_i < blockTime$ and ts is the timestamp of the current executable (the timestamp of *now*). Now Nodes, which have received this type of request, add an executable of type $IssueTransaction$ in their priority queues with the corresponding timestamp within $(ts, ts + t)$.

Next, the Coordinator empties its priority queue since new votes have to be collected after this iteration and only then grants permission to the Node to mine the block.

Respectively, the Node gets notified, labels the block it's working on as mined and adds it to its blockchain. Moreover, the Node deletes this work request from its priority queue since it has already been executed.

In the next step, the Node creates new executables of type $PropagateOwnBlock$ based on the parameter *propagationDelay* and adds them to its queue.

The Node now asks the NodeRepoActor to notify all other Nodes in the network that they have to make new best guesses for $MineBlock$. Please recall that the rest of the Nodes still have $MineBlock$ executables in their priority queue, based on the already mentioned probability function. However, the probability function does not guarantee anything about any other guesses other than the minimum one that has expected value of $blockTime$ and has just been executed. So, to generate another best guess for $MineBlock$ with expected value $blockTime$, each Node has to call the probability function once again and revote with the new result.

The ultimate goal here is to once again have another executable of type $MineBlock$ after $blockTime$ has passed since the last block has been solved.

Once Nodes receive the message from the NodeRepoActor to recalculate their guesses, they delete their old executables of type $MineBlock$, generate new ones, add them once again to their priority queue and finally vote. The Coordinator collects once again all new votes and carries on.

Excursus: Probabilty Function and New Best Guesses

Note that the request to make a new best guess once a block has been mined does not directly correlate with functionality in a blockchain based network, it is a consequence of the nature of the probability function.

The Nodes will still "work" on the same block until a new one has arrived, but the expected mining time will dynamically change in order to preserve average of $blockTime$ across the network until a solution has been found.

In essence, this is what happens in any other blockchain based network - expected time to mine a block should be $blockTime$. This is exactly what we simulate as well. The outcome from the simulation is the same, but the means are different.

Executable - IssueTransaction

Remember the Coordinator has just distributed requests for Nodes to create transactions and the next mining will happen only after $blockTime$ on average, since a block has just been mined. This means, the next executable in the Coordinator's queue will most likely be one that creates a transaction ($IssueTransaction$) and we'll assume that.

Now the Coordinator retrieves the executable and notifies the respective Node. However, in contrast to executable of type $MineBlock$, the Coordinator now does not empty its whole queue. The reasons is that in this case Nodes' votes are not affected by the fact that another Node will execute $IssueTransaction$. For $MineBlock$ the other Nodes' queues were affected, because they had to reset their guesses and revote. For $IssueTransaction$ it is sufficient for the Coordinator to simply delete the current (winner) executable from the queue and wait for a new vote from the Node that now has the permission to issue a transaction.

The Node receives the permission, issues a transaction, adds it to the next block it works on and immediately updates its queue with executables of type *PropagateTransaction* to all its neighbours and a hardcoded time for transaction propagation of $150ms$. The number is based on average time for a small TCP packet to propagate through half of the globe, but could be later parametrized.

The Node now votes with its new queue that also contains executables of type *PropagateTransaction*.

Executable - PropagateTransaction

For the sake of the explanation, we'll assume the next fast-forward executable is *PropagateTransaction*. The Coordinator would again notify the respective Node and will additionally delete two executables from its queue: The executable which was permitted and the Node's executable which will receive the propagated transaction.

The Node that has received permission will execute the *PropagateTransaction* work request, fast-forward the network and vote once again.

Now, if the recipient Node hasn't already received this transaction from someone else and is a valid one, it will add it to the pool of transactions for the next block. The Node will add itself to the list of recipients this transaction holds (so we can later backtrack the events in the network, usually in a blockchain based network this list does not exist). Moreover, the Node will also update its queue and add new executables of type *PropagateTransaction* in order to propagate the transaction to its neighbours.

Finally, the recipient Node will cast a new work request from its updated queue and the Coordinator will collect it. After that the Coordinator carries on as before.

Executable - PropagateBlock

The last executables we haven't discussed yet are *PropageteOwnBlock* and *PropagateExternalBlock*. They have the same functionality and could be summarized in *PropagateBlock*.

The process works similarly to *PropagateTransaction* with the difference that the blockchain of the recipient Node is synchronized as described in Section 2.7, most commonly by simply adding the received block on top. The pool of uncommitted transactions is synchronized as well (transactions that are included in newly received blocks are excluded from the pool). During synchronization any orphan blocks are rolled back and their transactions synchronized. Of course, this happens only if the recipient Node has blockchain height lower than the one carried by the propagated block.

Also, the recipient Node would add new executables of type *PropagateExternalBlock* to its priority queue in order to propagate the block to its neighbours as well.

End

The algorithm works until an executable's timestamp exceeds the *simulateUntil* configuration parameter.

4.3.7 Improved Algorithm

As one can notice from the presented communication flow on Figure 4.4 and the naive algorithm explanation, the Coordinator is the bottleneck in the described scenario, because only a single Node at a time is permitted to execute the next work request in their priority queue while all others are blocked.

This results in many simultaneously pending executables that would ideally be executed in parallel as much as possible. It is a very unfortunate scenario if processing of one request blocks all other pending jobs. This could lead to constantly increasing queues and render the system unresponsive.

To leverage concurrency and parallelism we make an assumption: The order of execution of sequential *PropagateTransaction* executables, with no other executable of different type between them, does not matter.

Note that *PropagateTransaction* is the most common work request in a simulation. In the improved algorithm, the Coordinator collects all sequential *PropagateTransaction*

in its priority queue and labels them all as permitted simultaneosly.

This assumption does not render the behaviour of the system incorrect and we'll show that with an example.

Assume the Coordinator has a priority queue with executables such as $PropagateTransaction1$, $PropagateTransaction2$, $MineBlock$, $PropagateTransaction3$. In this case, the Coordinator will collect the sequential $PropagateTransaction1$ and $PropagateTransaction2$ in its queue and send permissions to the respective Nodes to execute them.

As already explained, when the Coordinator gives permission for multiple work requests to be executed, Akka assigns a thread to each involved NodeActor and since Akka's scheduler takes control over the threads, their order of execution is in no way guaranteed.

In this sense, if $PropagateTransaction1$ and $PropagateTransaction2$ have timestamps $t1$, $t2$ with $t1 < t2$ and are both directed to NodeA, NodeA should receive them in order $PropagateTransaction1, PropagateTransaction2$, but due to the simultaneous execution, this is not guaranteed.

Ultimately the order of execution here does not matter, because the transactions will still arrive as candidates for the same (next block's) transaction pool in NodeA *as the one where they should have been candidates for even if they hadn't arrived in reverse order.* Also, the order of execution matters even less if transactions are directed to different Nodes.

As a consequence from this assumption, high levels of concurrency have been achieved and the bottleneck currently is not the Coordinator, but the limited number of cores a CPU has.

To highlight the importance of this decision one might say that, without this performance improvement, sending and receiving of transaction events would vastly decrease the performance of the simulator, because we would not be able to fast-forward so far ahead of time. This is why it is important to "compress" events whenever possible and fast-forward them simultaneously.

4.4 Technology Choice

In this section, we'll look at the technologies and why we have used Scala with Akka instead of, for instance, Java with own Thread implementations.

The reason is that Scala is a powerful programming language that embraces immutability first approach and makes functional concepts accessible. This minimizes side effects, which was one of our requirements.

Akka, on the other hand, gives us multiple advantages: It implements the actor model and formalizes a set of (arguably best) practices around managing threads effectively. It implements scheduler and a thread pool to better utilize concurrency and parallelism. Basically, Akka does the heavy lifting for the developer in regard to thread management.

Also, if we later decide to deploy the application on a cluster, Akka should make it easier for us since the framework has been developed with this goal in mind.

At last, I had great interest in learning how Akka works and decided to experiment.

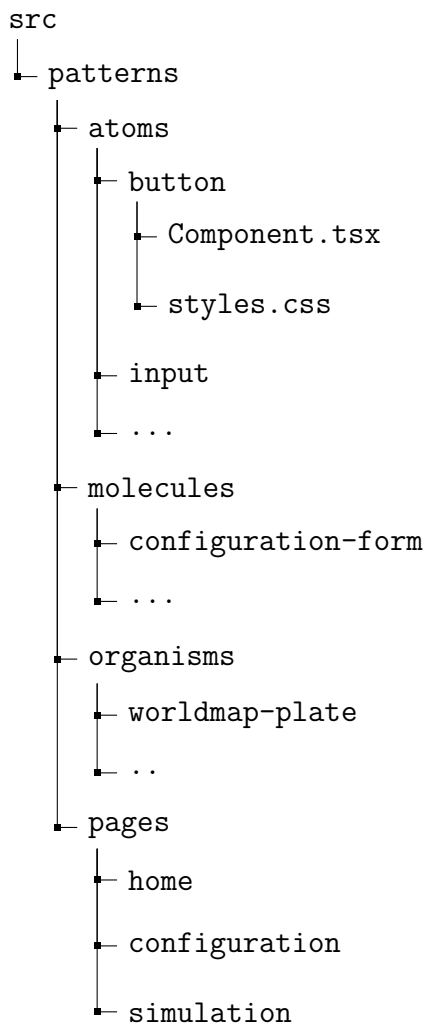
Other code specific details include the marshalling library we have used - circe [32], a code formatter [33] and an http module [34].

