Technische Universität München

Fakultät für Informatik

Master's Thesis in Informatik

Simulations of Scalable Autonomous Smart Contracts on Ethereum Plasma Blockchain

Georgi Aylov

Technische Universität München

Fakultät für Informatik

Master's Thesis in Informatik

Simulations of Scalable Autonomous Smart Contracts on Ethereum Plasma Blockchain

Simulationen von skalierbaren, autonomen Smart Contracts auf Ethereum Plasma Blockchain

Author: Georgi Aylov

Supervisor: Prof. Dr. Hans-Arno Jacobsen

Advisor: MSc. Pezhman Nasirifard

Submission: 24.01.2019

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

München, 24.01.2019

(Georgi Aylov)

Abstract

Diverse blockchain technologies such as Bitcoin, Ethereum and Hyperledger have gained public awareness and have become a subject to undergoing intense studies in the last couple of years with both positive and negative forecasts for their mass adoption in future. One of the core problems cryptocurrency systems face is the inability to keep up with the increasing number of users. Permissionless public blockchains can process less than 20 transactions per second, while other mainstream but centralized services such as Visa process more than 1500 transactions per second. To meet the high demand, researchers are developing various constructs that extend the functionality of blockchain systems with the main goal to increase the capacity of processed transactions. One of these constructs is Plasma. Plasma is a framework for running nested blockchains as a scalable autonomous smart contracts on a Turing-complete blockchain platform such as Ethereum aiming to significantly increase the number of processed transactions. Since Plasma is only a framework and not a specified solution, its security and reliability properties are highly disputed and not yet sufficiently analyzed. To better understand the behaviour of Plasma we propose a configurable Plasma simulator running as a large-scale peer-to-peer system. The simulator is capable of simulating multiple Plasma nested chains running as a smart contract on an Ethereum-like simulated blockchain. With Plasma simulator, users can explore important characteristics of Plasma and its interactions with Ethereum, and argue about different approaches for improving the design of existing and future concepts.

Inhaltsangabe

Diverse Blockchain-Technologien wie Bitcoin, Ethereum und Hyperledger haben in der Öffentlichkeit Beachtung gefunden und sind in den letzten Jahren Gegenstand intensiver Studien mit positiven und negativen Prognosen für ihre künftige Massenanwendung geworden. Eines der Hauptprobleme, mit denen Kryptowährungssysteme konfrontiert sind, ist die Unfähigkeit, mit der steigenden Anzahl von Benutzern mitzuhalten. Genehmigungslose öffentliche Blockchains können weniger als 20 Transaktionen pro Sekunde verarbeiten, während andere etablierte aber zentralisierte Dienste wie Visa mehr als 1500 Transaktionen pro Sekunde verarbeiten. Um die hohe Nachfrage zu befriedigen, entwickeln Forscher verschiedene Konstrukte, die die Funktionalität von Blockchain-Systemen erweitern, mit dem Hauptziel, die Kapazität verarbeiteter Transaktionen zu erhöhen. Eines dieser Konstrukte ist Plasma. Plasma ist ein Framework, um verschachtelte Blockchains als skalierbare autonome Smart-Verträge auf einer Turing-Complete-Blockchain-Plattform wie Ethereum auszuführen, die darauf abzielt, die Anzahl der verarbeiteten Transaktionen erheblich zu erhöhen. Da Plasma nur ein Framework und keine spezifizierte Lösung ist, sind seine Sicherheits- und Zuverlässigkeitseigenschaften höchst umstritten und noch nicht ausreichend analysiert. Um das Verhalten von Plasma besser zu verstehen, schlagen wir einen konfigurierbaren Plasmasimulator vor, der als großes Peer-to-Peer-System ausgeführt wird. Der Simulator ist in der Lage, mehrere in Plasma geschachtelte Ketten zu simulieren, die als Smart Contract in einer Ethereum-ähnlichen simulierten Blockchain ausgeführt werden. Mit dem Plasma-Simulator können Benutzer wichtige Eigenschaften von Plasma und seine Wechselwirkungen mit Ethereum erkunden und sich über verschiedene Ansätze zur Verbesserung des Designs vorhandener und zukünftiger Konzepte austauschen.

Acknowledgment

I would like to thank my advisor M.Sc. Pezhman Nasirifard for his support and advices throughout the entire process of writing this thesis and for making it such an enjoyable time for me. He was always willing to meet on short notice and to share thoughts that were of enormous worth for me. I would also like to thank my supervisor Prof. Hans-Arno Jacobsen for giving me the opportunity to write this thesis at the chair of Business Information Systems at Technical University of Munich.

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

3DES	Triple Data Encryption Standard.
AES	Advanced Encryption Standard.
CDN	Clients Discovery Node.
CIA	Confidentiality, Integrity, Availability.
DDoS attack	Distributed Denial of Service attack.
EVM	Ethereum Virtual Machine.
P2P	Peer-to-Peer.
PoA	Proof-of-Authority.
PoS	Proof-of-Stake.
PoW	Proof-of-Work.
SoC	Separation Of Concerns.
SRP	Single Responsibility Principle.

UTXO	Unspent Transaction Output.
VIBES	Visualizations of Interactive, Blockchain, Extended Simulations.

Chapter 1

Introduction

In order to ensure accurate financial reporting, reduce errors, and to prevent fraudulent financial activity the merchants of Venice invented the double-entry bookkeeping system in the early 13th century [7]. This system was revolutionary as it changed the entire world economy and enabled multiple parties from different parts of the world to trade efficiently. It also enabled the establishment of banks which function as a trusted third party when two other parties are willing to exchange monetary values. Economy without banks as well as transactions between two parties without a trusted third one were long seen as an unimaginable tasks. This all changed when a paper from unknown person (or a group of people) called Satoshi Nakamoto was published. It was about a new approach to exchange value in a form of a digital currency without a trusted third party - Bitcoin. The technology that Bitcoin is based on and that is poised to change the way people trust each other in the financial world is called Blockchain. Blockchain represents a distributed ledger technology that efficiently records transactions between two parties in a perpetual and accountable manner. Blockchain as a system with its components and characteristics will be discussed in Chapter 2.

As the ledger is distributed across the world, which means that multiple participants (computer machines) in a blockchain network communicate with each other and each of them stores a copy of this ledger, thus building a trustless distributed system, one could imagine that blockchain could be also employed for other purposes besides monetary transactions like for example legal contracts. To integrate legal or other types of contracts into blockchain one would need to add some kind of computation that will ensure fair and finite execution of such a contract. Such type of contracts are defined as smart contracts [8] and are applied in Ethereum [9].

The basic idea behind Ethereum smart contracts is that developers can add logic to a

smart contract in form of a source code, which code will execute each time a transaction with particular data is passed to its corresponding smart contract.
All blockchains, as already discussed, are distributed all around the world, Hence, there is no central authority that decides the balance (amount of money) of each participant. This means that all machines that are running the blockchain and are storing the ledger need to come up to one, synchronized among all participants, ledger. Such synchronization is achieved via different types of consensus algorithms such as Proof Of Work (PoW) [10] and Proof Of Stake (PoS) [11] .
These algorithms ensure finality and fairness within the network as well as they employ gamification mechanisms to create incentives for the nodes in the network to maintain the ledger and to act in an honest way.
With such characteristics blockchains gained popularity and more people have started to use them. As the number of users increased dramatically over the past couple of years the scalability of different blockchains became one of the main topics that is being discussed. The main issue with current state of art of blockchains is that they add limited number of transactions per second to the ledger - 3-7 in Bitcoin, 7-15 in Ethereum [12]. This is the reason why blockchains cannot face massive adoption and why researchers discuss ways to solve this critical problem.
Some of the most popular proposed solutions of the scalability problem are Lightning Network [13], Sharding [14] and Plasma framework [4]. The latter is the main concern of this paper and will be thoroughly discussed in the following chapters.

1.1 Motivation

Improving scalability of a blockchain comes along with particular tradeoffs that must always be considered when designing or redesigning a blockchain. These tradeoffs are usually coined as Blockchain Scalability Trilemma [15].
It indicates that a blockchain system can only posses two of the following three attributes:

- **Security** - a distributed network must be secured and must remain resistant to all possible attacks (Sybil attacks, 51% attacks, Distributed Denial of Service (DDoS) attacks, etc.). Furthermore, blockchain system must be fault-tolerant, meaning that the system will continue to operate in case components of the system fail.
- **Decentralization** - everyone possessing a computer unit connected to the internet can join a blockchain network. Most importantly, there is no central authority that controls the workflow of a blockchain. Regulations and policies are selected by all

participants in the network and there is no censorship on the content being stored on the chain.

- **Scalability** - ability of a blockchain to process increasing number of transactions with increasing number of participants. In other words, a scalable blockchain should effortlessly rise its capacity of processed transactions per second without additional stress on the network.

Based on this definition, improving the scalability of a blockchain means decreasing its security and/or decentralization. Such phenomenon can be observed with "easy solutions" such as splitting applications among multiple blockchains and increasing the block size limit within a blockchain [14]. Such solutions that attempt to make blockchains more scalable by upgrading the blockchains themselves are often called "layer 1" scaling solutions (since they modify the base layer).

There are also projects that focus on building on top of existing systems - Plasma Framework [4], Lightning Network [13], State Channels [16] and Truebit [17]. Such solutions are defined as "layer 2" or "off-chain" scaling solutions (as they create an additional layer without altering an already existing system).

One of the most promising layer 2 solutions is Plasma. It is a framework for conducting off-chain transactions while relying on an underlying root chain (e.g. Ethereum) to guarantee its security. Plasma comes with a novel technique for organizing off-chain transactions. Instead of creating channels between clients as in State Channels or Lightning Network, Plasma allows the creation of "child" blockchains attached to "main" blockchain (e.g. Ethereum). These child blockchains can themselves spawn new child blockchains and these child blockchains can spawn another level of child blockchains and so on. This allows the performing of multiple operations on child-chain level running applications or transferring of transactions with extensive number of users, with a minimal interaction with the root chain. As the Plasma child-chain operates on a partial state of the global state, that the Ethereum main chain maintains, it can move faster and charge less transactions fees.

The Plasma Framework was introduced by Joseph Poon and Vitalik Buterin in 2017 as a white paper that aims to form guidelines for concrete implementations that utilize the framework. It is not a defined solution as Bitcoin is but rather a model, which allows the formation of more formal and explicit Plasma applications such as Minimal Viable Plasma (Plasma MVP) [18], Plasma Cash [19] and Plasma Debit [20]. Due to the novelty of the approach, its high abstraction level and the early stage of adoption of the framework through particular applications, it would be highly useful to be able to simulate the behaviour of such complex system and get more insights into the internal communication between the parent blockchain and the children blockchains. Furthermore, experimentation at an early stage would ideally show the real practical potential of the Plasma Framework. For these reasons, this papers proposes a configurable Plasma simulator that aims to simulate the behavior of an Ethereum Blockchain and its communication with functional and correct Plasma child blockchains.

1.2 Problem Statement

To achieve mass adoption and to change the status quo not only in the financial sector but also in sectors where a decentralization of processing and storing information could be applied and be beneficial, the blockchain systems must significantly improve their scalability. One of the most promising solutions is the Plasma Framework. It spawns child blockchains into an additional layer that report their data to an existing blockchain which supports the execution of smart contracts. As this concept is quite new, there is high uncertainty in the practical use of such a solution. Research groups develop actual implementations of Plasma MVP [21], Plasma Cash [22] and Plasma Debit [23] as a proof of concept that later should be ideally converted into production applications. As these applications are mainly developed as clients, it is extremely hard to understand the consequences of such applications at large scale. Furthermore, the Plasma Framework as proposed in the paper from Vitalik Buterin and Joseph Poon could be used to come up with different, better and more secure varieties of Plasma solutions.

Building a simulator that simulates the behaviour not only of Plasma child blockchains but also on the underlying main chain would be particularly useful for researchers and developers as they would be able to analyze the network, recognize security weaknesses, improve the overall design of Plasma solutions and easily extend the simulator according to their approaches. Thus, this thesis proposes a Plasma simulator to help research groups, blockchain users and university students gain more insights into child blockchains running as scalable autonomous smart contracts on an Ethereum Plasma blockchain.

At the time of writing this thesis, there are a few approaches that aim to simulate the behaviour of blockchains. Given the fact that we aim to simulate Plasma contracts on an Ethereum-like blockchain, simulators focused on simulating Ethereum are especially relevant for us. Such simulation systems are Ganache [24], HIVE [25] and the Testnets - Ropsten [26], Kovan [27] and [28]. These tools have different objectives such as testing smart contracts and clients, but none of them represents a reliable way to include side chains and to analyze their behaviour regarding their internal characteristics and interactions with a main chain.

The goal of this thesis is to develop a simulator that mimics the behaviour of an Ethereum blockchain (transaction propagation, block generation, smart contract execution, node discovery) together with a Plasma blockchain(s) (Plasma transaction generation, deposit transaction, block creation, block submission) running on top of this Ethereum blockchain. This simulator also extends the functionality of the above mentioned proof of concept implementations of Plasma as it adds the possibility to simulate multiple nested blockchains. The Plasma simulator is designed to keep track of all interactions that happen inside the simulator such as detailed transactions, blocks and client balances. Another important aspect taken into consideration is the ability to simulate multiple blockchains on a single computer in a scalable and fast way.

To evaluate this approach, the paper concentrates on 7 different criteria:

- Correctness will be evaluated based on empirical analysis and will be ideally backed up by numerical proofs.
- Resource Utilization marks out the potential of the simulator to simulate hundreds of nodes on an personal computer and thousands of Nodes on a distributed cluster of multiple physical machines.
- Scalability refers to the ability of the simulator to handle growing amounts of work observed when the number of nodes, transactions and blocks increases.
- Flexibility is evaluated based on the ability to run a simulator with multiple configurations.
- Extensibility defines the ability of the system to be used as a framework where developers can implement and easily add new features.
- Plasma Framework Compliance addresses the features implemented in our simulator compared with the features of a general Plasma framework presented in its white paper.
- Powerful visuals appoints the attempt to provide the user with a useful user interface

that marks out the important characteristics of the simulated system.

1.3 Approach

To achieve the presented objectives, an appropriate set of tools was selected in order to set the ground for the simulator's scalability and to ensure good readability of the code with minimum boilerplate code. To make sure the architecture of the system can be easily extended we developed it in accordance with Domain Design Principles [29]. Furthermore, by following the principles of Reactive Manifesto [30], we built a robust, resilient and flexible simulation system. The visual representation of the simulator - the user interface, was designed based on the Redux Architecture style [31].