4.5 Frontend

The Frontend has been designed in Sketch [35] and implemented using TypeScript [36] as a primary language, React [37] as a View and PostCSS [38] as a stylesheet. The bundle is packaged via Webpack [39] and the code formatted using tslint [40]. For the map visualization we have used DataMaps [41].

The most interesting part here is that our frontend is based on Atomic Design [12] -

philosophy that encourages composition of entities. Essential components such as form inputs and buttons are called *Atoms*. Bigger entities such as forms are called *Molecules* and they are assembled from *Atoms*. Several *Molecules* build up an *Organism* and several build *Organisms* a *Page*. As a result, the directory structure looks like this:



It is important to note that the Frontend currently does not visualize everything captured by VIBES. At the end of a simulation, VIBES carries information about each interaction that has happened in the network. A lot of this information has not yet been analyzed or visualized, because of time constraints.

Our frontend currently displays, due to time constraints, only very basic output such as simulation duration, blockchain length, blockchain size, propagation times, number of nodes received the first and last block. The choice of output is based on information that

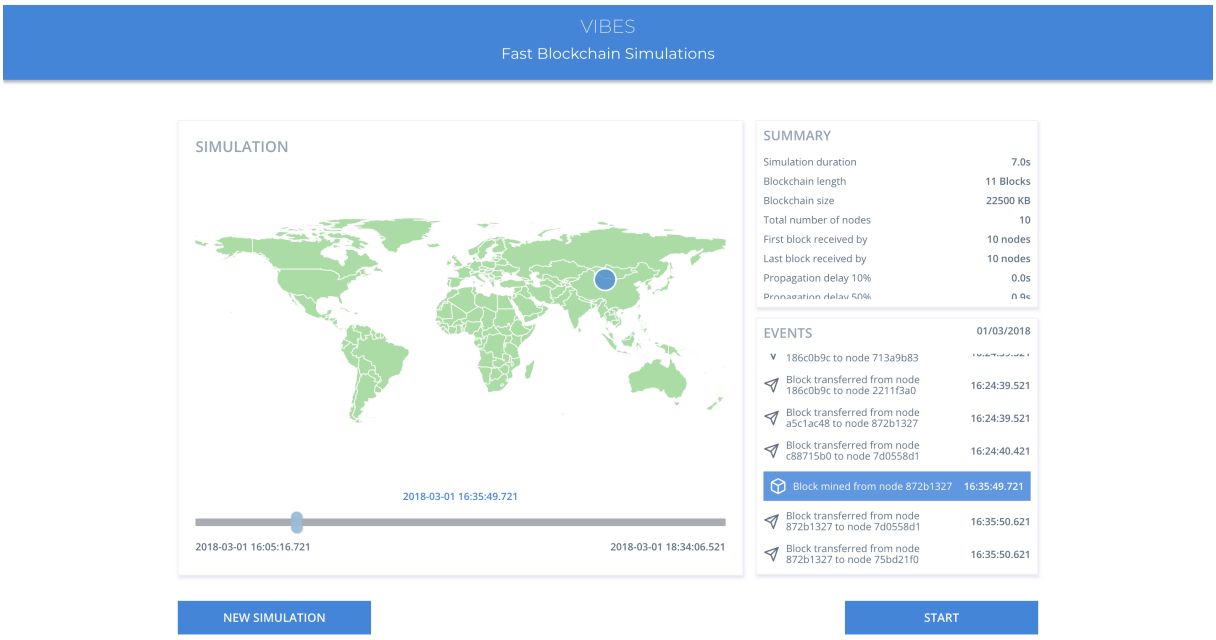


Figure 4.5: Screenshot Visualization

could validate the correctness of the simulation and also on relevant research questions such as block propagation times.

Moreover, the user is able to backtrack and explore all events during the simulation via a world map as shown on the screenshot below.

A great opportunity for future work would be to extend the frontend to display other insights such as probability of forks to occur. This falls under the category of data analysis and is out of scope for this thesis.

Chapter 5

Evaluation

We're going to evaluate the simulator according to 6 criteria: Speed, Scalability, Flexibility, Extensibility, Correctness and Powerful visuals as we defined them in Chapter 1. All tests have been conducted on a MacBook Pro Early 2015, 2.7 Ghz Intel Core i5, 8 GB 1867 MHz DDR3.

5.1 Correctness

To validate the correctness of the simulator, we'll use a combination of formal approach and an empirical analysis. We would like to validate the consistency between the input parameters, expected behaviour and the simulated output.

To this end, we will assert the correctness of: expected and simulated block generation rate, expected and simulated blockchain length, expected and simulated blockchain size, expected and simulated number of transactions, expected and simulated propagation times, expected and simulated number of stale blocks.

5.1.1 Expected and simulated block generation rate

Let's recall we have an input parameter *blockTime* which represents expected time to solve a block. Empirical analysis shows that *blockTime* is extremely consistent with the interval at which blocks are being generated while simulation takes place. This is no

coincidence since we can prove the used formula for the probability function formally as follows:

Let each Node $node_1, node_2, ...node_n$ make a best guess to mine a block noted as $X_1, X_2, ...X_n$, where X_i are uniformly distributed random values and correspond to number of seconds from $currentTime$.

Now we are looking for an a and b such as for $Y = min(X_1, X_2, ...X_n)$ and $X_i \in (a, b)$, $E(Y) = blockTime$. In other words, we want after n best guesses in the range (a, b) to have an expected minimum best guess (noted as $E(Y)$) equal to $blockTime$.

To express $E(Y)$ in terms of a and b we'll need to get the probability density function of Y, $f(y)$. In order to get there, we'll need the cumulative distribution function $F(Y)$. By definition, $F(y) = P(Y \leq y) = 1 - P(Y > y) = 1 - P(min(X_1, X_2, ...X_n))$.

Since X_i are independent and identically distributed random variables $\in (a, b)$, then $F(y) = 1 - P(X_1 > y)^n$ which yields

$$F(y) = \begin{cases} 1 - \left(\frac{b-y}{b-a}\right)^n & : y \in (a, b) \\ 0 & : y < a \\ 1 & : y > b \end{cases}$$

Now we can calculate the probability density function $f(y)$ by taking the derivative of $F(y)$:

$$f(y) = \begin{cases} \frac{n}{b-a}\left(\frac{b-y}{b-a}\right)^{n-1} & : y \in (a, b) \\ 0 & : \text{otherwise} \end{cases}$$

By integrating over $f(y)$ we can calculate that $E(Y) = \frac{b+na}{n+1}$

$Q.E.D$ The proof is based on [42].

Now we simply have to choose a, b such as $blockTime = \frac{b+na}{n+1}$. We can simply assume $a = 1$ and calculate b so that $blockTime = \frac{b+n}{n+1}$. In our code the probability function looks

Run 1	Run 2	Run 3	Run 4	Run 5	Simulated	Expected
20	22	20	15	15	18.4	18

Table 5.1: Simulated and Expected blockchain length

Run 1	Run 2	Run 3	Run 4	Run 5	Simulated	Expected
42 500 kB	39 500 kB	47 500 kB	39 750 kB	50 000 kB	43 850 kB	45 000 kB

Table 5.2: Simulated and Expected blockchain size

like this:

```
    def createTimestampForNextBlock(now: DateTime): DateTime = {
        val numberOfVotes = VConf.numberOfNodes
        val expectedTime  = VConf.blockTime
        // b = blockTime * (n + 1) - n
        val until = expectedTime * (numberOfVotes + 1) - numberOfVotes

        now.plusSeconds(Random.nextInt(until))
    }
```

5.1.2 Expected and simulated blockchain length

Again empirical analysis shows extremely consistent results. To give an example, let's assume we want to simulate 3 hours ahead of time with 10 minutes block generation rate. Given each Node has sufficient number of neighbours the simulator repeatedly outputs average blockchain length of 18 blocks as seen on Table 5.1.

5.1.3 Expected and simulated blockchain size

Let's run the same simulation as above with transaction size 250 kB and average of 10 Transactions per Block (*throughput*). Table 5.2 once again showcases consistent results.

Run 1	Run 2	Run 3	Run 4	Run 5	Simulated	Expected
1200	1799	1900	2584	1389	1774.4	1800

Table 5.3: Expected and simulated number of transactions

5.1.4 Expected and simulated number of transactions

For this experiment we ran the simulations with 100 transactions per block, 3 hours ahead of time and 10 minutes block rate. The expected number of transactions in the longest chain, assuming sufficient neighbours, should roughly be 100 * 3 * 6 = 1800. Table 5.3 proves correct results.

5.1.5 Expected and simulated propagation times

This is extremely difficult to validate property of the network because of the lack of reliable reference results.

Block propagation times have been evaluated in two papers until now. *On scaling decentralized blockchains* [29] and *Information Propagation in the Bitcoin Network* [19] which yield times for 10%, 50% and 90% in the ballpark of 1s, 10s and 60s respectively with number of Nodes anything between 3500 and 10000. The measurements are simply approximation because of the difficulty of the evaluation.

Also, bitcoinstats.com [43], based on *Information Propagation in the Bitcoin Network* [19], shows varying results with anything between 1s and 60s for 2013 - 2014 and 1s - 10s for 2018.

It is out of scope for this thesis to explain the difference in propagation times between 2013 and 2018. A speculative explanation would be that nowadays there are fewer miners as Nodes now are more likely to join a mining pool.

More importantly, recently captured propagation times from bitcoinstats in 2018 are very consistent with VIBES' output. We evaluated 2000 Nodes on the MacBook Pro and results were in the range of 1s, 4s and 7s for 10%, 50% and 90% respectively assuming

enough neighbours and a propagation delay (transmission + verification time) of roughly 1s.

Interestingly enough, we didn't observe exponential decrease in the propagation times as percentage of reached Nodes increased. A theory of mine is that the exponential information distribution in a gossip-based network compensates for the more difficult to reach Nodes at the upper percentages. Most probably, a more complex behaviour must be simulated in order to realistically capture propagation rates in a blockchain based network.

5.1.6 Expected and simulated number of stale blocks

Currently the simulator does not yield any output for the number of stale blocks in the web interface, but forks can be observed in the terminal as VIBES logs the level of each block being mined as shown on Figure 5.1.

Our empirical analysis confirms that the number of stale blocks is, as expected, a function of the propagation delay, block time and number of neighbours. If neighbours, for instance, are not sufficient or the block time is too short, it is very likely that forks occur more often. More forks lead to smaller blockchain height and as a result the probability of double spend attack increases. Therefore, finding the right parameters in the network is essential for security.

5.1.7 Conclusion

Since this is a very complex simulation with a lot of dependency between Nodes, input and output so consistent make us think that the simulator is pretty close to realistically capturing behaviour in the network.

Also, as mentioned in Chapter 4.3, we can observer that CPU utilization slightly decreases for a moment once a Node has mined a block. This is once again consistent with the expected behaviour, because at this point in time only one actor could be active as explained in Chapter 4. Figure 5.2 and Figure 5.3 demonstrate that.

```
BLOCK MINED AT SIZE 0..... 2018-02-28T13:34:06.635+01:00, akka://VSystem/user/Master/NodeRepo/$c
BLOCK MINED AT SIZE 1..... 2018-02-28T13:43:57.635+01:00, akka://VSystem/user/Master/NodeRepo/$d
BLOCK MINED AT SIZE 2..... 2018-02-28T13:53:06.635+01:00, akka://VSystem/user/Master/NodeRepo/$f
BLOCK MINED AT SIZE 3..... 2018-02-28T14:15:39.635+01:00, akka://VSystem/user/Master/NodeRepo/$j
BLOCK MINED AT SIZE 4..... 2018-02-28T14:16:49.635+01:00, akka://VSystem/user/Master/NodeRepo/$h
BLOCK MINED AT SIZE 5..... 2018-02-28T14:18:52.635+01:00, akka://VSystem/user/Master/NodeRepo/$c
Update neighbours table...
BLOCK MINED AT SIZE 6..... 2018-02-28T14:38:58.635+01:00, akka://VSystem/user/Master/NodeRepo/$a
BLOCK MINED AT SIZE 7..... 2018-02-28T15:01:08.635+01:00, akka://VSystem/user/Master/NodeRepo/$g
BLOCK MINED AT SIZE 8..... 2018-02-28T15:01:38.635+01:00, akka://VSystem/user/Master/NodeRepo/$h
BLOCK MINED AT SIZE 9..... 2018-02-28T15:05:41.635+01:00, akka://VSystem/user/Master/NodeRepo/$j
Update neighbours table...
BLOCK MINED AT SIZE 10..... 2018-02-28T15:19:06.635+01:00, akka://VSystem/user/Master/NodeRepo/$c
BLOCK MINED AT SIZE 11..... 2018-02-28T15:41:48.635+01:00, akka://VSystem/user/Master/NodeRepo/$g
BLOCK MINED AT SIZE 12..... 2018-02-28T15:44:49.635+01:00, akka://VSystem/user/Master/NodeRepo/$j
BLOCK MINED AT SIZE 13..... 2018-02-28T15:49:30.635+01:00, akka://VSystem/user/Master/NodeRepo/$g
BLOCK MINED AT SIZE 14..... 2018-02-28T15:54:17.635+01:00, akka://VSystem/user/Master/NodeRepo/$a
BLOCK MINED AT SIZE 15..... 2018-02-28T15:58:30.635+01:00, akka://VSystem/user/Master/NodeRepo/$d
Update neighbours table...
BLOCK MINED AT SIZE 16..... 2018-02-28T16:12:24.635+01:00, akka://VSystem/user/Master/NodeRepo/$d
BLOCK MINED AT SIZE 17..... 2018-02-28T16:31:49.635+01:00, akka://VSystem/user/Master/NodeRepo/$a
```

Figure 5.1: VIBES raw logs

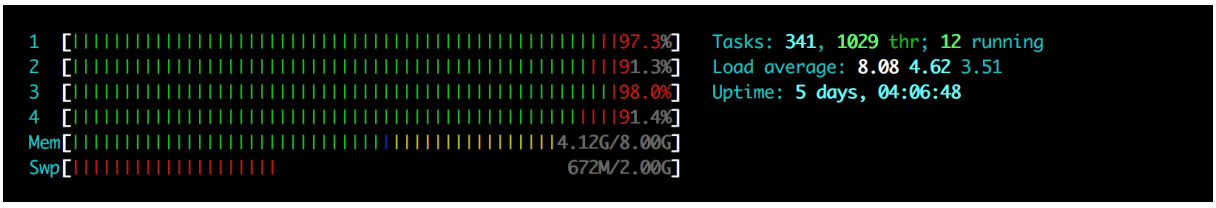


Figure 5.2: CPU Utilization, all cores at full power

5.2 Speed

We were able to simulate on the small MacBook 500 Nodes with 500 Transactions and 8 neighbours 5 times faster than real time. Without any transactions the MacBook is able to simulate more than 3000 Nodes, this simulation replicates the non-transactional behaviour of the Bitcoin-Simulator [11] capped at 6000 Nodes.

On a high-end personal computer the numbers will be more substantial. With numbers such as $10 - 20$ Nodes that could be realistically simulated in a testnet [28] network, VIBES could fast forward months ahead of time in a matter of minutes. The speed comes from the high CPU utilization by Akka and the introduced in Section 4.2 concept of fast-forward computing.

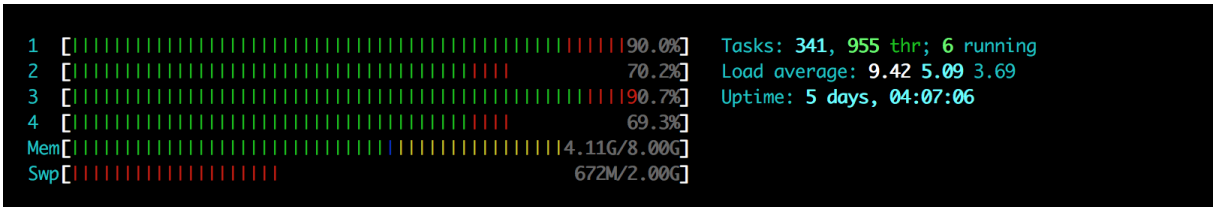


Figure 5.3: CPU Utilization, slight decrease when block mined

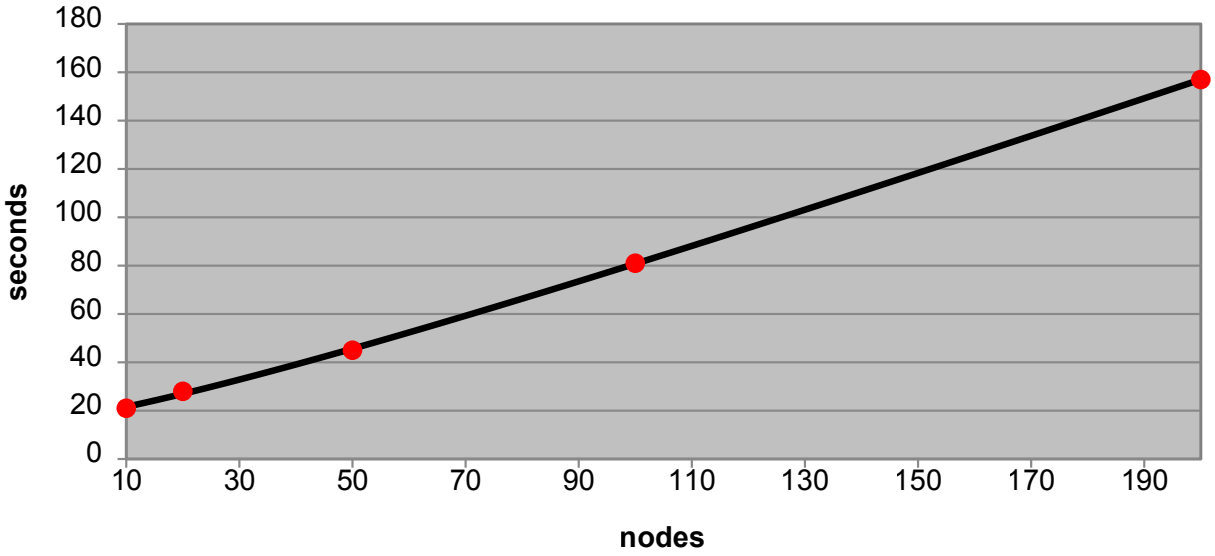


Figure 5.4: VIBES - linear in number of nodes

5.3 Scalability

A number of experiments have been conducted to quantify scalability of VIBES. Note that the simulation time increases only in in number of Nodes, transactions and neighbours. The rest of the input parameters such as block size do not have influence on performance - VIBES scales there in $\mathcal{O}(1)$.