1.4 Contribution

Plasma is quite new approach for solving the scalability issue that Ethereum blockchain is facing. It represents a framework and does not suggest a concrete solution. That is why researchers from different communities and foundation gather on regular basis to discuss implementation details about a particular Plasma implementation. This directly shows the necessity of the simulator that we built. Our Plasma simulator focuses its behaviour on the Plasma nested blockchain(s) and is designed to be extended and improved by developers. It also has the ability to satisfy researchers' needs of understanding different metrics around Plasma.

1.5 Organization

The remaining of the thesis is organized as follows: In the next chapter we discuss all fundamental concepts needed to understand the rest of the paper. In Chapter 3 we address related works to better position our thesis. Chapter 4 describes in detail the architecture of our project together with its design and implementation, which should serve as reference to developers and should support them on better understanding the workflows and algorithms applied in the system. In Chapter 5 we perform a series of evaluations to validate and verify the system behaviour. Chapter 6 concludes the thesis by providing the current status of the system and by suggesting future improvements for the simulator.

Chapter 2

Background

The following chapter will go through the fundamentals needed in order to comprehend the rest of this thesis. First we will define the characteristics of a peer-to-peer network. Subsequently we will take a look at security concepts that a blockchain relies on. We will later explain what the blockchain actually is, what problems it solves and what kind of implementations are currently using this technology. Moreover, we will go in-depth into the internal components of a blockchain and will focus especially on Ethereum. At last, we will describe the Plasma Framework and how it intertwines with Ethereum.

2.1 Distributed System

A distributed system is a set of computers that communicate with one another via messages and coordinate to accomplish a common objective. This collection of computers (often called nodes), while working together, appears to the user as a single coherent system [32].

2.2 Peer-to-Peer Network

Peer-to-Peer (P2P) is a group of computers that communicate directly to each other and are able to exchange resources and services without the involvement of a centralized servers. Each of the computers in the network typically acts as both a server and a client *(symmetric role)* and can respectively issue requests or serve them. Furthermore, P2P systems do not require ”all-to-all communication”, which enables scalability of up to

thousands if not millions of nodes. P2P protocols allow the connection of computers and devices that can be heterogeneous in terms of hardware. This means that a node in the network can span from a simple smartphone up to a powerful super computer. [1]
Figure 2.1 illustrates how P2P nodes operate for data sharding applications. Normally, each node maintains some kind of metadata that facilitates searching.

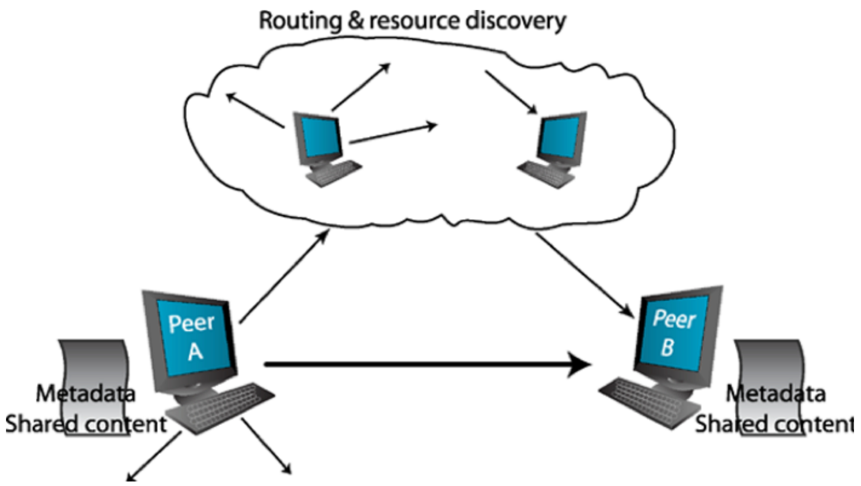


Figure 2.1: Peer A wants to send a request to Peer B. In order to do that Peer A has to locate Peer B via other peers in the network. Once Peer A found Peer B, they will communicate directly with each other. [1]

2.2.1 Architecture of P2P Systems

An architecture of a system is a cornerstone for the implementation of high-level applications on top of it. For that reason, we will take a brief look at P2P system architecture.
In general, we can categorize the P2P Systems into two wide categories - *centralized* and *decentralized*. Unsurprisingly, most of the research targets decentralized systems. When designing a decentralized P2P system one need to consider two crucial aspects - structure (flat versus hierarchical) and topology (structured versus unstructured).
There also exist a third category - a hybrid one, that incorporates both the centralized and decentralized architectures to leverage the advantages of both architectures [1].
Decentralized p2p systems are of most interest to us as blockchain systems are based on them. Some of the major advantages these systems bring are: single-point of failure resistance, high scalability and high performance.
As depicted on Figure 2.2, decentralized systems can take either a flat or hierarchical structure.

- Flat (single-tier) structure refers to the uniform distribution of functionality and load among the nodes
- Hierarchical (multi-tier) structure represents multiple layers of routing structures - e.g. corporation routes to division, division connects to department, department routes to manager.

Another design aspect that needs to be considered is the logical network topology (alias overlay network). It can be defined as unstructured or structured.

- Unstructured network topology
  - In an unstructured topology there is no predefined way in which the nodes are structured. Every node is responsible for its own data and it tracks a set of neighbours that it can propagate requests to.
- Structured network topology
  - In a structured topology, on the other hand, there exist some predefined strategies of data allocation (e.g. distributed hash table). This means that there is a mapping between data and users.

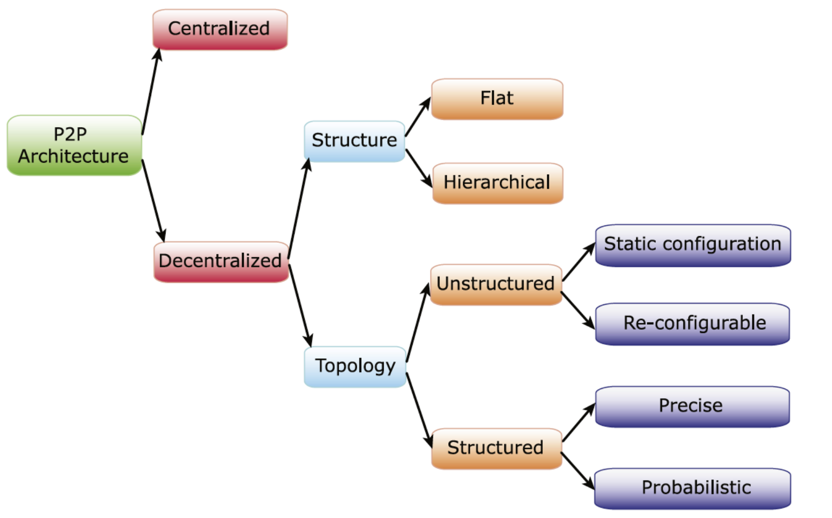


Figure 2.2: Taxonomy of Peer-to-Peer Systems [1]

2.3 Information Security

The term Information Security addresses the prevention of modification, unauthorized write or read access, misuse and destruction of information. It's main concern is to ensure Confidentiality, Integrity, Availability (CIA) of data. [33]

- *Integrity* refers to the protection of data against unauthorized and unnoticed modification.
- *Confidentiality* is reached when only authorized users can gain access to particular data.
- *Availability* means that data is available to users and applications whenever they need it. It should, under no circumstance, become unavailable due to malicious attacks to the system.

2.4 Cryptography

Cryptography is the art and science of encryption. It covers important methods for secure transfer and storage of data. Some of the most important constructs that are important to us and to blockchain are digital signatures, hash functions, asymmetric and symmetric encryption [34]. The main focus of cryptography is to ensure an accurate and protected way of moving sensitive data over an insecure communication channel as well as secure storage of data on unknown medium. An example of insecure channel at this point will be the P2P network that blockchains work on.

2.4.1 Encryption

As already mentioned parties on the internet tend to communicate in an insecure way using different kinds of communication channels. Bringing a a sensitive data from a sender A to a recipient B could be caught and read by everyone listening on the network. To prevent that we use encryption.
Encryption is a way of encoding a plain text message in such a way that only authorized parties can decode and access it. Such encoded message is usually referred as cypher text. To encrypt a message, encryption algorithms generate a pseudo-random key which they pass together with a plain text as an input to an encryption function. The output of the

encryption function is the above mentioned cypher text. In general there are two strategies for encryption of messages - Symmetric Encryption and Asymmetric Encryption.

Symmetric Encryption

Symmetric encryption is the simplest encryption as it involves one secret key for both enciphering and deciphering the transferred information. This secret key is shared between the sender and the recipient and is used to encrypt and decrypt all messages on both sides of the communication channel. Some of the most widely used symmetric encryption algorithms are Triple Data Encryption Standard (3DES) and Advanced Encryption Standard (AES) [35].

Asymmetric Encryption

Symmetric key encryption is a reasonable strategy when speed is of great importance. Nevertheless, there exist the problem of the key exchange. How can both sender and recipient exchange the same key while making sure no one else gets access to it. Furthermore, if someone needs to communicate with n other computers then he or she would need n different keys. If they built a group and each of them wants to transmit sensitive data to every member of this group, then all $n+1$ sides would need to exchange $\frac{n(n+1)}{2}$ keys. This becomes quite complex, which is why Public-Key Encryption was introduced.
Asymmetric Encryption, alias Public-Key Encryption, uses two separate keys - public and private. The public key is used for encryptions while the private key is used for decryption. Both keys are derived from one another and each user needs to generate a key pair for itself. The public key is then published with everyone while not information about the private key should be shared with others. This approach makes the exchange of keys simpler and more secure as everyone can encrypt a message (message could be a symmetric key) but only one can decrypt it. Among the most broadly used algorithms are RSA [36] and DSA [37].

2.4.2 Decryption

Decryption is the reverse operation of encryption. It takes a cypher text message and decodes it to a plain text message. Decryption should be ideally possible only for authorized parties

2.4.3 Cryptographic hash function

A cryptographic hash function takes a text string of an arbitrary size as an input and manipulates it to a fixed size of a bit string (called hash). Important requirement for hash functions is that they need to be one-way function. This means that it should be trivial to compute a hash value for any given string, but it must be impossible to compute a plain message given its hash value. Another important property is that hash functions must be constructed in such a way that they are collision resistant. Collision resistance is defined as the infeasibility of finding the same hash output for two distinct input strings.
Some of the most utilized cryptographic hash functions are MD5 [38] SHA-2 [39], SHA-3 (a subset of Keccak family, standardized by NIST) [40]. Interestingly, Ethereum uses the original proposed version of Keccak [41], which is called eth-hash [42].
Hash functions are especially useful in cases, when data's integrity needs to be verified.

2.4.4 Digital Signature

Given the knowledge we gained after considering Asymmetric Encryption and Cryptographic hash function we can define the term of Digital Signature.
Digital Signature is a way of committing to a particular message that ensures its integrity and authenticity. Commitment (also called Signature) is based on Public-Key encryption. The encryption here takes slightly different approach from what we learned in asymmetric encryption and that is the usage of the keys. When generating a digital signature of a file or a document, one **encrypts** the message with the **private key** and every other recipient possessing the **public key** can **decrypt** it. By decrypting the cypher, it can be verified whether the message was transmitted by the expected origin.

2.4.5 Application of Cryptography in Blockchain

Cryptography in Blockchain plays a significant role regarding blockchain's credibility and is applied in three ways:

- Verifying the identity of a transaction's source and hence transaction's ownership (digital signature)
- Guaranteeing the integrity of a transaction (digital signature)
- Securing blockchain transaction history, thus ensuring its integrity and reliability (one way hash-function)

2.5 Distributed Consensus

Distributed Consensus is a fundamental problem in distributed systems. The main idea behind a distributed consensus is reaching an agreement among the nodes on some issue and to synchronize nodes' state within final number of steps [32]. Consensus in blockchains is necessary to secure the network and to validate the state of the system.

2.6 Distributed Ledger

Ledger is a book or another type of collection used for recording of financial transactions of some monetary units between different accounts. The development of cryptography and Peer-To-Peer networks along with the discovery of consensus algorithms facilitated the formation of distributed ledgers.
Distributed Ledger represents a database of transactions distributed among multiple computers typically running on a unstructured decentralized P2P network (see subsection 2.2.1). Hence, there is no central authority that transmits records. Instead, records are independently propagated between various nodes and each node processes every transaction building its own conclusions. These conclusions are then again communicated to the other machines on the network in order to meet certain majority agreement, thus reaching a distributed consensus [43].

2.7 Blockchain

The examination of all necessary fundamentals allows us to explore blockchain.
Blockchain serves as a distributed ledger containing a growing list of transactions collected in blocks, which blocks are linked up by cryptography. To ensure integrity of the ledger and to prevent future modifications each block contains a hash value of the previous block in the chain. Figure 2.3 illustrates a simple blockchain.
Each blockchain is maintained by nodes (computers or every device possessing appropriate computation power) that allow it to function and survive. Normally, these nodes communicate with each other inside a peer-to-peer network. Each node has its own copy of the distributed ledger, as the longest chain (the one with most blocks) becomes officially the accepted chain.

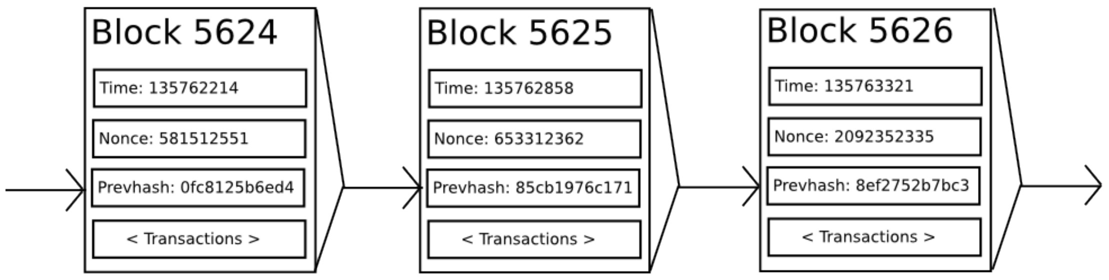


Figure 2.3: Two blocks of data linked together to form a blockchain. [2]

2.7.1 Double-spending problem

Double-spending problem is the act of spending the same digital currency more than once. It leads to the creation of counterfeit money which could lead to devaluation of the currency and distrust in the financial system [44].