5.3.1 Varying Nodes

For this experiment we wanted to simulated 3 hours ahead of time, 100 transactions per *blockTime* and 8 neighbours. As see on Figure 5.4 VIBES scales linearly in number of Nodes. As we increase them, simulation times increase linearly.

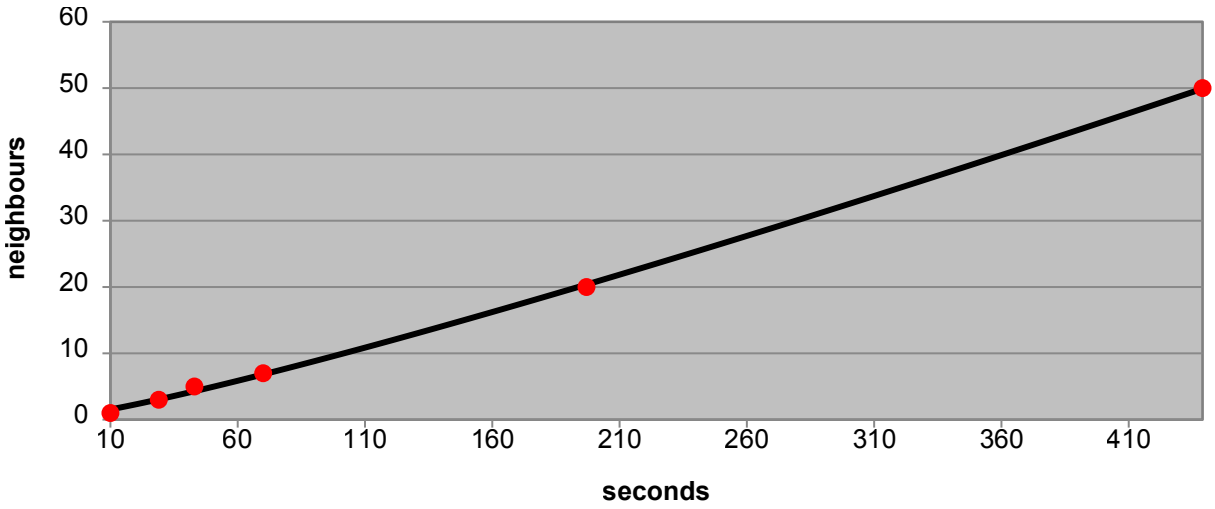


Figure 5.5: VIBES - linear in number of neighbours

5.3.2 Varying Neighbours

This simulation again was 3 hours ahead of time, with 100 Nodes and 100 Transactions per $blockTime$. We varied the number of neighbours there to conclude that the complexity was $\mathcal{O}(n)$ once again as seen on Figure 5.5

5.3.3 Varying Transactions

As with other experiments, we simulated 3 hours ahead of time, 100 Nodes with 8 neighbours, but changed the number of transactions per $blockTime$ ($throughput$). Results are shown of Figure 5.6.

5.3.4 Conclusion

Finally, we can conclude that VIBES scales in $\mathcal{O}(n)$ in number of Nodes, neighbours and transactions. All other parameters have time complexity $\mathcal{O}(1)$.

5.4 Flexibility

The complexity of any system increases exponentially with the variability, therefore it is very difficult to walk the line between not flexible enough and too complex. In its first

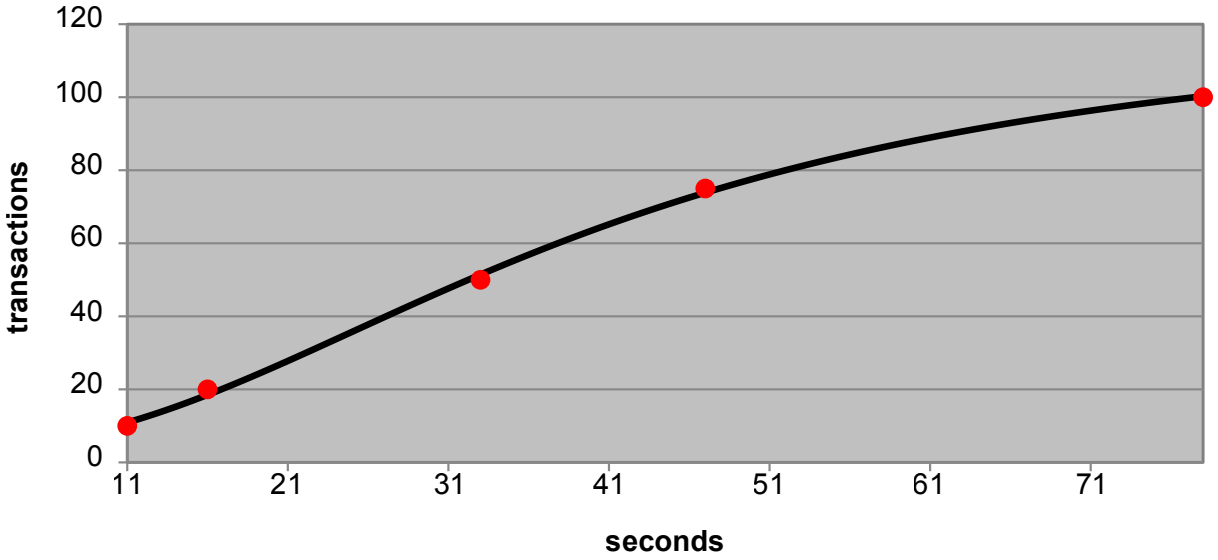


Figure 5.6: VIBES - linear in number of transactions

version VIBES provides the most important parameters to simulate a blockchain network as follows

- $numberOfNodes$: number of nodes
- $blockTime$: expected time to solve a block
- $neighbourDiscoveryInterval$: the interval at which nodes to update their neighbours table
- $numberOfNeighbours$: number of open connections from a node to other peers in the network
- $throughput$ - expected number of transactions in a block
- $transactionSize$ - size of transaction in KB
- $propagationDelay$ - transmission + verification time
- $simulateUntil$ - the moment in the future where simulation stops

These parameters lay out the foundation to simulate a typical blockchain network. However, in future versions, the inputs could be broken down to increase granularity. For instance, $propagationDelay$ could be represented by three separate components for more accurate results such as: network bandwidth, block size and distance between Nodes. This is another great opportunity for future work, but as with each first iteration of a software product, we kept it reasonably simple.

5.5 Extensibility

VIBES is based on Akka, a framework to build powerful reactive, concurrent, and distributed applications more easily [44], and simulates the network at an event level while trying to avoid implementation details of any specific flavour of blockchain - be it bitcoin, ethereum or hyperledger. To show the extensibility of the simulator, we'll give two examples:

Scenario 1

Developer A wants to extend the simulator to use Proof of Work in order to determine how much resources the network needs (CPU, Electricity, Bandwidth, etc).

To achieve that, the developer would need to extend the NodeActor class and add relevant parameters such as: CPU, electricity cost, hash rate and other PoW specific details. Once a Node has solved a block, the whole network gets notified and every Node could finaly compute for itself how much resources it has spent based on the timestamp. Another Actor could also aggregate the results for each Node in order to calculate a system wide cost calculations.

Scenario 2

Developer B wants to implement Proof of Stake.

Now PoS has many different flavours [21], but let's assume a simple scenario for the sake of clarity: Every Node has a chance to solve the next block based on its stake (or crypto assets it holds). To recreate this type of setting, Developer B needs to deploy Actors that are aware of the system's total stake. Let's call them ContractorActor.

When making best guesses, Nodes will simply send their stakes to their Contractors and let the Contractors make best guesses for them since they're aware of the likelihood that a Node will find a solution for the next block. Remember that the likelihood is based on the system's global stake that the Contractors are aware of and also on the Node's own stake.

Figure 5.7: Screenshot Home Screen

Verdict

These two example scenarios presented above prove the extensibility of the simulator to more specific use cases beyond the current implementation.

5.6 Powerful Visuals

Our initial proposal envisioned a simple display of the final output in a readable format, ideally in a web browser instead of the terminal. This goal has been accomplished and on top we were able to uniquely visualize the interactions between Nodes during the simulation on the world's map as showcased on the screenshots below.

5.7 Case Study - Optimal Number of Neighbours in a Blockchain Network

We'll use this case study to show that a simulator could be used to reason about a potential improvements in the network.