2.7.2 Transaction

A transaction in blockchain is used to transfer digital currency (coins) between two parties in the network. Depending on the blockchain there exist different data structures that define a transaction. We will define two transaction models: Unspent Transaction Output (UTXO) model and Account model.

UTXO Based Transaction Model

Coins circulating in a blockchain utilizing an UTXO model are similar to cashier checks. They cannot be exchanged for custom amounts but instead one must spend the entire amount of a coin. All coins are defined as unspent transaction outputs (UTXOs) and form the state of a particular blockchain (e.g. Bitcoin [2]).
In order to spend a coin, it needs to be referenced in a transaction as an input. An input contains a reference to an existing UTXO and a digital signature produced with the private key of the coin's owner.
A transaction contains one or more inputs and one or more outputs, with each output containing a new UTXO. The newly created UTXO(s) by a transaction are then added to the state. Figure 2.4 illustrates the UTXO model.

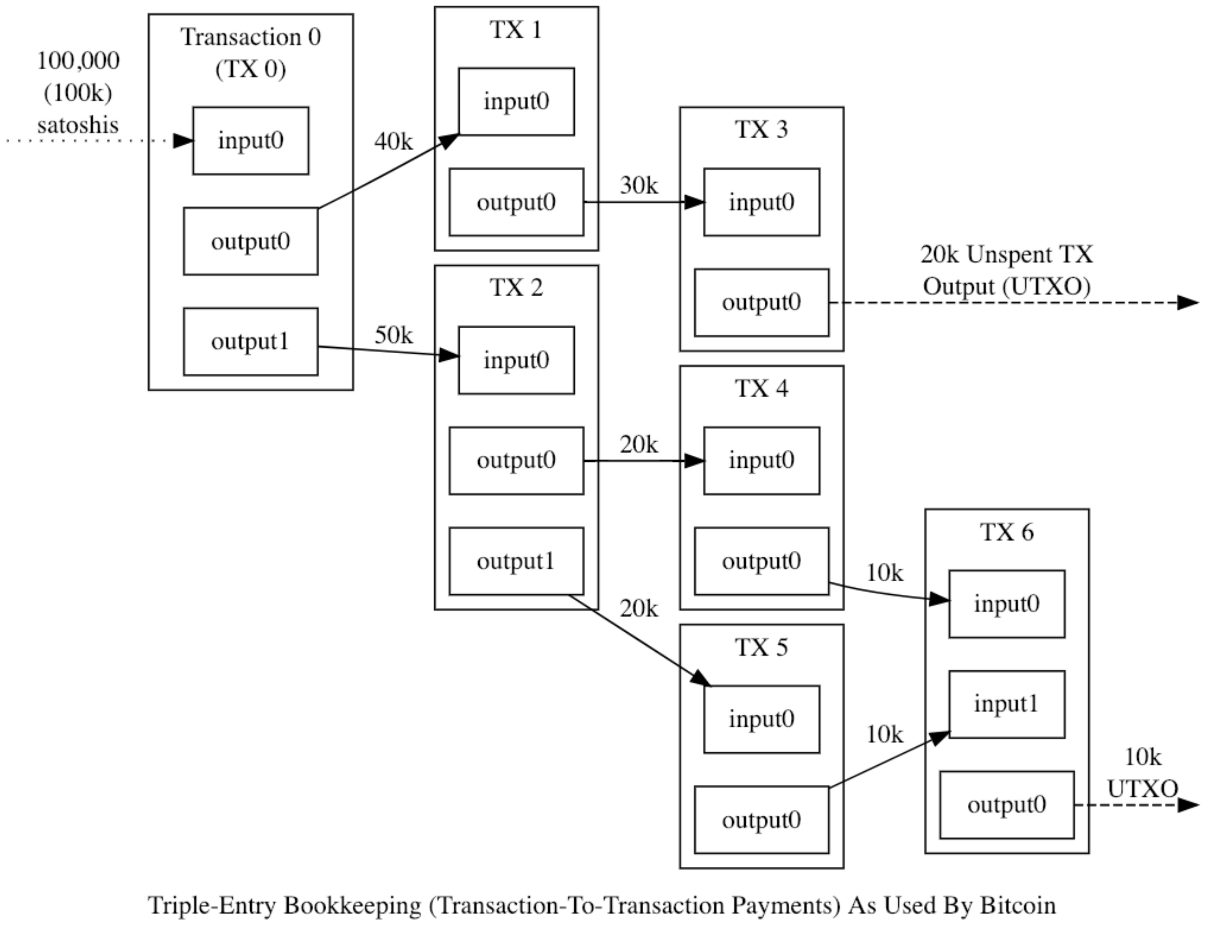


Figure 2.4: UTXO Model [3].

Account Based Transaction Model

Blockchains' record-keeping based on Account Model works analogously to that in a bank. Each participant has its own account. This account contains information such as balance and a nonce that tracks the number of transactions the account created. When a user wants to spend coins, the validator of the transaction checks whether this user's account has enough balance and then approves or ignores the transaction [9].

2.7.3 Block

A block (Figure 2.5) is a container data structure. It is composed of a block header and a list of transactions [3].
Block header incorporates data about the block called metadata. This metadata contains

different compartments such as:

- Previous block hash - since we are dealing with a chain of blocks, every new block points to the previous one based on previous block's hash (also addressed as hash pointer). This kind of referencing is crucial for guaranteeing the integrity of the blockchain. It is important to note, that the hash value of the new block depends on the hash value of the previous one.
- Mathematical puzzle - in order to reach consensus in the network nodes need to solve a sophisticated mathematical puzzle. This is a requirement for adding new blocks to the blockchain.
- Merkle tree root - the root of the constructed merkle tree of transactions (discussed later in this section)

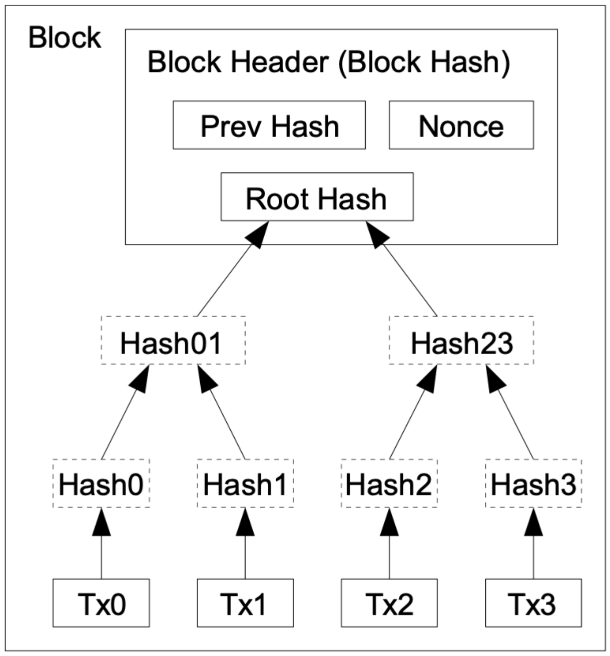


Figure 2.5: Block containing a block header and the transactions that are hashed in a Merkle Tree [2].

When a node attempts to add a block, it first needs to find a solution to the mathematical puzzle (as discussed above). While it is difficult to solve the puzzle, the verification is trivial. Right after the puzzle is solved, the node adds the block to its blockchain and propagates it to the other peers on the network. When a peer receives a block it will verify it and if it is part of the longest blockchain, it will add it to its local blockchain.

2.7.4 Merkle Tree

As described earlier a block in a blockchain contains a list of transactions. We also mentioned that hash functions can be applied in order to verify whether some data has been modified or not.
Merkle tree [45] is a data structure, similar to binary trees that allows efficient and secure verification of consistency of data. It summarizes all transactions in a block by generating a digital fingerprint of the entire set of transactions, thereby allowing a user to audit whether a transaction is included in a block or not. The digital fingerprint is often called Merkle root and corresponds to the root of the tree. All nodes in the tree contain a hash value except the leafs, which are regular transactions. [46]

Construction

The tree can be constructed by taking the leafs (list of transactions) and generating a hash value for each transaction. Every two of the resulting values are then concatenated, and the result is hashed. This process is repeated until the root is reached. Figure 2.5 demonstrates this process.
Once built, one can verify that a transaction is part of the tree in $\mathcal{O}(\log n)$, where n is the number of transactions, using only the root hash (*i.e.* Merkle Proof).
Auditing works by recreating the branch starting from the transaction (leaf node) and climbing up to the root. If the root hash corresponds to the original root hash of the block then the transaction is truly part of the block.

2.8 Mining

Mining is a mechanism that allows blockchains to achieve decentralized security. It is a process of adding new blocks in the blockchain executed by some nodes in the network called miners. Miners collect and validate new transactions with the goal to pack them into a block and add them to the distributed ledger. To add a new block, miners compete to solve a difficult mathematical puzzle. The solution of such puzzle is called Proof-Of-Work and it proves that a miner has spent a lot of resources to find the solution. If a block is successfully mined, it is first being propagated to other peers in the network and in case it becomes part of the longest chain, miners get rewarded.

2.9 Bitcoin

The first blockchain was invented in 2008 by Satoshi Nakamoto and was given the name Bitcoin [2]. Bitcoin is the first digital currency that solved the double-spending problem without the need of a trusted authorities such as banks. The design of Bitcoin inspired all future blockchains which digital currencies are often called altcoins [47].

2.10 Ethereum

Ethereum is a decentralized system that inherited the technology of Bitcoin - blockchain and expanded its capabilities. One of the most important contributions of Ethereum is that it allows users to develop decentralized applications and run them on Ethereum. This makes Ethereum a distributed software platform, which represents the main difference compared to Bitcoin. The precise definition of Ethereum as state in its white paper [9] is the following:

"Ethereum is a blockchain with Turing complete programming language allowing anyone to write smart contracts and decentralized applications where he or she can create their own arbitrary rules for ownership, transaction formats, and state transition functions."

Based on this definition, we will explain the main building blocks of Ethereum that are relevant for this thesis.

2.10.1 Smart Contracts

The idea behind smart contracts was originally introduced by Nick Szabo in 1996 [8]. Smart contracts facilitate the exchange of anything of value such as money, liens, bonding, shares, property, etc. without a trusted third party such as court. Ethereum applied this idea as a computer source code running on the top of a blockchain containing a set of rules under which the participating parties agree upon.

2.10.2 Ethereum Virtual Machine (EVM)

Since we know what a smart contract is, we can further explain how a smart contract code gets executed on a blockchain.
Users participating in Ethereum can develop their own applications in the form of a smart

contract and upload them to the network. Each node on the network stores that code as bytecode that can later be executed. The execution of the smart contract bytecode takes place in the Ethereum Virtual Machine (EVM). EVM is a *quasi*-Turing-complete machine that executes smart contract codes and so alters the system state. *"The quasi qualification comes from the fact that the computation is intrinsically bounded through a parameter gas, which limits the total amount of computation done"* [48].

2.10.3 Ethereum accounts

Ethereum employs the Account Based Transaction model. Hence, the state of the blockchain is made up of objects named **accounts** where each account has an address, balance and a nonce. The nonce describes the total amount of transactions an account created and serves as security mechanism against replay attacks.
Ethereum distinguishes two types of accounts:

- Externally Owned Accounts - accounts controlled by private keys, thus external clients. They can send messages by creating and signing a transaction with their private keys.
- Contract Accounts - accounts controlled by their source code. Such accounts are only activated by receiving messages by externally owned accounts or another contract accounts. Upon receiving a message, the contract's code executes allowing it to change its internal storage or to send messages to other contract accounts.

2.10.4 Gas

Smart Contracts can be implemented in Turing-complete languages. This is could be a potential weakness in Ethereum's security as attacks such as Distributed Denial of Service attack (DDoS attack) can be easily applied. To prevent such exploitations as well as to avoid other issues of the network, all programmable computation in Ethereum is subject to fees. The fee is specified in units of **gas**. Hence, every piece of programmable computation is a particular cost in terms of gas.

2.10.5 Ethereum Transaction

Each transaction in Ethereum has a specific amount of gas associated with it: **gasLimit**. This is the amount of gas that each transaction creator is willing to purchase, so that the transaction can be executed by the nodes. The purchase is made at some **gasPrice**, which is also part of the transaction. Thus, when sending a transaction, its initiator has to posses at least $gasLimit * gasPrice$ coins (in Ethereum the coin is called **Ether**) in his account balance, otherwise the transaction will be considered as invalid. It is called **gasLimit** as any unused gas will be refunded to the sender's account after the transaction execution.

2.11 Consensus algorithms

To reach a consensus within a decentralized distributed system, nodes in the network must come to a mutual agreement for the state of the system. In this section we will examine some of the widely used consensus algorithms.

2.11.1 Proof-Of-Work

Proof-of-Work is the first consensus algorithm used in blockchains. To reach consensus nodes in the network compete against each in solving a mathematical puzzle for each new block (mining). Once a solution for the puzzle is found, the block together with its solution is propagated to the network. Upon receiving the block, other nodes can validate the correctness of the solution and the block is added to the blockchain. While finding a solution is extremely resource intensive and time-consuming, validation is effortless and fast. The effort that nodes put into solving the puzzle is related to extreme usages of electricity and CPU power, which represents an immense obstacle for new and light users to join the mining process of blockchains.

2.11.2 Proof-Of-Stake

Proof-Of-Stake is an alternative algorithm for reaching distributed consensus. Recently, it became a well-known solution for blockchains, due to the problems of Proof-Of-Work. In essence, the next block validators is selected in a semi-random process based on validator's stake and a piece of randomness. User's stake refers to the amount of cryptocurrency this user has deposited as a collateral. The more cryptocurrency a user stakes the better chances it has to be the next block validator. The randomization mechanism prevents the system from centralization, otherwise the wealthiest individual will always be selected as the next block validator and will continuously increase their wealth and as a result control the entire system.

2.11.3 Proof-of-Authority

Proof-of-Authority is another replacement consensus algorithm for Proof-of-Work. It is based on the selection of certain authorities in a chain - nodes that are allowed to create new blocks and secure the blockchain [49]. The majority of the authorities need to agree on the latest block to be added to the chain, in which case the block becomes part of the permanent record. Since, this algorithm is rather centralized than decentralized, it is most widely used in private chains.