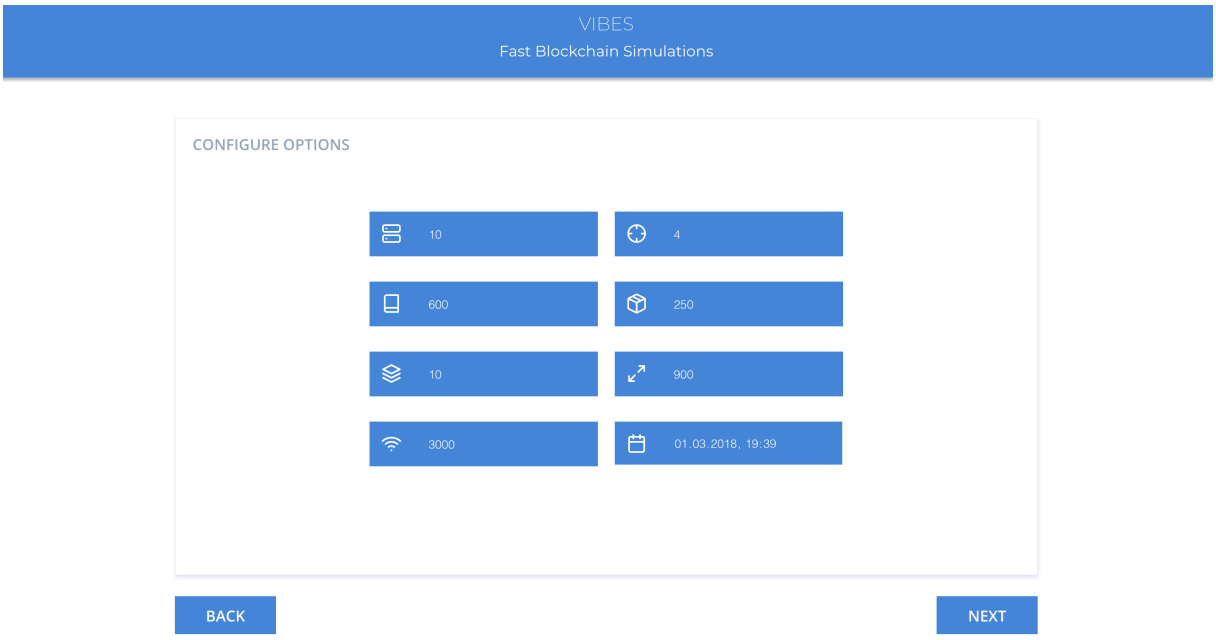


Figure 5.8: Screenshot Config Screen

As I was doing research about bitcoin implementation details, I asked myself how many neighbours a typical Node has. In other words - how many open connection does a Node have and propagate blocks to? I could not find the answer, so asked the community. The response of one of bitcoin's core developers was:

It is impossible to know how many Nodes each Node is connected to or has in its peers table. This information is not broadcast publicly and the information that is available publicly (by asking a Node) is not representative of what the Node actually knows. [45]

However, this is an important metric. If there are not enough neighbours, propagation times in the network will increase, a lot of Nodes will be wasting work on old blocks and the security of the network will also be compromised. On the other hand, if there are too many open connections, Nodes are wasting extra resources to propagate blocks to Nodes that have most likely already received them.

With VIBES we can determine the optimal number of neighbours in the network. Let's assume 100 Nodes and each Node updates its neighbours table in roughly 10 minute intervals. After running a VIBES simulation, we conclude that the minimal required number of neighbours that does not dramatically increase propagation times or number

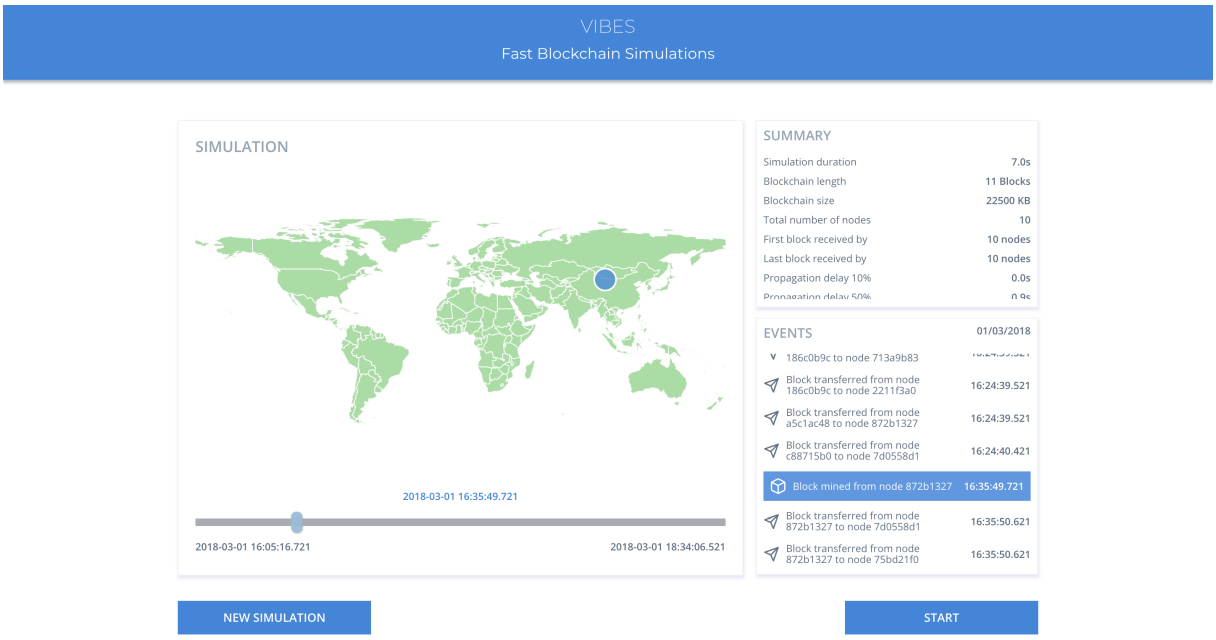


Figure 5.9: Screenshot Visualization - Mine Block

of stale blocks is 4. Below 4, we have observed many stale blocks and bigger propagation times. So in this setting, the number of optimal neighbours is about $4 - 5$.

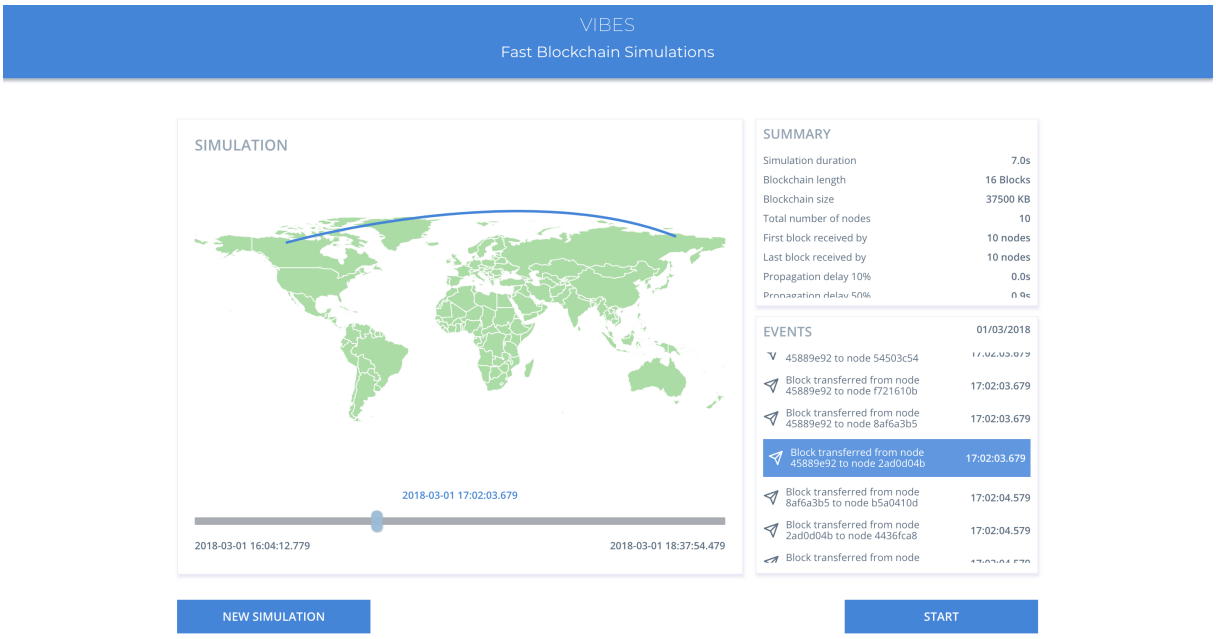


Figure 5.10: Screenshot Visualization - Transfer Block

Chapter 6

Summary

In this paper, we have proposed, conceptualized and implemented a configurable, scalable and fast blockchain simulator. VIBES is designed to be extensible beyond bitcoin and is the first of its kind to be able to simulate transactions in the network. Previous work [11] simulates the network at block level only. We were able to replicate the parallel and concurrent nature of the network, span multiple threads to carry out expensive computations and achieve full CPU utilization. VIBES is based on the Actor Model and is implemented as a reactive, message-driven system. Immutability first and no shared memory between entities complete the philosophy of our simulator. Moreover, to complement the backend we have designed and implemented a single-page application to visualize the interactions between nodes during a simulation and display summary of the results. The design decisions behind the frontend architecture have been driven by atomic design.