2.12 Plasma Framework

As mentioned in Chapter 1, Plasma is a proposed network for composing multiple off-chain blockchains (also addressed as sidechains, plasma chains, child chains) that maintain a local state of transitions (transactions and blocks) with the fundamental goal to increase the scalability of any smart contract enabled blockchain (usually called main, primary or root chain). Each of these off-chains should have its own consensus mechanism. It is important to note, that an off-chain could be used for scaling many different smart contract applications (Figure 2.6) beside transactional payments [4].
The basic idea behind the framework is to take some assets from the main chain and to transfer them to another off-chain blockchain by locking up the main chain assets in a smart contract and generate them on the sidechain. Such assets can change their owner multiple times in the plasma chain by submitting transactions and creating blocks. A plasma chain user can at any time decide to withdraw its funds, which the smart contract executes by destroying the assets on the off-chain and unlocking them on the main chain.

2.12.1 State commitment

Plasma blockchains do a majority of their work outside of the primary chain. This work transforms the local state multiple times without informing the main chain about each change. This multiple off-chain state transformation, based on thousands or even million transactions makes the main chain more scalable. This scalability comes at the cost of

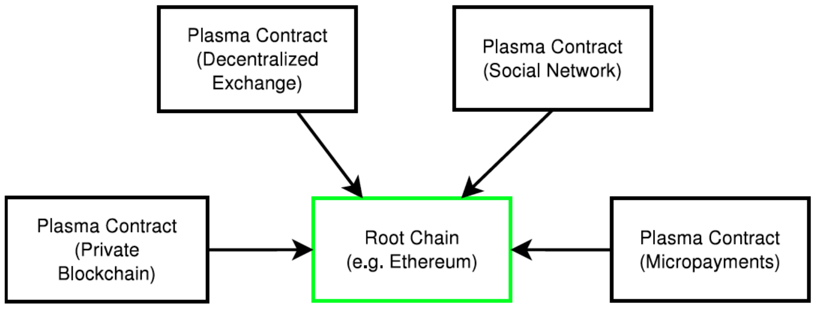

Plasma Contract (Decentralized Exchange)
Plasma Contract (Social Network)
Plasma Contract (Private Blockchain)
Root Chain (e.g. Ethereum)
Plasma Contract (Micropayments)

Figure 2.6: Plasma chains can be created for smart contract scalability for many different cases [4].

lacking finality on the Plasma chains, as user assets are still locked on the main chain. To make off-chain changes final, Plasma chains must commit a compressed version of their state to the root chain in the form of a cryptographic hash value called **state commitment**. These commitments represent a save points of the chain and correspond to a single block merkle root. That is, whenever a block on a sidechain is mined, its block root hash is sent to the main chain as a transaction to the smart contract managing the sidechain. The smart contract keeps track on all state commitments that are later used for validation of user exits.

2.12.2 Fraud proof

To protect plasma chains against different attacks and misuses such as block withholding attack, to guarantee its security and reaching a consensus, sidechains participants create fraud proofs (Figure 2.7). Fraud proofs represent merkle proofs with the goal to prove that a certain chain tip is invalid. To better illustrate what a fraud proof is, let us give an example: A node working on creating blocks on a plasma chain includes an invalid transaction into a block and sends the block header hash as state commitment. As all other participants in the network validate all generated blocks, they will encounter the invalid transaction. Participants can then create a fraud proof to show that the transaction is indeed invalid and send this proof to the corresponding smart contract. The contract will verify the fraud proof and reject the fraudulent block.

2.12.3 Exit

State commitments are especially useful in cases when a user wants to withdraw its funds from the Plasma chain. To withdraw the assets (usually referred as exiting the sidechain)

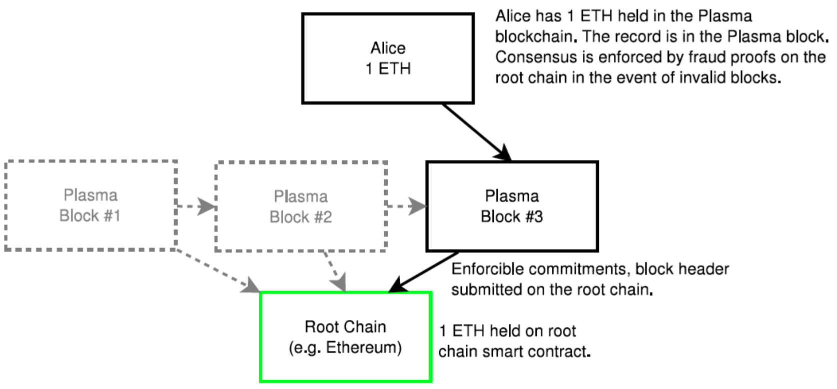

Alice
1 ETH
Alice has 1 ETH held in the Plasma blockchain. The record is in the Plasma block. Consensus is enforced by fraud proofs on the root chain in the event of invalid blocks.
Plasma Block #1
Plasma Block #2
Plasma Block #3
Enforcible commitments, block header submitted on the root chain.
Root Chain (e.g. Ethereum)
1 ETH held on root chain smart contract.

Figure 2.7: Plasma chain within a main chain. Instead of disclosing all the contents of a plasma blockchain, block header hashes are submitted on the primary chain. In case of fraudulent activity, fraud proofs are sent to the root chain and then the faulty block is rolled back and the creator is penalized. [4].

a user needs to prove to the smart contract that it owns these assets. This is possible by generating Merkle proofs. Exits can also be performed in the event of faulty Plasma chain as shown on Figure 2.8.

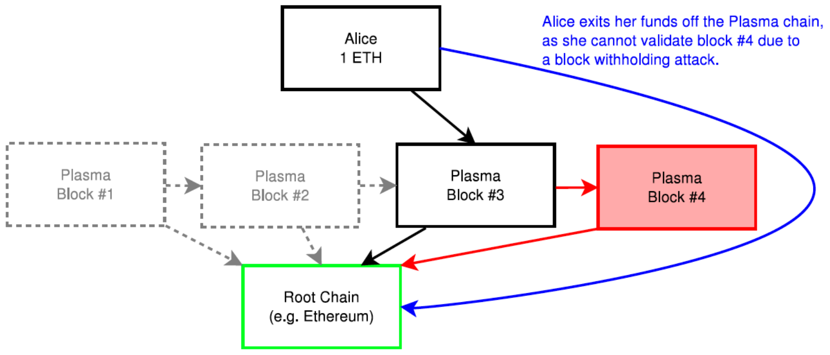

Alice
1 ETH
Alice exits her funds off the Plasma chain, as she cannot validate block #4 due to a block withholding attack.
Plasma Block #1
Plasma Block #2
Plasma Block #3
Plasma Block #4
Root Chain (e.g. Ethereum)

Figure 2.8: Exit of funds in the event of block withholding. The red block (Block # 4) is a block which is withheld and committed on the root chain, but Alice is not able to retrieve Plasma block # 4. She exits by broadcasting a proof of funds on the root blockchain [4].

2.12.4 Mass-exit

When a fraudulent Plasma chain is at place, all participants have the opportunity to exit the sidechain based on the last valid state with minimal losses. In case all participants leave the off-chain, they are executing a mass-exit.

2.12.5 Blockchains in Blockchains

The design of Plasma allows a construction similar to a court system. A hierarchical tree of nested blockchains can be composed, so that each further level down the tree makes state commitments periodically to its direct parent (Figure 2.9). This allows for great scale of computation as the process of work done is similar to MapReduce function. Map phase corresponds to commitment and assignment of work to child blockchains. Each child blockchain on n-th depth level achieves many state transitions, hence generating blocks. Reduce phase refers to the state commitment to the n-1st depth level. The hash committed to the root chain encompasses very high amount of data and work.

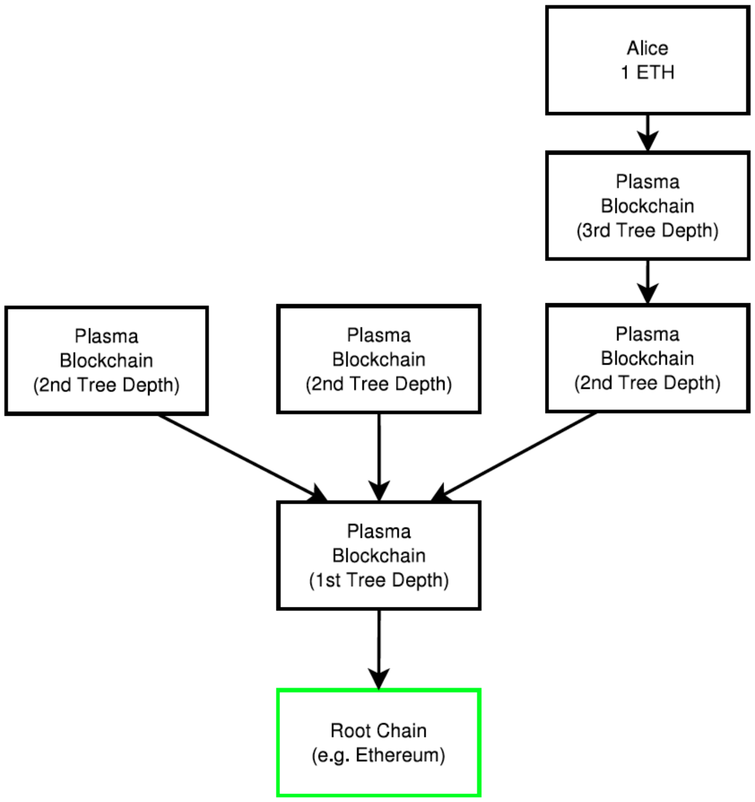

Alice
1 ETH
Plasma Blockchain (3rd Tree Depth)
Plasma Blockchain (2nd Tree Depth)
Plasma Blockchain (2nd Tree Depth)
Plasma Blockchain (2nd Tree Depth)
Plasma Blockchain (1st Tree Depth)
Root Chain (e.g. Ethereum)

Figure 2.9: Plasma nested blockchains. Each chain reports its compressed state to its parent ultimately being reported to the root chain [4].

Chapter 3

Related Work

In this chapter we discuss different approaches that aim to simulate either peer-to-peer networks or blockchain systems. In this way we can better position our simulator comparing it with the other solutions.

3.1 Ganache

Ganache is a tool for creating a personal blockchain that is mainly used for Ethereum development [24]. Using it one can deploy contracts, execute commands, run tests and explore the insights of blocks and transactions. Furthermore, it provides the opportunity to create accounts and configure advanced mining for setting block times. Overall the Ganache allows the users to adjust the following settings:

- Network connection settings such as hostname, port, network ID, etc.
- Number of accounts that Ganache will create.
- Chain settings such as gas limit and gas price per block and per transaction.

Ganache represents a great tool for development. When a developer implements a smart contract Ganache is the environment that will greatly support testing the contract. Nevertheless, this approach highly deviates from what we try to achieve. The main differences between Ganache and Plasma simulator are listed as follows:

- Event though, a smart contract can be tested in Ganache, there is no way to connect a different software to this smart contract. This is quite relevant for us, since we want to span a sidechain that needs to be part of Ethereum blockchain and at the same time create its own environment for creating transactions and blocks.

- Ganache is aimed mainly for development. This means that its users are rather interested in smart contract development than to improving blockchain protocols and frameworks. Our main goal, on the other hand, is to provide a way to analyze and experiment with the Plasma framework.
- The internal network of Ganache is not transparent to its users, hence they cannot analyze metrics such as block propagation time.

3.2 Testnets

During our research we found three testnets that are currently in use. Each of them behaves similarly to a production Ethereum blockchain and developers user them to test their planned upgrades. These three testnets are:

- Ropsten - A PoW network that resembles Ethereum. Mining of Ether is possible [26].
- Kovan - A PoA blockchain, started by Parity Tea. Ether cannot be mined [27].
- Rinkeby - A PoA blockchain, started by Geth team [28].

Even though these testnets simulate the behaviour of the actual Ethereum, one can use them only by connecting to them using specified for this purpose tools. This highly limits the possibility of exploring the Ethereum blockchain internal interactions as well as it hinders the ability of connecting external software subsystem such a sidechain.

3.3 HIVE

HIVE is an Ethereum end-to-end test harness [25]. Its main goal is to test different implementations of clients validating conformity to basic quality attributes and clients interoperability. Test can be written in any programming language, keeping in mind that harness is meant to do black box testing without the possibility of inspecting client's internal state. The focus is rather put on adherence to official specifications under different circumstances.
HIVE supports the simulation of multiple clients, running simultaneously under specific

conditions in order to monitor clients' behaviour. Such simulation enables the examination of clients' interactions without explicitly creating a blockchain.

3.4 PeerSim

PeerSim is a peer-to-peer simulator built in Java that focuses mainly on simulating different network protocols. The simulator can reach extreme scalability by running thousands of nodes simultaneously. It is composed of two simulation engines - cycle-based and event-driven engines. Using these engines one can write communication protocols and experiment with them.
Due to the ability of the simulator to be extended with a variety of network protocols, implementation of a simple blockchain protocol is possible. The focus of such simulation would rather lie on analyzing the properties of the network protocol a blockchain uses (e.g. structured vs. unstructured) with the purpose to improve the network connectivity and robustness. Moreover, PeerSim can be extended only in Java, which deviates from our goals for polyglot extensibility. Based on the documentation provided on the official web page of the simulator, connection between multiple instances of the simulator seems rather unrealistic. This condition is relevant to our approach, as we want to simulate multiple blockchains.

3.5 VIBES

Visualizations of Interactive, Blockchain, Extended Simulations (VIBES) is a configurable blockchain simulator that is capable of conducting general-purpose blockchain simulations in order to analyze the insights of a blockchain system [5]. The system architecture is based on the Actor Model [50], hence the simulator achieves large-scale network simulations on an event-level. Simulations can be conducted both on a single computer and on a distributed cluster of machines. VIBES served as a starting point for our work, thus we will discuss it in more details.

3.5.1 Architecture

The architecture of VIBES is illustrated on Figure 3.1. The main component of VIBES is the Orchestrator. It is responsible for managing the entire simulation by sending

coordination messages to all further nodes in the system. It monitors the system state and takes care of reaching consensus between the blockchain nodes. Blockchain nodes are constructs that show similar behaviour to Bitcoin mining nodes, thus creating blocks and extending the blockchain.
VIBES offers an user interface (UI) where users can start/stop a simulation and monitor outputs of a simulation such as number of blocks, propagation time, number of nodes, etc. The outputs propagated to the UI are produced by a Reduce, which collects all results during a simulation and then transmits them to the Orchestrator when the simulation ends.

3.5.2 Issues

Technology

VIBES is implemented using akka [51] framework in the Scala programming language [52]. This particular selection of technology represents an obstacle for the future extensibility of the simulator because developers are required to implement new features using only akka and scala. This deviates from our goals, to build a system that is polyglot and can be extended using various programming languages.

Extensibility via side chains is not possible

The design of VIBES currently does not allow implementation of extensions such as side chains, since the Orchestrator controls the entire simulation system including the simulated blockchain. Adding multiple off-chains on top of it will overload the synchronization tasks the Orchestrator needs to perform, which will inherently slow down the whole system.

Real-time UI output

The web app VIBES maintains, receives simulation results only at the end of a simulation. Thus, there is no real-time updates of the UI, which harms the user experience and reduces user-system interactions.

Transaction generation

Transactions in VIBES are created by every blockchain node, but their validity and correctness is not considered, as a transaction can contain a random number of tokens. This means, that transactions are simulated only based on their quantity and not their quality and correctness.

Conclusion

Even though, VIBES has some issues regarding technology choice, extensibility and user experience, it is a great tool for simulating general-purpose blockchains. It inspired our system architecture and facilitated the design of the Plasma Simulator.

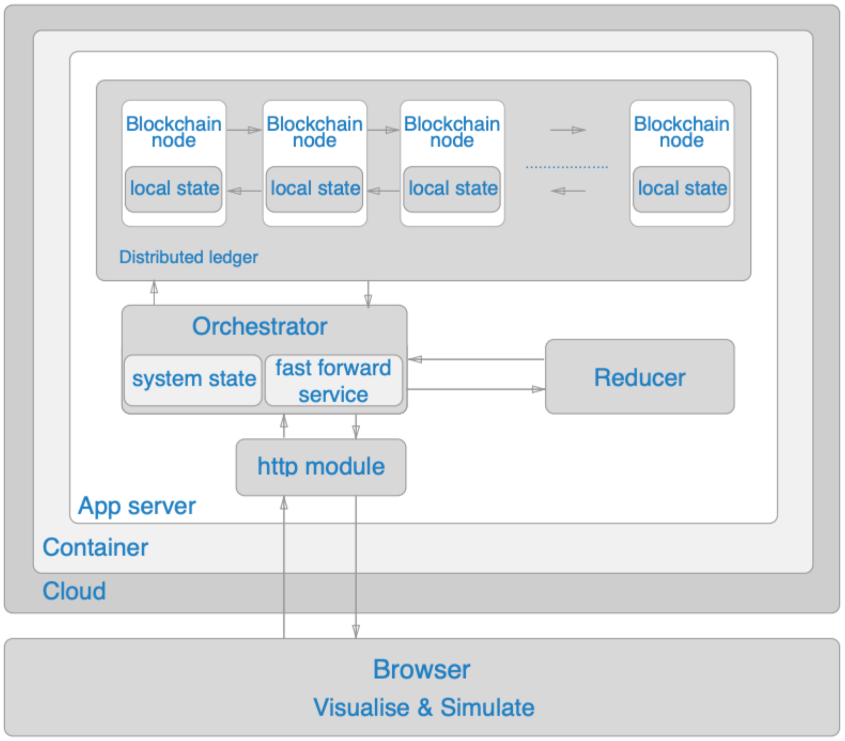


Figure 3.1: VIBES Archictecture [5]

3.6 eVIBES

At the time of writing this thesis a continuation project of VIBES called eVibes (Ethereum VIBES) was published [53]. Its main purpose is to simulate an Ethereum Blockchain by

replicating features such as the account transaction model, state transitions and smart contracts.
Users of eVIBES are able to monitor various metrics of a simulation and are allowed to upload their own contracts. It claims that sidechains are possible to implement, which is a good opportunity for our simulator, since we simulate Plasma blockchains, which effectively are considered as sidechains. At the time of publishing, we already have implemented most of our simulation system and eventual interoperability between our simulator and eVIBES simulator is rather seen as future work. It is worth mentioning, that eVIBES is implemented with the same technology as VIBES, which could constitute a hindrance for future interoperability.

Chapter 4

Approach

In this chapter we will discuss the design and architecture of the Plasma simulation system as well as the most important workflows and concepts. At first, we will go through the requirements the simulator should meet and will later on introduce the system architecture as well as the implementation details of the system.

4.1 Requirements

Blockchains are run on e decentralized network containing many computer machines. Hence, the simulator should be able to simulate hundreds of nodes that are working simultaneously on heterogeneous tasks such as transaction processing, block creation and data propagation. To be able to conduct such simulations on a single machine, we should make us of the multi-threaded environment each modern computer possesses. Furthermore, we should be able to run simulations with thousands of nodes, which can be done either by vertical or by horizontal scaling. Vertical scaling refers to the adding of additional resources on a single machine, whereas horizontal scaling addresses adding multiple machines to our pool of resources. Horizontal scaling is another requirement that our simulator should meet, which is why we should be able to span a single simulation over multiple ideally heterogeneous computer machines.

Blockchains usually employ unstructured P2P networks in order to achieve decentralization. As we want to run simulations in scalable fashion on a single computer, we need to simulate sockets and TCP connections at an event-level. Thus, our simulator must be message-driven.

Another important requirement is the opportunity for developers to extend our simulator

with additional features. If that can be done in only one programming language, this would limit the interest to our simulator only to developers that know the programming language we chose. To give possibility to developers to develop features in a programming language of their choice, we should use a technology that is polyglot.
To meet all these requirements, we implemented our simulator as a reactive system, that is scalable, message-driven, responsive, resilient and robust.

4.2 Concept

Before explaining our architecture and design we should first explain the concept behind our simulator. As mentioned in Chapter 2, Plasma is a *framework* for building scalable applications. There is no single project that is called ”Plasma”. Rather, the paper written by Vitalik Buterin and Joseph Poon describes the main building blocks of Plasma but do not consider any particular implementation. This is the reason why there are Plasma projects such as Plasma MVP, Plasma Cash and Plasma Debit. All these projects have different ideas for a particular utilization of the framework and one of our goals is to allow developers to extend our system in an easy and feasible way in order to simulate their software projects. Hence, we built a simulator that represents the backbone of a Plasma software. It consists of components that a Plasma project would need in order to simulate different workflows independent from consensus algorithms and transaction models.
Our Plasma chain characteristics are strongly inspired by Plasma MVP, as this project addresses the minimal viable product and should enable multiple future projects to extend it.

Our system simulates both an Ethereum blockchain and Plasma blockchains. The simulation of Ethereum contains only the most important constructs for our proof-of-concept simulator such as mining nodes, block creation and validation, transaction creation, data propagation and smart contract execution. It is important to note, that we mimic Ethereum’s transaction execution without consideration of EVM and data trie structure. Currently, we support only one running smart contract, responsible for managing a Plasma chain.

A Plasma chain, on the other hand, follows the UTXO based transaction model. It enables high-throughput payment transactions created by Plasma clients and propagated to the

other peers on the network. For a consensus algorithm, we selected the Proof-of-Authority (PoA) algorithm, where a node called Operator takes the mining responsibility and effectively runs the entire Plasma chain. The decision to use PoA is based on the following two reasons. First, we built our system to be fair, which means that neither clients, nor Operator would try to cheat by performing diverse attacks. Second, we wanted to reduce implementation complexity related to algorithms such as PoS or PoW. This approach could be normally used in real applications since the ultimate source of truth remains the Ethereum blockchain and in case of malicious activity in a Plasma chain, all clients can still perform a safe exit.
Currently, the Plasma contract supports functions such as asset deposit, block submission and asset withdraw. In the Plasma MVP Specification [54] the Plasma contract supports exit challenges for cases of fraudulent behaviour, but since our simulator is developed to be honest, we did not consider such method. Nevertheless, the simulator can be extended to simulate vicious attacks, where such method will be especially valuable. Such extension represents a great option for future works.
Let us take a look at the methods our Plasma contract supports.

- Asset deposit - when an Ethereum account wants to participate in a Plasma chain, they need to send a deposit transaction to the Plasma contract. This transaction will be added as a single block to the Plasma chain, which opens a balance for this Ethereum account on the Plasma chain and turns them into Plasma client.
- Block submission - when a block on a Plasma chain is mined, the operator must propagate the block's merkle root as a transaction to the Plasma contract, so that the contract is up-to-date with the state of the Plasma child.
- Asset withdraw - when a Plasma Client decides to leave the Plasma chain, they need to send a withdraw transaction to the contract, which would destroy their assets on the Plasma chain and will unlock their assets on the Ethereum blockchain

One of the most important contributions our simulation makes is the simulation of second level Plasma chains. That is, when an Ethereum Plasma Contract has a first level Plasma child chain, which itself has one or more second level child chains. To the best of our knowledge, this is the first attempt to implement more than one level Plasma chains, which brings a unique opportunity for Plasma developers to simulate how multi-level Plasma chains behave regarding security.

4.3 Event-Driven Architecture Pattern and Vert.x

Event-Driven architecture pattern is a well-know distributed asynchronous pattern used to produce highly scalable applications [55]. The event-driven architecture is composed of highly decoupled, single-purpose event processing components that asynchronously receive and process events over an event-loop. We call such components verticles (Figure 4.1) based on the utilized for this simulator reactive framework Vert.x [6]. Received events can take different form such network buffers, timing events or messages sent by other verticles. The event-loop (Figure 4.2) is a typical construct for asynchronous programming models that waits and dispatches events in the system. It is always attached to a thread as Vert.x attaches two event loops per CPU core thread. This means that a regular vertical always processes events on the same thread, thus there is no need to use thread coordination and synchronization mechanisms to manipulate a verticle state.

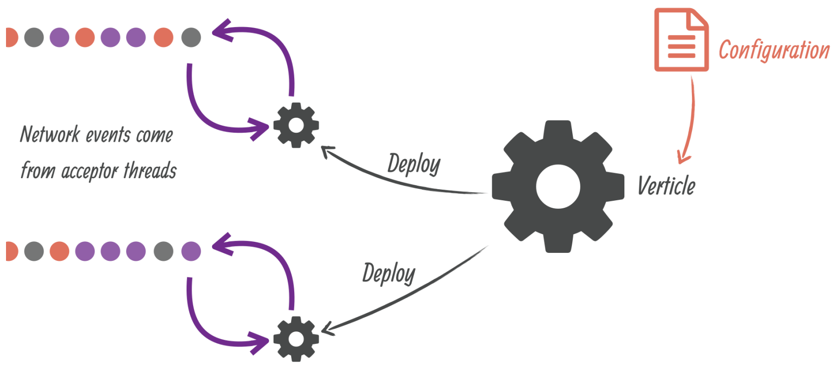


Figure 4.1: A verticle can be passed some configuration and can be deployed multiple times [6].

Communication between different verticles is enabled by asynchronous message passing over an event bus . The event-bus allows passing any kind of data. Given our requirements we will use the most preferred exchange format - JSON, since this allows verticles written in different programming languages to communicate. A simple communication using the event bus is described on Figure 4.3.
One of our requirements is to run our simulator both on a single machine and as a distributed system on multiple computers. The event bus helps us meet this requirement as it can be used not only on a same process in a virtual machine, but can be also distributed when a network clustering is activated. This allows verticles from different networks to communicate with each other over the same event bus.

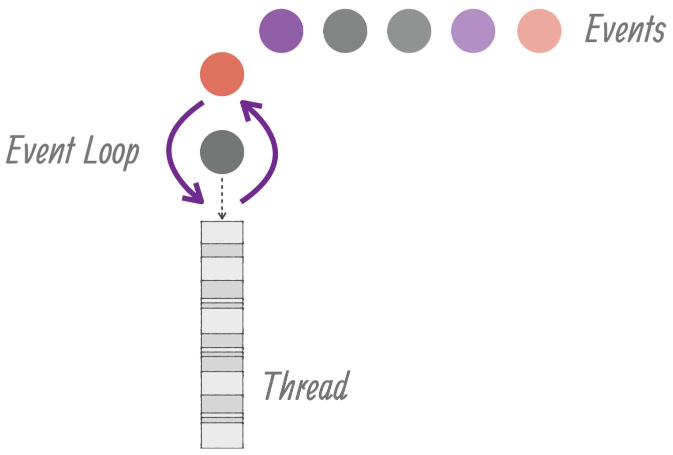


Figure 4.2: Event loop is always attached to a thread [6].

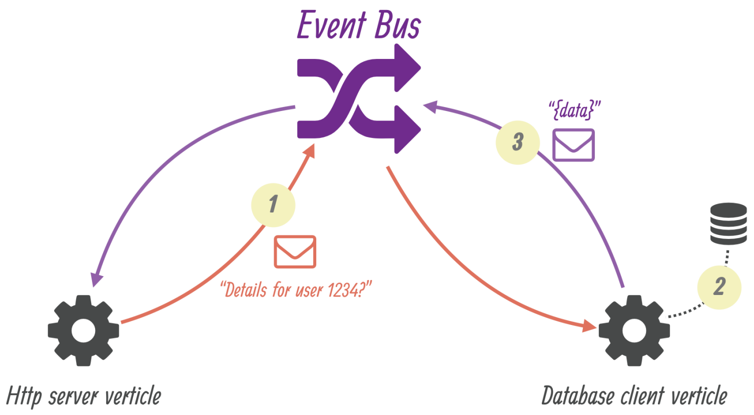


Figure 4.3: An HTTP verticle sends a request to the database verticle over the event-bus. The database verticle performs an SQL query and responds back to the HTTP verticle [6].

4.4 System Architecture

After we discussed the concept behind our solution and examined the model that allows us to built a highly-scalable simulator, we present our system architecture. It is illustrated in Figure 4.4 and the individual components are discussed below.