6.1 Status

We were able to complete our initial goal and propose a simulator that differentiates itself from previous approaches that emulate the network in an unscalable fashion. The approach proposed in this thesis lays down a very solid foundation for future work. Although we have a very strong evidence that VIBES realistically captures the behaviour in the network, it could be improved and extended as elaborated in Section 6.3.

6.2 Conclusions

It was very interesting working on this topic, in particular because it is a seemingly unexplored field with great potential. On the other hand, it was very challenging. For the most part because of the lack of references how to solve the problem of replicating the behaviour of thousands of nodes that typically require as much electricity as a whole country on a single PC. Apart from Bitcon-Simulator [11] and now VIBES there have been no attempts to simulate a blockchain network. However, realistic simulations could be invaluable tool for developers to reason about improvement proposals in the network and their consequences as shown in our case study in Chapter 5. Realistic simulations could help build better blockchain networks.

6.3 Future Work

There are several ways to go about future work that we will present here.

6.3.1 Improve Core and Configuration Parameters

The first way is to improve the simulator as it is, to improve the core.

Propagation Delay

As described in Section 5.4, we have the parameter *propagationDelay* which represents verification and transmission time of a block. No matter how big the block size or how long the distance between the sender and recipient of a block is, the propagation always takes *propagationDelay* time.

This parameter could be broken down into three different parameters as follows: block size, network bandwidth and distance between Nodes. Then propagation delay could be inferred from them to result in a more accurate simulation.

Maximal Block Size

Currently the block size is not capped and could be arbitrary big. If the block size was capped, maybe there will emerge a scenario where transactions could not be fitted in a block and have to wait until the next one is mined. To more accurately represent the behaviour of the system, I recommend to include a maximal block size parameter in the configuration of the network.

Segwit2x

Once $propagationDelay$ is broken down into its three building blocks - block size, distance between nodes, network bandwitdh and the parameter $maximalBlockSize$ introduced, VIBES will be able to analyze the implications of Segwit2x in bitcoin by simply doubling $maximalBlockSize$.

Transaction Incentives

Moreover, when we introduce $maximalBlockSize$, incentives will start playing a role in the simulation as well. The transactions with highest incentives will be included in the next block, while all others will have to wait if the block size is not big enough. Therefore, it also makes sense to include and parametrize incentives in the network once $maximalBlockSize$ has been introduced.

Speed

Although, VIBES is pretty fast already, I could not spend as much time on performance optimization as I wanted in regard to better choice of data structures and memory utilization. For instance, one could use mutable variables with care at some places instead of their slower, thread-safe immutable variants. I believe there is still a lot of potential to further improve performance.

Block Propagation Times

Another major challenge is to explore propagation times in a blockchain network and adjust VIBES as needed to capture this behaviour more realistically. As described in Subsection 5.1.5, there are still a few open questions there.

Node Generation

Currently Nodes are arbitrary generated on the worlds map. It would be nice if user could specify in an intuitive way where and how to distribute nodes on the map. Once the *propagationDelay* parameter is broken down into more components and distance between nodes better captured by VIBES, nodes' distribution would be another important factor for more accurate simulations.

Topology of the Network

VIBES provides no way to specify the topology of the network. It might be interesting to explore how the the efficiency and behaviour of blockchain based networks change as different topologies are deployed.

Full Node vs Miner

In its first implementation VIBES does not differentiate between a full Node and a miner. In future work this behaviour could be implemented.

Smart Contracts

Also, VIBES does not include Smart Contracts. However, this could be implemented by including them as part of an executable.

Attacker Simulation

To analyze security properties of the network, in future versions, one might conceptualize a way to include attackers in the network.

6.3.2 Extend to Implement PoW or PoS

Another way to continue with future work is to actually extend the core and implement more specific behaviour such as PoW or PoS. In a PoW type of setting it would be interesting how much electricity the whole network needs or how much does a transaction actually cost. Once VIBES has been extended to implement more specific use cases different ideas might emerge how to better generalize the core and improve its extensibility.

6.3.3 Compare vs NS-3

A good opportunity for future work is to directly compare the amount of events NS-3 and VIBES could process within certain amount of time, because both are based on a scheduler that works similarly.

6.3.4 Frontend and Data Analysis

One could improve VIBES in direction data analysis. Our Frontend currently displays very basic information from the network, although the simulation itself captures much more complex behaviour that could be analysed. One example for that is the rate at which forks occur based on different configuration parameters.

6.3.5 Frontend and Lazy Evaluation

Currently the Frontend receives all data analyzed by the backend for a particular simulation. The data includes the longest blockchain with all its transactions. Furthermore, it includes all events and interactions that happened in the network. This behaviour does not scale well, because at some point there are too many events, especially if the simulation was long and included a lot of nodes. The Frontend can not process so much data at once. Instead, data from the simulation should be fetched lazily, on demand.

6.3.6 Database

In order to lazily evaluate data, the backend needs to save the result of a certain simulation somewhere. This might require setting up a database.

6.3.7 Tests

It is important to write regression tests for the simulator in order to be able to develop with confidence. Initially, I wrote a lot of tests, but they soon became obsolete as I decided to implement an improved version of the initial algorithm. Without regression tests, I was less confident to make changes for already working versions of the simulator.