4.4.1 Simulation Manager

Simulation Manager is the main verticle in our simulator and the first one that is being started. It is responsible for setting up the simulation by creating Ethereum Manager,

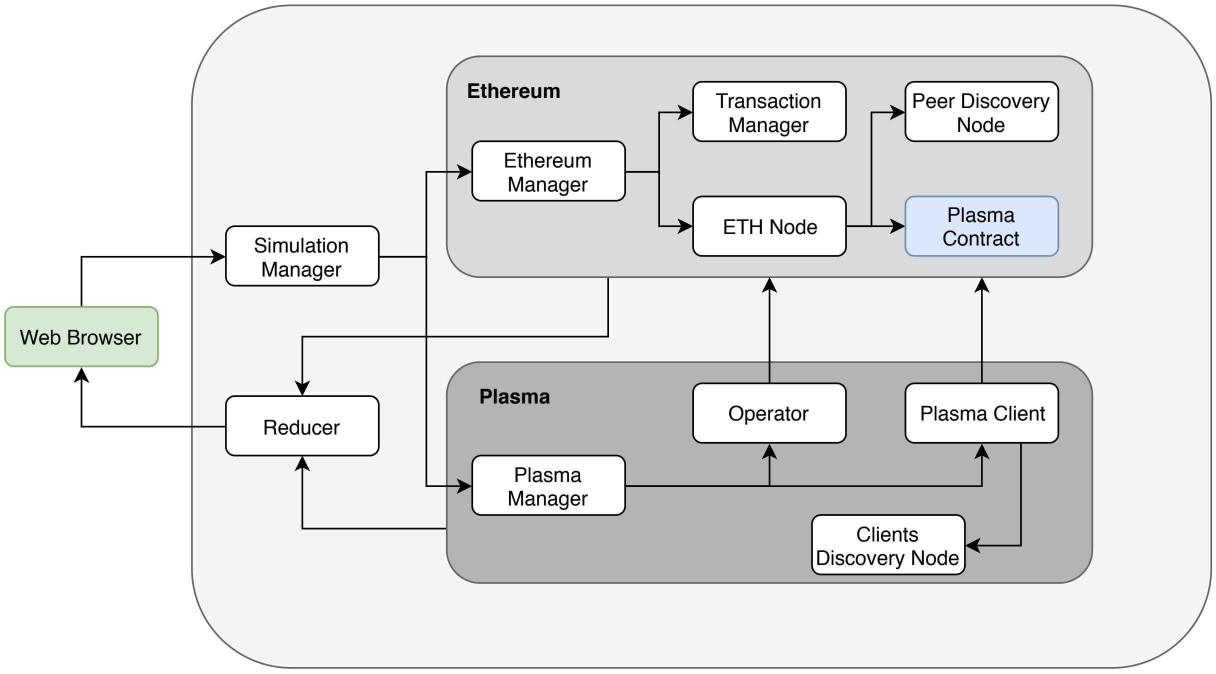


Figure 4.4: Top-level Architecture. All white components are implementations of a verticle.

Plasma Manager, Peer Discovery Node and Simulation Manager and sending them the previously received configuration from the Web Browser. Furthermore, it is also the only component that can start or stop the simulation. These actions are triggered from the simulator user. All further relevant actions that need to be taken in Ethereum and Plasma environments are delegated respectively to Ethereum Manager and Plasma Manager.

4.4.2 Ethereum Manager

Ethereum Manager is the main component that controls the workflows in Ethereum environment. It is responsible for creating all verticles in the environment besides Peer Discovery Node - Ethereum Node and Transaction Manager. It also receives the configuration parameters from the Simulation Manager based on which it can start a timer for periodical generation of external transactions (user transactions).

4.4.3 Ethereum Node

Each Ethereum Node represents a peer in the simulated Ethereum P2P network as it continuously receives messages from the other peers (Ethereum nodes, Plasma clients, or Operator) that are being processed, validated and further propagated to the other nodes.

It is the only component that moves the Ethereum blockchain forward by generating new blocks.

4.4.4 Transaction Manager

If a simulator user enabled the generation of external transactions on Ethereum, then the Transaction Manager becomes a significant component in the network as it starts generating transactions and propagating them to the other peers.

4.4.5 Peer Discovery Node

To simulate a P2P Network on an event-level, verticles need to know the peers they can communicate with. The exclusive source of connectivity in our Ethereum Network is the Peer Discovery Node. It provides each peer with the necessary information about their peers.

4.4.6 Plasma Contract

In comparison with the other components in our diagram, the Plasma Contract is the only one that is not a verticle. It is responsible for enforcing the state in a Plasma blockchain and is being called by an Ethereum Node when a proper transaction is being executed. We will discuss the Plasma Contract in further detail in the implementation part.

4.4.7 Plasma Manager

Plasma Manager is a single verticle that controls the workflows inside a Plasma chain. It generates at least one Plasma chain containing a Plasma Client, Operator and a Clients Discovery Node. Based on the provided configuration it can also span multiple second-depth level Plasma child chains connected to a single first-depth level Plasma Chain. Furthermore, it starts a timer, similar to the Ethereum Manager, for generation of client transactions within each Plasma chain.

4.4.8 Plasma Client

Plasma Clients simulate the behaviour of real clients that would use a Plasma blockchain. They create and send deposit transaction, through which they become part of a Plasma chain and generate transactions every time the timer on the Plasma Manager ticks.

4.4.9 Operator

Operator is a verticle that is mainly responsible for moving the Plasma chain forward. It receives transactions from all Plasma clients and collects them into blocks that are being propagated to the Ethereum network. It also receives deposit transactions from a Plasma contract, for each of which it creates a block on the Plasma Chain. In case, second-depth level blockchains are started, it is the only component that manages the communication between the different chains.

4.4.10 Clients Discovery Node

In order to create valid transactions, each Plasma client needs to know the addresses of all other clients. Clients Discovery Node is the construct in every Plasma Chain that provides each Plasma client with all necessary information about the remaining clients.

4.4.11 Reducer

Reducer is a verticle that summarizes information about valuable interactions happening during a simulation and transfers them in real time to the Web Browser, where the data is being presented to the user.

4.5 System Design

System architecture gave us an overview over the main components of the system and their interactions. In this section we will examine how each component behaves internally and we will introduce further helpful components.

4.5.1 Simulation Manager

The main goal of a Simulation Manager is to configure properly a single simulation based on the configuration provided by the Web Browser user. After all parameters are set by the user, they can at any time decide to start a simulation. Starting the simulation is a responsibility of the Simulation Manager as well as stopping it. When an attempt to start a simulation is triggered, the Simulation Manager checks whether a simulation has been already started and if that is the case then it refuses the attempt. If not, then it starts the simulation by creating the Ethereum Manager, the Plasma Manager, Peer Discovery Node and the Reducer. At the time of the simulation start, the Simulation Manager starts a timer, that is used to track the time of the entire simulation.
Based on Figure 4.4 one may ask why the Simulation Manager starts the Peer Discovery Node, when the Peer Discovery Node is placed in the Ethereum environment. From a logical perspective of view, Peer Discovery Node is a verticle used only by Ethereum peers. Nevertheless, the Plasma participants (Plasma Client and Operator) should take part in the Ethereum P2P network in order to communicate with the other Ethereum Actors.
Stopping the simulation is executed in two cases. Either when a user manually stops the simulation from the browser or when the simulation finishes. When stopping, the Simulation Manager notifies all verticles it created, so that they can destroy their states. After their states are destroyed, the Simulation Manager destroys the verticles, which leaves the system with only one verticle running - Simulation Manager. This feature enables starting and stopping a simulation multiple times without restarting the entire simulator.

4.5.2 Ethereum Manager

Ethereum Manager controls the workflow inside the Ethereum environment as it tracks the entire state of the blockchain. It's the component that helps the peers to reach consensus when adding new blocks to the blockchain. At the time, when it is initialized, it itself creates an n number (n is taken from the configuration previously passed from the Simulation Manager) of Ethereum nodes and a single instance of Transaction Manager. Furthermore, based on the configuration, it can create a periodical timer for sending messages to the Transaction Manager for creating external transactions. The idea behind the external transactions is to enable simulation of user transactions, which would significantly increase the total number of transactions in the network and will consequently increase the number of blocks. Figure 4.5 illustrates the process of creating

external transactions and sending them to Ethereum nodes.
When a simulation is being stopped, Ethereum Manager receives a message from Simulation Manager and destroys all Ethereum Nodes, the Transaction Manager an the periodical timer for external transactions, in case it is still running.

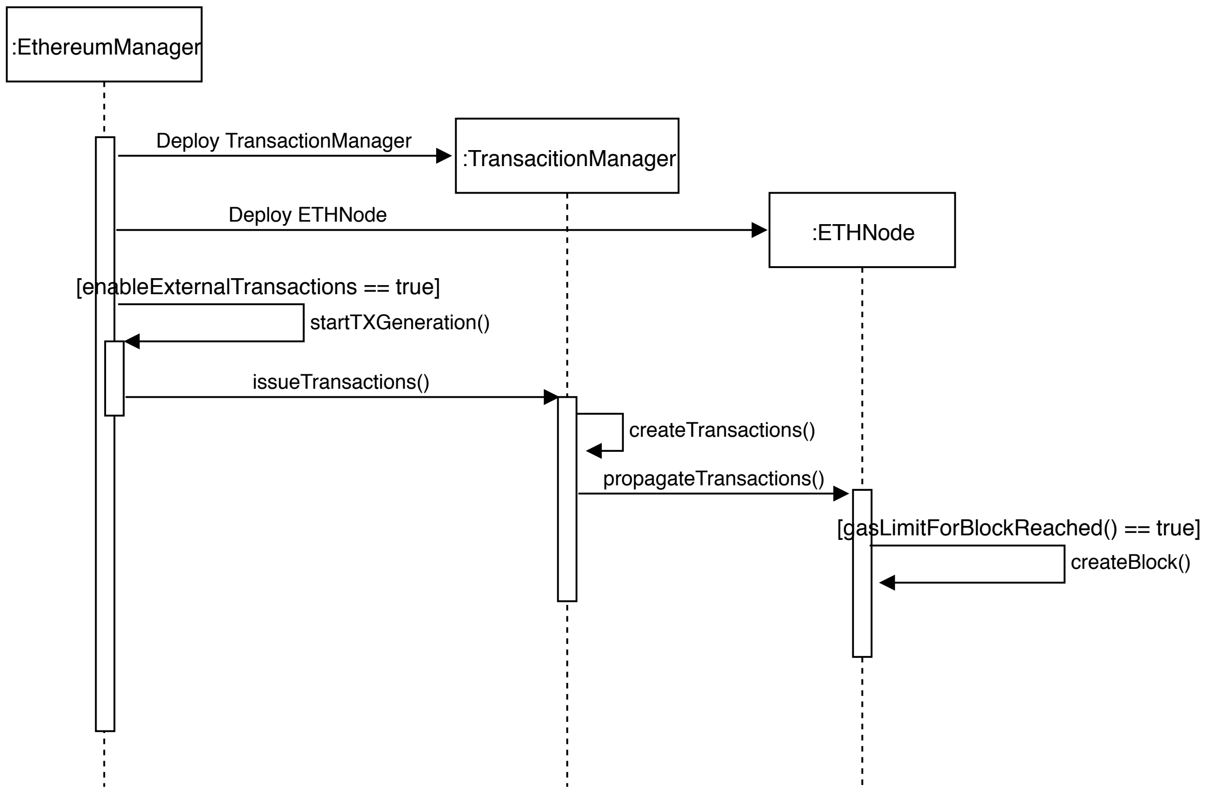


Figure 4.5: A workflow of creating external transactions and propagating them to Ethereum Node. A block created by a node contains both external transactions and contract transactions. Contract transactions are created and sent by Plasma participants (Plasma Clients and Operator)

4.5.3 Peer Discovery Node

Peer Discovery Node is a verticle that updates peer's neighbours tables at a regular time interval, previously set in the configuration and passed by the Simulation Manager. It further considers a configuration parameter that determines how many neighbours each peer should have. The set of neighbours is determined in random way, hence there is no particular protocol that we employ. This is a decision that has been taken based on the objectives we have, which are not related to simulating particular neighbours discovery protocols. We leave the opportunity for future users of our simulator to implement a protocol they are willing to analyse. This can be easily achieved by extending or replacing the current randomization principle we utilize. In section 4.6, we will demonstrate the method used for determining neighbours for each peer.

4.5.4 Ethereum Node

Ethereum Node takes care of mining blocks and adding them to the Ethereum blockchain. It collects transactions that are propagated from other peers of the network and in case these transactions are valid, they are being added to a list of transactions. A block can be created when the list of transactions contains enough transactions, which gasLimit sum reaches a gasLimit for a block. Block gasLimit is a configuration parameter that the Ethereum Node receives from the Ethereum Manager. If the node is allowed to mine the Block, it adds it to its blockchain, executes every transaction in the block and finally propagates it to its neighbours.

4.5.5 Transaction Manager

Transaction Manager is a component that serves only for generating of external transactions. Depending on a configuration parameter set by the user of the simulator, it generates n number of transactions for n number of accounts when it receives a message for doing so from the EthereumManager. Since the Transaction Manager is part of the Ethereum network, it propagates the newly created transactions to its neighbours.

4.5.6 Plasma Contract

Currently, we have only one smart contract that our simulator employs. Its behaviour is close to the behaviour of Plasma MVP contract specification [54]. A Plasma contract takes care of asset deposit, block submission and asset withdraw.

In order to enforce the state of a Plasma chain, the contract maintains local list of blocks, which must by synchronized with the blocks in its child blockchain. When a contract receives a deposit transaction, it creates a block based on this transaction and adds it to its local list of blocks. Afterwards, it emits an event containing the details about the deposit block, which is being transmitted to the Operator of the Plasma chain.

When a *submitBlock* transaction is being sent to the contract, the contract receives the hash value of the block and adds it to the blockchain. Synchronization of order of deposit blocks and Plasma blocks is very important for the proper function of the Plasma contract. We came up with an algorithm that achieves that, which we will introduce in section 4.6.

Withdraw transactions are sent from a particular Plasma client that is willing to exit the Plasma chain and to unlock the assets on the Plasma chain. Our simulator can withdraw the assets of the Plasma clients at the end of each simulation.

4.5.7 Plasma Manager

Plasma Manager is the verticle that manages the workflow within Plasma blockchains. Based on the configuration provided by the user, Plasma Manager can initialize a single first-depth level Plasma blockchain with or without multiple second-depth level Plasma chains. For each Plasma chain, Plasma Manager deploys n number of Plasma clients with a balance of a certain amount of tokens and one Operator. These tokens are then used for creating transactions between the clients. Furthermore, the manager starts an interval timer based on which a message for creating a transaction is being sent to all Plasma clients of a certain Plasma chain. The number of intervals and their duration is again defined by the user. Figure 4.6 depicts the process of creating a transactions that includes a the actions performed by the Plasma Manager for a single Plasma Chain.