Appendices

Appendix A

Appendix

A.1 Source code

Available at https://github.com/lustoykov/vibes

A.2 Coordinator's source code

```
package com.vibes.actors

import akka.actor.{Actor, ActorContext, ActorRef, Props}
import akka.util.Timeout
import com.vibes.actions._
import com.vibes.utils.{VConf, VExecution}
import org.joda.time._

import scala.collection.immutable.SortedSet
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Promise
import scala.concurrent.duration._
import scala.util.Random

/**
  * This class represents the coordinator that controls the nodes and
↪  the execution order in the network.
  *
  * Currently there are assumptions that this actor makes. Also
↪  assumptions that the node actors make about this actor
```

```
  * which leads to unnecessary coupling and a couple of WTFs, but this
↪  is all for performance reasons.
  * For instance, to be able to utilize parallelism the MasterActor
↪  could fast forward multiple PropagateTransactions
  * aka give permission to multiple nodes to fast forward their
↪  propagation executables for transactions
  * because the order in which they are propagated does not matter as
↪  long as there is no other executable between them.
  * Meaning that sequential executables of type PropagateTransactions
↪  are simply executed in parallel.
  * Which means recipients of transaction might not be ordered by the
↪  timestamp, but the node will always receive
  * all due to transactions for a block / mempool within a correct time
  * TL:DR; We achieve parallelism by allowing transactions within a
↪  single block to be arbitrary reordered
  * Multiple actors are able to propagate multiple transactions between
↪  each other w/o asking for permission from the
  * MasterActor each time, because it already has issued them, multiple
↪  ones, as permitted.
  * Note that in the original bitcoin implementation a similar type of
↪  anarchy exists. If there is no solution
  * to the current block the node could change the order of transactions
↪  as it it suits him to start looking for a new
  * nonce / solution.
  *
  * The implications are whenever nodes need to mine a block, the
↪  MasterActor ensures all transactions are completed
  * that are permitted, empties all workRequests and queries all nodes
↪  for a new workRequest so it can get back the
  * system to a synchronized state.
  *
  * An alternative, cleaner solution, that I've tried in the beginning
↪  was the following:
  * Reset all the workRequests after an execution has taken place and
↪  ask all nodes for their workRequests once again
  * However, the MasterActor becomes the bottleneck because only 1 actor
↪  at a time can execute an operation,
  * while blocking all others. The current solution blocks only in
↪  particular cases, therefore is much faster,
  * but messier
  */
class MasterActor extends Actor {
  implicit val timeout: Timeout = Timeout(20.seconds)
```

```
  /**
    * NodeActors ask for work permission and MastreActors issues them so
↪   that they can fast forward
    * Only issue a permission once all currentNodeActors have voted
    */
  private var workRequests: SortedSet[VExecution.WorkRequest] =
   ↪  SortedSet.empty

  /**
    * This is a simple, but not robust solution to solve following
↪   problems: MasterActor must have reference to all
    * currentNodeActors to distribute work and also make sure they have
↪   casted their work requests.
    * Note: A better solution would be for them to register / deregister
↪   themselves via messages, but I did not bother
    * doing it in a prototype. So instead once an Actor votes, it is
↪   added to the set of currentNodeActors until
    * VConf.numberOfNodes has been reached.
    */
  private var currentNodeActors: Set[ActorRef]          = Set.empty
  private var numberOfWorkRequests: Map[ActorRef, Int] = Map.empty

  /**
    * Makes sure to update neighbours table at roughly
↪   lastNeighbourDiscoveryDate time intervals
    */
  private var lastNeighbourDiscoveryDate = DateTime.now

  /**
    * Returns final result of computation to be delivered to the client.
↪   Note that here the ask ? pattern should
    * be used instead of Promise, because this solution would break for
↪   different JVMs communicating between
    * each other, cause currently I use reference to the promise.
↪   Instead, the sender should be passed around
    * combined with the ask pattern.
    */
  private var currentPromise = Promise[ReducerIntermediateResult]()

  /**
    * Injected Actors that are children of the MasterActor. Again, they
↪   should register themselves via messaging
```

```
    * instead, and proper error handling should be implemented if
↪   intended to use in a distributed setting
    */
  val discoveryActor: ActorRef =
    context.actorOf(DiscoveryActor.props(VConf.numberOfNeighbours),
    ↪  "Discovery")
  val reducerActor: ActorRef =
    context.actorOf(ReducerActor.props(self), "Reducer")
  val nodeRepoActor: ActorRef =
    context.actorOf(NodeRepoActor.props(discoveryActor, reducerActor),
    ↪  "NodeRepo")

  /**
    * Delegate work to NodeRepo to register NodeActors
    */
  (1 to VConf.numberOfNodes).foreach(_ => nodeRepoActor !
  ↪  NodeRepoActions.RegisterNode)

  override def preStart(): Unit = {
    println(s"MasterActor started ${self.path}")
  }

  override def receive: Receive = {
    case MasterActions.Start =>
      /**
        * Again, since strictly message-based communication would be
↪   much more involved, for the prototype I just
        * assume that every NodeActor would be alive after X seconds and
↪   let the DiscoveryActor Announce the neighbours
        * and start
        */
      context.system.scheduler.scheduleOnce(3000.millisecond) {
        discoveryActor ! DiscoveryActions.AnnounceNeighbours
      }

      context.system.scheduler.scheduleOnce(7000.millisecond) {
        nodeRepoActor ! NodeRepoActions.AnnounceStart(DateTime.now)
      }

      sender ! currentPromise

    case MasterActions.FinishEvents(events) =>
      println("FINISH EVENTS...")
```

```
      currentPromise.success(events)

    case MasterActions.CastWorkRequest(workRequest) =>
      /**
        * because some nodes receive the right to workRequest more than
↪   once (for instance if we fast forward multiple
        * transactions that are being sent to the same node / actor,
↪   he'll workRequest multiple times. We're only interested
        * in the last workRequest he submitted, because then we know at
↪   the time of the last submission his execution queue
        * was complete
        */
      numberOfWorkRequests.get(workRequest.fromActor) match {
        // discard all workRequests but the last one
        case Some(int) if int > 0 =>
          numberOfWorkRequests += (workRequest.fromActor -> (int - 1))
        case _ =>
          // the last workRequest goes on
          currentNodeActors += workRequest.fromActor
          // assert no workRequest should be received more than once
          assert(!workRequests.contains(workRequest))
          workRequests += workRequest

          // have all workRequests been collected?
          if (workRequests.size == VConf.numberOfNodes) {
            // number of actors that requested work should be the smae
             ↪  of number of nodes (aka each requested work once)
            assert(currentNodeActors.size == VConf.numberOfNodes)
            // assert each requested work once, checked in an
             ↪  alternative way
            assert(workRequests.map(_.fromActor).toSet.size ==
             ↪  workRequests.size)

            // Do neighbour discovery / update neighbour tables
            // this is a convenient, but not 100% correct place to
             ↪  execute this type of functionality
            val priorityWorkRequest = workRequests.head
            if (new org.joda.time.Duration(lastNeighbourDiscoveryDate,
             ↪  priorityWorkRequest.timestamp)
                  .isLongerThan(
                    new org.joda.time.Duration(
                      lastNeighbourDiscoveryDate,
                      lastNeighbourDiscoveryDate
```

```
            .plusSeconds(VConf.neighboursDiscoveryInterval))
        )) {
      lastNeighbourDiscoveryDate = priorityWorkRequest.timestamp
      discoveryActor ! DiscoveryActions.AnnounceNeighbours
    }

    // if simulation over, announce end
    if
    ↪  (VConf.simulateUntil.isBefore(priorityWorkRequest.timestamp))
    ↪  {
      nodeRepoActor ! NodeRepoActions.AnnounceEnd
    } else if (priorityWorkRequest.executionType ==
    ↪  VExecution.ExecutionType.MineBlock) {
      // if mining of a block should be performed,
      // first of all distribute the throughPut of transactions
       ↪  to the nodes for the next
      // interval of mining
      currentNodeActors = Random.shuffle(currentNodeActors)
      val actorsVector = currentNodeActors.toVector
      // distribute randomly requests to NodeActors to create
       ↪  throughput number of transactions within blockTime
      (1 to VConf.throughPut).foreach { index =>
        val randomActorFrom =
         ↪  actorsVector(Random.nextInt(actorsVector.size))
        val randomActorTo   =
         ↪  actorsVector(Random.nextInt(actorsVector.size))
        val now             = priorityWorkRequest.timestamp
        randomActorFrom ! NodeActions.IssueTransaction(
          randomActorTo,
          // this means that within the blockTime the throughPut
           ↪  number of transactions will be issued
          // by the nodes. Note that block could be mined in
           ↪  less or more than blockTime, therefore this only
          // ensures average number of transactions per mined
           ↪  block and exact number of transactions per
          // blockTime
          now.plusMillis(VConf.blockTime * 1000 / (index + 1))
        )
      }

      // clear all workRequests
      workRequests = SortedSet.empty
      // let the first actor mine the block
```

```
        // in this executable it would also
         ↪ AnnounceNextWorkRequestAndMine to other actors
        priorityWorkRequest.fromActor !
         ↪ NodeActions.ProcessNextExecutable(priorityWorkRequest)
      } else if (priorityWorkRequest.executionType ==
       ↪ VExecution.ExecutionType.PropagateTransaction) {
        // if propagate transaction as execution type, then
         ↪ collect the workRequests until
        // ExecutionType.PropagateTransaction is in the pipeline
         ↪ and forward all of them and give permission
        // to nodes to execute them. Nodes that are executing them
         ↪ will most likely be receiving transactions
        // from other nodes as well, so we'll need to figure out
         ↪ how many times a node will be voting which
        // is done via numberOfWorkRequests.
        // Imagine A1 transfer transaction to A2, but A2 also
         ↪ needs to transfer transaction and after that
        // propagate a block. Imagine now A2 transfers transaction
         ↪ and workRequests with propagate block. Later
        // A2 receives the transaction from A1 and now has in the
         ↪ queue on top of propagate block propagate
        // transaction. Luckily, A2's first workRequest will have
         ↪ been discarded because of numberOfWorkRequests > 1
        // and now A2 will be able to correctly workRequest with
         ↪ the most recene peace of executable on top
        // (which is propagate transaction)
        var propagateWorkRequests: List[VExecution.WorkRequest] =
          List.empty
        while (workRequests.nonEmpty &&
         ↪ workRequests.head.executionType ==
         ↪ VExecution.ExecutionType.PropagateTransaction) {
          propagateWorkRequests ::= workRequests.head
          workRequests = workRequests.tail
        }

        // remove all actors that requested work that are going to
         ↪ receive something, cause they should request work
         ↪ again
        workRequests = workRequests.filter(
          workRequest =>
            !propagateWorkRequests
              .map(_.toActor)
              .contains(workRequest.fromActor))
```

```
        // figure out how many workRequests we'll receive from
         ↪  each actor so that we can discard all of them but
        // the last one
        numberOfWorkRequests = Map.empty
        propagateWorkRequests.foreach { workRequest =>
          numberOfWorkRequests.get(workRequest.toActor) match {
            case Some(int) =>
              numberOfWorkRequests += (workRequest.toActor -> (int
               ↪  + 1))
            case _ => numberOfWorkRequests += (workRequest.toActor
             ↪  -> 0)
          }

          numberOfWorkRequests.get(workRequest.fromActor) match {
            case Some(int) =>
              numberOfWorkRequests += (workRequest.fromActor ->
               ↪  (int + 1))
            case _ => numberOfWorkRequests +=
             ↪  (workRequest.fromActor -> 0)
          }
        }

        propagateWorkRequests.foreach(workRequest =>
          workRequest.fromActor !
           ↪  NodeActions.ProcessNextExecutable(priorityWorkRequest))

      } else {
        // if it's anything else such as propagate a block, just
         ↪  execute it by 1 single actor
        // this happens rarely in comparison to transactions so no
         ↪  need for extra performance boost here
        // simply block the whole system, let the 2 actors
         ↪  involved in block propagation do their work
        // and carry on
        // The "from" actor will propagate the block and then cast
         ↪  a new workRequest, the receiving actor
        // will receive the block and in ReceiveBlock cast a new
         ↪  workRequest as well
        workRequests = workRequests.tail
        workRequests = workRequests.filter(_.fromActor !=
         ↪  priorityWorkRequest.toActor)
```

```
                priorityWorkRequest.fromActor !
                 ↪ NodeActions.ProcessNextExecutable(priorityWorkRequest)
              }
            }
        }
    }
}

object MasterActor {
  def props(): Props = Props(new MasterActor())
}
```

A.3 Fun Stats

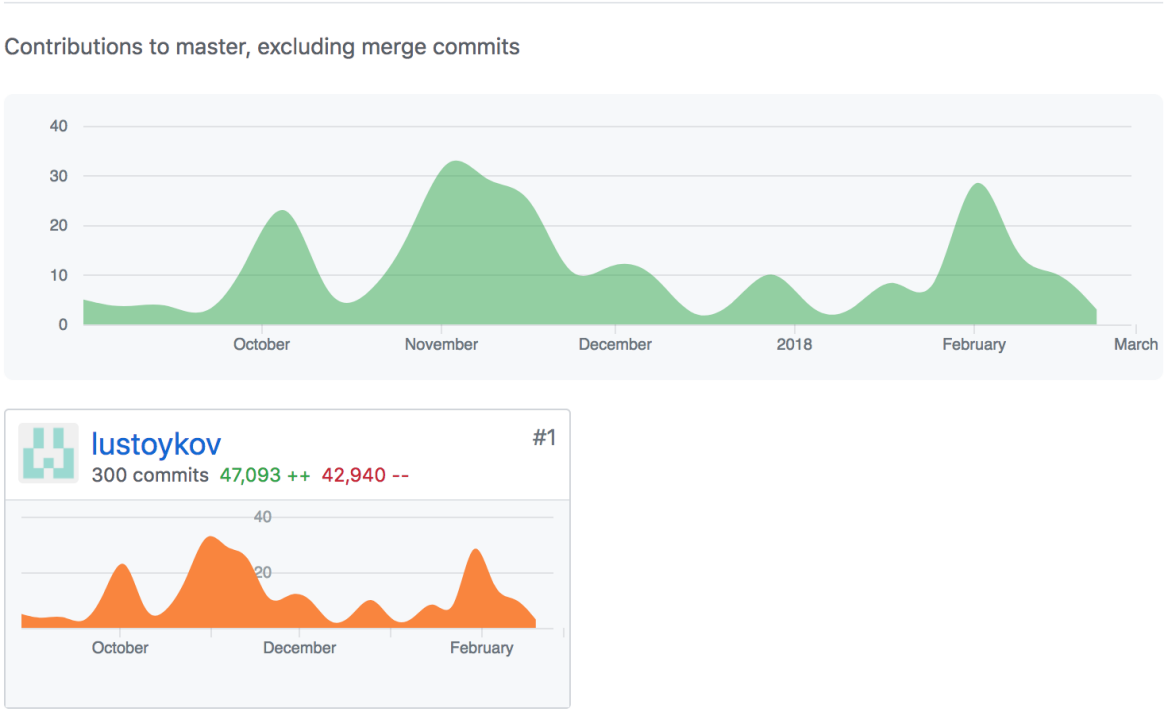


Figure A.1: VIBES - commit chart

A.3.1 Commit Chart

Presented on Figure A.1

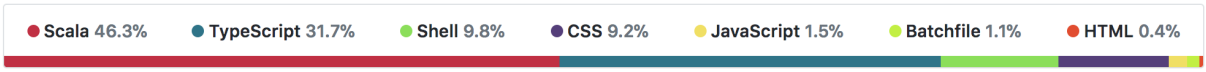


Figure A.2: VIBES - Programming Languages Distribution

A.3.2 Programming Languages Distribution

Presented on Figure A.2

A.4 Installation Instructions

A.4.1 Backend

SBT

The only thing you need to install is SBT (interactive scala build tool) as documented: https://www.scala-sbt.org/

Run server

Navigate to the root folder and run *sbt server/run*. VIBES's Server will be available on localhost, port 80802.

A.4.2 Frontend

Install npm

You'd need to install the *npm* package manager as documented: https://www.npmjs.com/

Instal yarn

Then install *Yarn*: https://yarnpkg.com/lang/en/docs/install/

Install package.json dependencies

After the installation is done you can navigate to the *frontend/* folder of the project and run *yarn install*.

Run server

The last step is to run *yarn dev*. The web interface will be accessible on localhost:8080.

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." https://bitcoin.org/bitcoin.pdf, 2008. Accessed: 2018-02-23.

[2] "Bitcoin wiki." https://en.bitcoin.it/wiki, Feb 2018. Accessed: 2018-03-12.

[3] M. D'Alliessi, "How does the blockchain work." https://medium.com/@micheledaliessi/how-does-the-blockchain-work-98c8cd01d2ae, Jun 2016. Accessed: 2018-02-23.

[4] "Akka actors intro." https://doc.akka.io/docs/akka/current/guide/actors-intro.html, 2018. Accessed: 2018-02-23.

[5] Gartner, "Gartner hype cycle." https://www.gartner.com/technology/research/methodologies/hype-cycle.jsp, 2018. Accessed: 2018-02-23.

[6] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. Weed Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," *ArXiv e-prints*, Jan. 2018.

[7] S. Popov, "The tangle." https://iota.org/IOTA_Whitepaper.pdf, Oct. 2017. Accessed: 2018-02-23.

[8] "The future of commerce. a decentralized network for payment requests.." https://request.network/assets/pdf/request_whitepaper.pdf, Jan. 2018. Accessed: 2018-02-23.

[9] "Bitcoin scaling problem, explained." https://cointelegraph.com/explained/bitcoin-scaling-problem-explained, 2017. Accessed: 2018-03-09.

[10] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K. Tan, "BLOCKBENCH: A framework for analyzing private blockchains," *CoRR*, vol. abs/1703.04057, 2017.

[11] A. Gervais, G. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the*

23nd ACM SIGSAC Conference on Computer and Communication Security (CCS), ACM, 2016.

[12] B. Frost, "Atomic web design." `http://bradfrost.com/blog/post/atomic-web-design/`. Accessed: 2018-01-03.

[13] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Proc. of the First International Conference on Peer-to-Peer Computing*, pp. 101 – 102, 09 2001.

[14] S. Santra and P. P. Acharjya, "A study and analysis on computer network topology for data communication," *International Journal of Emerging Technology and Advanced Engineering*, vol. 3, Jan. 2013.

[15] R. Sobti and G. Ganesan, "Cryptographic hash functions: A review," vol. Vol 9, pp. 461 – 479, 03 2012.

[16] R. Kaur and A. Kaur, "Digital signature," in *2012 International Conference on Computing Sciences*, pp. 295–301, Sept 2012.

[17] UK Government, Office for Science, "Distributed ledger technology: beyond block chain." `https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf`, 2016. Accessed: 2018-02-23.

[18] TheEconomist, "Blockchains: The great chain of being sure about things." `https://cointelegraph.com/explained/bitcoin-scaling-problem-explained`, 2015. Accessed: 2018-03-10.

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

[20] "Bitcoin developer guide." `https://bitcoin.org/en/developer-guide`, 2017. Accessed: 2018-03-12.

[21] "Proof of stake faq." `https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ`, 2018. Accessed: 2018-01-03.

[22] "Ns-3." `https://www.nsnam.org/`. Accessed: 2018-03-09.

[23] "Peersim." `http://peersim.sourceforge.net/`. Accessed: 2018-03-09.

[24] "Omnet++." `https://omnetpp.org/`. Accessed: 2018-03-09.

[25] "Opnet technologies." `https://www.riverbed.com/de/products/steelcentral/opnet.html`. Accessed: 2018-03-09.

[26] "Which simulator is the best for simulating peer-to-peer topology?." `https://www.researchgate.net/post/Which_simulator_is_the_best_for_simulating_peer-to-peer_topology`. Accessed: 2018-03-09.

[27] "Events and simulator." `https://www.nsnam.org/docs/manual/html/events.html`. Accessed: 2018-03-09.

[28] "Testnet." `https://en.bitcoin.it/wiki/Testnet`, Feb 2018. Accessed: 2018-02-23.

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

[30] K. R. T. M. Bonér J., Farley D., "Reactive manifesto." `https://www.reactivemanifesto.org/`, Sept 2014. Accessed: 2018-02-03.

[31] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[32] "A json library for scala powered by cats." `https://circe.github.io/circe/`, 2018. Accessed: 2018-02-03.

[33] "Scalafmt - code formatter for scala." `http://scalameta.org/scalafmt/`, 2018. Accessed: 2018-02-03.

[34] "Akka http." `https://doc.akka.io/docs/akka-http/current/index.html`, 2018. Accessed: 2018-02-03.

[35] "Sketch - the digital design toolkit." `https://www.sketchapp.com/`. Accessed: 2018-02-03.

[36] "Typescript - javascript that scales." `https://www.typescriptlang.org/`. Accessed: 2018-02-03.

[37] "React - a javascript library for building user interfaces." `https://reactjs.org/`. Accessed: 2018-02-03.

[38] "Postcss." `https://github.com/postcss/postcss`. Accessed: 2018-03-03.

[39] "Webpack - a module bundler." `https://webpack.js.org/`. Accessed: 2018-02-03.

[40] "Tslint - an extensible linter for the typescript language.." `https://palantir.github.io/tslint/`. Accessed: 2018-05-03.

[41] "Datamaps." `http://datamaps.github.io/`. Accessed: 2018-04-03.

[42] "Expectation of minimum of n i.i.d. uniform random variables.." `https://math.stackexchange.com/questions/786392/expectation-of-minimum-of-n-i-i-d-uniform-random-variables`, 2014. Accessed: 2018-03-01.

[43] "Bitcoin stats." `http://bitcoinstats.com/network/propagation/`, 2010. Accessed: 2018-02-23.

[44] "Akka." `https://akka.io/`, 2018. Accessed: 2018-01-03.

[45] "Number of neigbhours in bitocin network." `https://bitcoin.stackexchange.com/questions/70988/how-many-neighbours-does-a-node-have`, Feb 2018. Accessed: 2018-01-03.