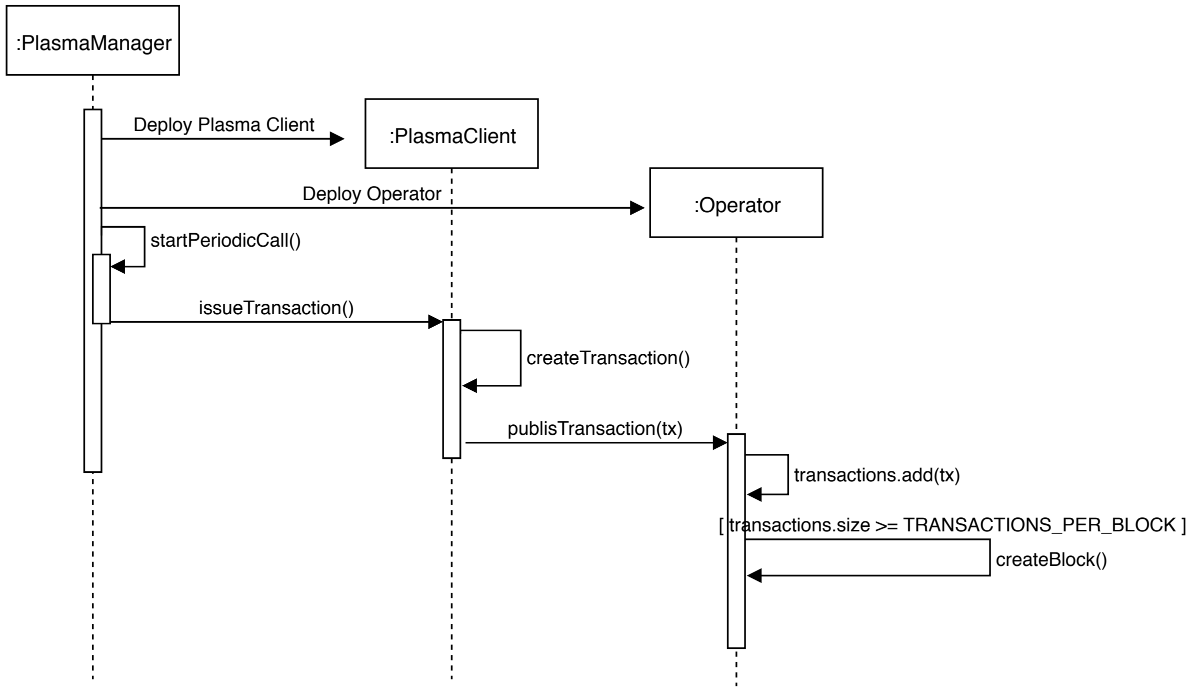


Figure 4.6: A workflow of creating a transaction and adding it to a block on a Plasma blockchain.

4.5.8 Plasma Client

The simulation of the behaviour of clients using a Plasma chain is a responsibility of the Plasma Client. Every Plasma Client receives a certain amount of tokens (defined by the user), which it uses to create valid transactions to other Plasma Clients in a Plasma Chain. Upon initialization, the Plasma Client sends a deposit transaction to the Plasma contract in order to join a Plasma Chain. Furthermore, each client maintains a list with all other clients' addresses in the chain. This list is used to randomly select a receiver for each transaction. At a particular interval, a client receives a message to issue a transaction. When creating a transaction, a Plasma Client first determines the receiver and then calculates an amount of tokens to send. The number of tokens depends on the current balance of the client.
Each Plasma Client receives every mined block on the Plasma Chain. This is necessary, as the clients must firstly monitor everything that happens on the blockchain and secondly compute how many tokens it possesses. In section 4.6 we will discuss in detail the process of monitoring all assets of a client.

4.5.9 Operator

Creating and adding blocks to the blockchain are tasks delegated to an Operator. An Operator is the only miner node in a Plasma Chain. It receives transactions from Plasma Clients and adds them to a list of transactions. When the size of this list reaches a certain threshold (configured again by the user), the operator creates a block and adds it to its blockchain. The block is then broadcasted to all Plasma clients and the operator creates a *submitBlock* transaction, which it sends to the Plasma Contract. Besides receiving transactions from clients, the Operator receives deposit blocks from the Plasma Contract. When a deposit block is received, it is being added to the blockchain and is again propagated to the Plasma Clients.

4.5.10 Clients Discovery Node (CDN)

To create a transaction each Plasma Client has to know the addresses of all other participating clients in the Plasma Chain. This collection of addresses is a task of the Clients Discovery Node.
When a Plasma client is deployed, it sends a message containing its address to the Client

Discovery Node. CDN collects all addresses of the clients and when all addresses are collected, the CDN broadcasts a message with all clients' addresses to all Plasma Clients.

4.5.11 Plasma nested blockchains

Our simulator allows the simulation of two levels nested Plasma blockchains. That is, when a a Plasma block of level 2 reports to a Plasma block of level 1, which reports its block to Ethereum. Figure 4.7 illustrates this process. The idea is, that an Operator of a second level blockchains creates a block and sends its hash value to the Operator of the parent (first level) blockchain. Upon receiving the hash value of the block of the child blockchain, the Operator creates a transaction for this block and adds it to its list of transactions. Later, when the Operator of the parent Plasma chain mines a block, it submits the the block hash value to the Plasma Contract. This enables the already discusses MapReduce approach and allows great scalability options.

4.6 Implementation

In this section we will take a closer look at more concrete implementation of the important workflows of our simulator.

4.6.1 Configuration

In section 4.5 we mentioned many times, that a certain behaviour of the simulator depends on the configuration the user sets. In this subsection we define all parameters as follows:

- *numberOfEthereumNodes* - number of Ethereum nodes (miners)
- *numberOfPlasmaClients* - number of Plasma clients that each Plasma blockchain contains
- *tokensPerClient* - number of tokens each Plasma client receives at the beginning of a simulation
- *transactionsPerPlasmaBlock* - number of transactions included in a single block on a Plasma blockchain
- *plasmaChildren* - number of Plasma second-level blockchains which report to a first-level Plasma blockchain (parent)

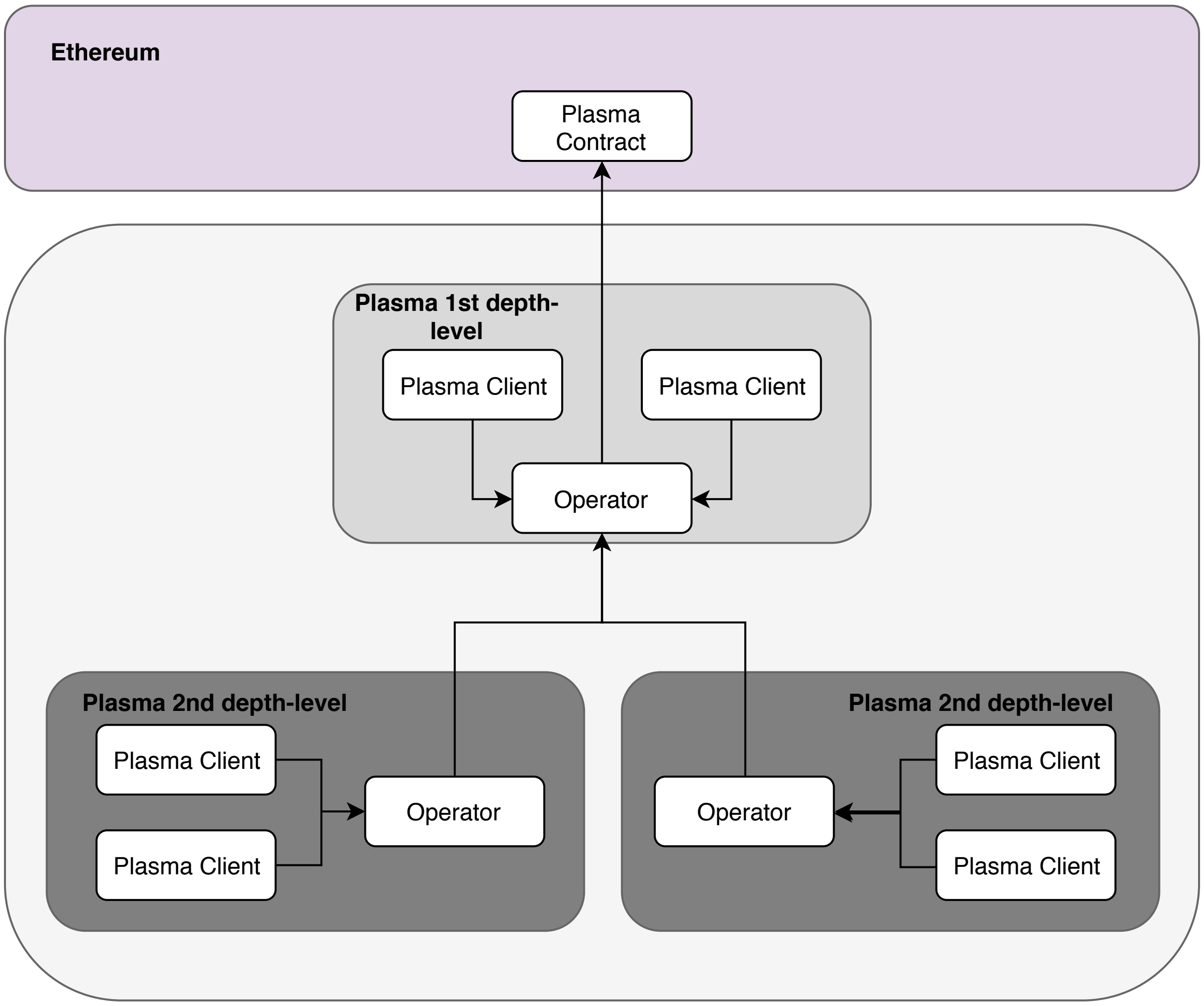


Figure 4.7: Plasma Nested Blockchains with two second-depth level blockchains. The number of blockchains on the second level is defined by the user.

- *numberOfNeighbours* - number of neighbours each node in the simulated P2P-Network is connected to
- *neighboursUpdateRate* - number of seconds after which neighbours of each peer are recomputed and updated
- *blockGasLimit* - gasLimit of an Ethereum block
- *plasmaTXGasLimit* - gasLimit of each transaction created from Plasma participants and submitted to Ethereum network
- *plasmaTXGasPrice* - gasPrice for each transaction created from Plasma participants and submitted to Ethereum network

- *enableExternalTransactions* - a flag to determine whether the simulator should generate externally owned transactions (user transactions) within Ethereum network
- *externalTxGasLimit* - gasLimit of each user transaction
- *externalTxGasPrice* - gasPrice for each user transaction
- *externalTransactionGenerationRate* - number of seconds after which transactions are automatically generated
- *numberOfEthereumExternalAccounts* - number of accounts for which user transactions will be generated
- *amountPerEthereumExternalAccount* - number of tokens each externally owner account receives (used for creation of user transactions)
- *transactionGenerationRate* - number of seconds after which transactions are automatically generated within Plasma blockchain (this rate is true for all Plasma blockchains)
- *numberOfTransactionGenerationIntervals* - number of intervals at which transactions are generated. After completion of the last interval, the simulation ends
- *plasmaBlockInterval* - interval that determines how many deposit blocks can a Plasma contract create between two blocks of transactions generated by an Operator

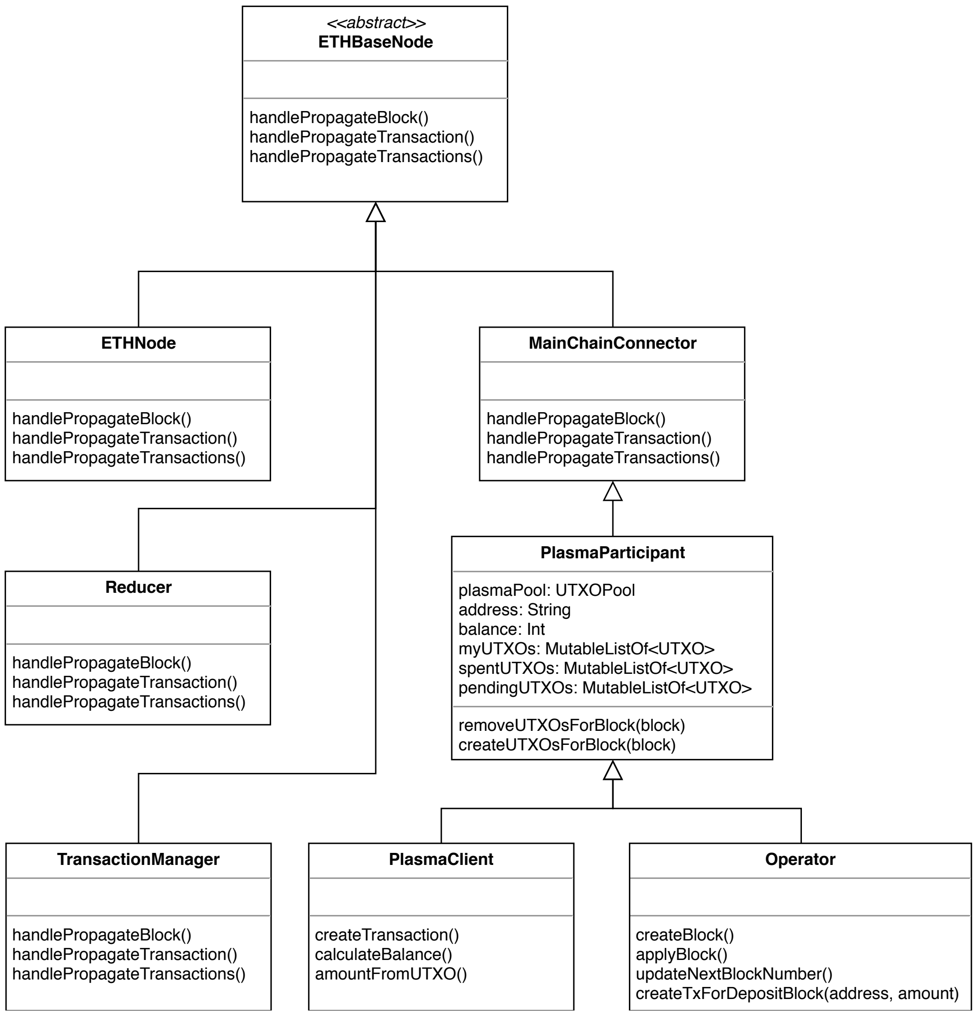
The configuration can be altered prior to starting the simulation. The individual parameters can be defined via our UI application.

4.6.2 Simulation Initialization

Simulation can be started only via our UI. When a start message from the web application is sent to the Simulation Manager, it checks whether a simulation is running and if that is not the case it can start the simulation. First of all, the manager fetches the last version of the Configuration, so that it can deploy the other verticles with that configuration. It also defines the *startTime* in order to measure the elapsed time of the simulation. On deployment of other verticles, the Simulation Manager, tracks their deployment Ids, so that they can be destroyed when the simulation stops.

4.6.3 Peers in Ethereum Network

Even though Plasma blockchain is mainly managed outside of the Ethereum environment, it is important to discuss all different peers in Ethereum. Besides Ethereum Nodes, the Plasma Clients and the Operator also participate in the network, since they broadcast transactions such as deposit transactions, submitBlock transaction and withdraw transaction. This arises the necessity of basic functionality that all nodes in the network need to share. This basic functionality is defined in an object called *ETHBaseNode* and comprises the handling of propagation events such as *propagateTransaction*, *propagateBlock*, *propagateTransactions* and *setNewPeers*. All peers extend the functionality of the ETHBaseNode as illustrated on Figure 4.8.



<<abstract>>
ETHBaseNode
handlePropagateBlock()
handlePropagateTransaction()
handlePropagateTransactions()
ETHNode
handlePropagateBlock()
handlePropagateTransaction()
handlePropagateTransactions()
MainChainConnector
handlePropagateBlock()
handlePropagateTransaction()
handlePropagateTransactions()
Reducer
handlePropagateBlock()
handlePropagateTransaction()
handlePropagateTransactions()
PlasmaParticipant
plasmaPool: UTXOPool
address: String
balance: Int
myUTXOs: MutableListOf<UTXO>
spentUTXOs: MutableListOf<UTXO>
pendingUTXOs: MutableListOf<UTXO>
removeUTXOsForBlock(block)
createUTXOsForBlock(block)
TransactionManager
handlePropagateBlock()
handlePropagateTransaction()
handlePropagateTransactions()
PlasmaClient
createTransaction()
calculateBalance()
amountFromUTXO()
Operator
createBlock()
applyBlock()
updateNextBlockNumber()
createTxForDepositBlock(address, amount)

Figure 4.8: Class Diagram of Peers in the P2P Network

4.6.4 UTXO Pool

Each Plasma chain implements an UTXO record keeping model. For keeping track on all UTXOs available on the system, we introduce UTXOPool. UTXOPool is important for the correctness of each Plasma chain, as it contains all UTXOs that can be used for generating transactions with appropriate amount of tokens. This means that a Plasma client cannot send more tokens to other clients than it currently possesses. The UTXOPool is being updated whenever a valid block is added to the blockchain (Listing 4.1). We encourage the reader to take a look at the Appendix where the source code of the UTXOPool is presented.

```
fun removeUTXOsForBlock(block: PlasmaBlock) {
    for(tx in block.transactions) {
      for(input in tx.inputs) {
        val utxoToRemove = UTXO(input.blockNum, input.txIndex,
 ↪  input.outputIndex)
        if(myUTXOs.contains(utxoToRemove)) {
          myUTXOs.remove(utxoToRemove)
        }
        plasmaPool.removeUTXO(utxoToRemove)
      }
    }
  }

  fun createUTXOsForBlock(block: PlasmaBlock) {
    for((txIndex, tx) in block.transactions.withIndex()) {
      for((outputIndex, output) in tx.outputs.withIndex()) {
        val newUTXO = UTXO(block.number, txIndex, outputIndex)
        if(address == output.address && !myUTXOs.contains(newUTXO)){
          myUTXOs.add(newUTXO)
        }

        plasmaPool.addUTXO(newUTXO, output)
      }
    }
```

Listing 4.1: Updating UTXOPool for a newly added block to the Plasma chain

4.6.5 Plasma Participants

All nodes working on a Plasma blockchain are called Plasma participants. They all share some components that help them maintain a healthy and correct blockchain. One of these components is a PlasmaChain. PlasmaChain supports participants on validating and adding new blocks to their local blockchain and on synchronization between a Plasma Contract and the Plasma chain (further explained in subsection 4.6.12). Furthermore, each individual participant has an instance of an UTXOPool, which helps the nodes to maintain the correctness of the Plasma chain. Figure 4.9 visualizes the dependencies between the above described objects.

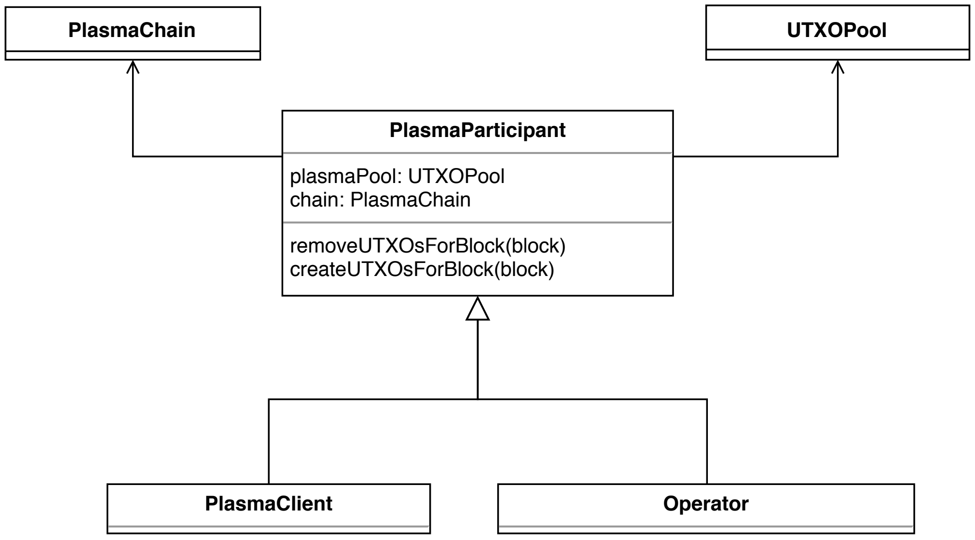


Figure 4.9: Dependencies between Plasma Participants, UTXOPool and PlasmaChain

4.6.6 Asset Deposit

Upon deployment, Plasma Clients create a deposit transaction with all available balance (Listing 4.2) and propagate it to the Ethereum Network. Ethereum Nodes receive the transaction and add it to their queue of transactions. When the deposit transaction is added to a block, it is being executed, which results in calling the deposit method on the Plasma Contract. For each deposit transaction, the Plasma Contract creates a block (Listing 4.3), adds it to its chain of blocks and emits the block to the Operator. When Operator receives the deposit block, it adds it to its blockchain and creates an UTXO for the single transaction this block contains. The block is propagated to the Plasma clients, which themselves validate the block and update their UTXOPool. For Operator's source refer to the Appendix.

```
fun deposit(address: String, amount: Int, chainAddress: String,
↪ parentPlasmaAddress: String?) {
    var data = mutableMapOf<String, String>()
    data.put("type", "plasma")
    data.put("method", "deposit")
    data.put("address", address)
    data.put("amount", amount.toString())
    data.put("chainAddress", chainAddress)

    if(parentPlasmaAddress != null)
      data.put("parentPlasmaAddress", parentPlasmaAddress)

    val tx: ETHTransaction =
↪ createTransactionToPlasmaContract(address, data)
    sendTransaction(tx)
  }
```

Listing 4.2: Deposit method for a Plasma Participant

```
fun deposit(address: String, amount: Int, chainAddress: String) :
↪ JsonObject{
    state.put(address, Account(0, address, amount))
    val rootHash =
↪ HashUtils.transform(HashUtils.hash(address.toByteArray() +
↪ amount.toByte()))
    childBlocks.put(depositBlockNumber, PlasmaBlock(rootHash))
    return JsonObject()
      .put("address", address)
      .put("amount", amount)
      .put("blockNum", depositBlockNumber++)
      .put("rootHash", rootHash)
      .put("chainAddress", chainAddress)
  }
```

Listing 4.3: Deposit method on Plasma Contract

4.6.7 Plasma Transaction Generation

When a Plasma Client possesses at least one UTXO in the UTXOPool, it is able to generate transaction. This happens when a client receives a message from Plasma Manager to do so. As explained in 4.5, Plasma Manager has a timer that runs *numberOfTransactionGenerationIntervals* times every *transactionGenerationRate* seconds. When it ticks the manager broadcasts an *issueTransaction* message to all Plasma clients of the corresponding Plasma Chain.
Consequent to receiving the message, the client creates a transaction and propagates it to the other Plasma Participants. It is important, that all clients see all transactions and all blocks, since it is their duty to monitor the entire chain for proper state transitions.

4.6.8 Plasma Block Creation

Operator receives all transactions Plasma Clients receive. It adds them to a FIFO queue and when the size of the queue reaches *transactionsPerPlasmaBlock*, it starts creating a block.
Each Plasma block must contain a Merkle root. To generate a Merkle root, the Operator has to first create a MerkleTree. This task is delegated to a MerkleTreeBuilder which builds the tree (Listing 4.4) and delivers the Merkle root hash. This hash value is further used to generate a *submitBlock* transaction (Listing 4.5). This transaction is propagated to the Ethereum network, where it will be included in a block. Once that happens, the transaction will be executed by the Plasma Contract and the new block will be added to its blockchain (Listing 4.6). In this way the blocks on the Plasma chain are synchronized with the local blockchain of the Plasma Contract

```
fun buildTree(trees: MutableList<MerkleTree>): MerkleTree {
    if(trees.size == 1) return trees[0]
    var parents = mutableListOf<MerkleTree>()

    for(i in 0 until trees.size step 2) {
      val parentTree = MerkleTree()
      val right = if(trees.size > i + 1) trees[i+1] else null
      parentTree.add(trees[i], right)
      parents.add(parentTree)
    }
    return buildTree(parents)
  }
```

Listing 4.4: MerkleTreeBuilder method for creating a Merkle tree.

```
fun submitBlock(from: String, rootHash: ByteArray, timestamp: Long) {
    val data = mutableMapOf<String, String>()
    data.put("type", "plasma")
    data.put("rootHash", HashUtils.transform(rootHash))
    data.put("method", "submitBlock")
    data.put("timestamp", timestamp.toString())
    val tx: ETHTransaction = createTransactionToPlasmaContract(from,
 ↪  data)
    sendTransaction(tx)
  }
```

Listing 4.5: New blocks on Plasma blockchain are submitted as an Ethereum transaction to the Ethereum network. This responsibility is undertaken by an Operator

```
fun submitBlock(rootHash: String, timestamp: Long) {
    childBlocks.put(plasmaBlockNumber, PlasmaBlock(rootHash,
 ↪  timestamp))
    updateBlockNumbers()
  }
```

Listing 4.6: Plasma Contract's method for submitting blocks, created in a Plasma chain.

4.6.9 Asset withdrawing

In the current setup clients are withdrawing their tokens at the end of the simulation. To do that, clients generate a withdraw transaction and send it to the Ethereum network. When it gets included in a block, the transaction will be executed and the smart contract will unlock the tokens of the client on Ethereum. This behaviour can be improved by adding dynamic withdraws happening during a simulation.

4.6.10 Plasma Nested Chains

The implementation of the nested chains is coordinated between the Plasma Manager and the created operators of the Plasma chains. The configuration parameter *plasmaChildren* determines how many second-depth level chains will be created. Based on this number, the Plasma Manager deploys $plasmaChildren + 1$ Plasma chains, each equipped with an Operator and with *numberOfPlasmaClients* Plasma clients. Additionally, every blockchain has its own address, that ensures the coordination. All second-level blockchains are independent from each other, but they share a common attribute - *parentChainAddress*, which corresponds to the chainAddress of the first-level chain. So, when a second-level blockchains creates a block, it sends it to the *parentChainAddress*. The parent Operator creates a transaction for each child block and adds it to its queue of transactions. On the other hand, the Operator of the parent chain maintains a list of all children addresses. This is needed, as all deposit blocks coming from Ethereum are propagated only to this Operator. When a deposit block is received, the Operator has to send it to the corresponding child chain Operator. This approach ensures the hierarchy of the nested chains and the proper propagation of messages.

4.6.11 Ethereum-Plasma Communication

We have already explained that each Plasma Participant is a peer in the Ethereum network. Nonetheless, for keeping clean design we need to separate the responsibilities, which a Plasma Participant has inside a Plasma Chain and outside of it. To The inside actions are performed by Plasma Clients and Operator, while the outside actions (Ethereum actions) are performed by MainChainConnector. The responsibilities of the MainChainConnector encompass handling all events of the Ethereum network together with creating Ethereum transactions for asset deposit,block submission and asset withdraw. Normally, the MainChainConnector would be a service, that each Plasma Participant would user, but given our event-driven architecture, it becomes a verticle. We had to decide whether each Plasma Participant will comprise two verticles - one responsible for Plasma chain and the other for the Ethereum Chain or we will implement one verticle with two different responsibilities. Since we are dealing with great amount of computation, it is hard to imagine that we would sacrifice performance for a better design. This is the reason why a MainChainConnector and a PlasmaParticipant are the same verticle (enabled by multiple inheritance as it can be seen on Figure 4.8) but at the same time these two classes implement different behaviour. In case a Plasma contract changes, then a developer would need to change only the MainChainConnector.

4.6.12 Plasma Contract-Operator block synchronization

Plasma block synchronization between the Plasma contract and the Operator is quite relevant for ensuring security. Synchronization problem arises from the fact, that blocks are generated on both sides - the contract creates a block for every deposit block, while the Operator creates a block for clients' transactions. Our approach regarding this problem is based on block numbers.
We define *plasmaBlockInterval* as the number of deposit blocks that can occur between two blocks created by an Operator. Every block created by an Operator will be a multiple of plasmaBlockInterval(e.g. plasmaBlockInterval = 100, then block numbers will be: 100, 200, 300, 400, etc.). On the other hand, every deposit block will have a number between two plasmaBlockIntervals (e.g. 1, 2, 3, 101, 102, 205, etc.).
When an Operator receives a deposit block, it adds it to the chain and increments the next expected deposit block number. The same applies for the Plasma contract. Whenever a Plasma contract receives a block from an Operator, it adds to the current *plasmaBlockNumber plasmaBlockInterval.* This process is difficult to describe without an example. Figure 4.10 describes an example of our approach.

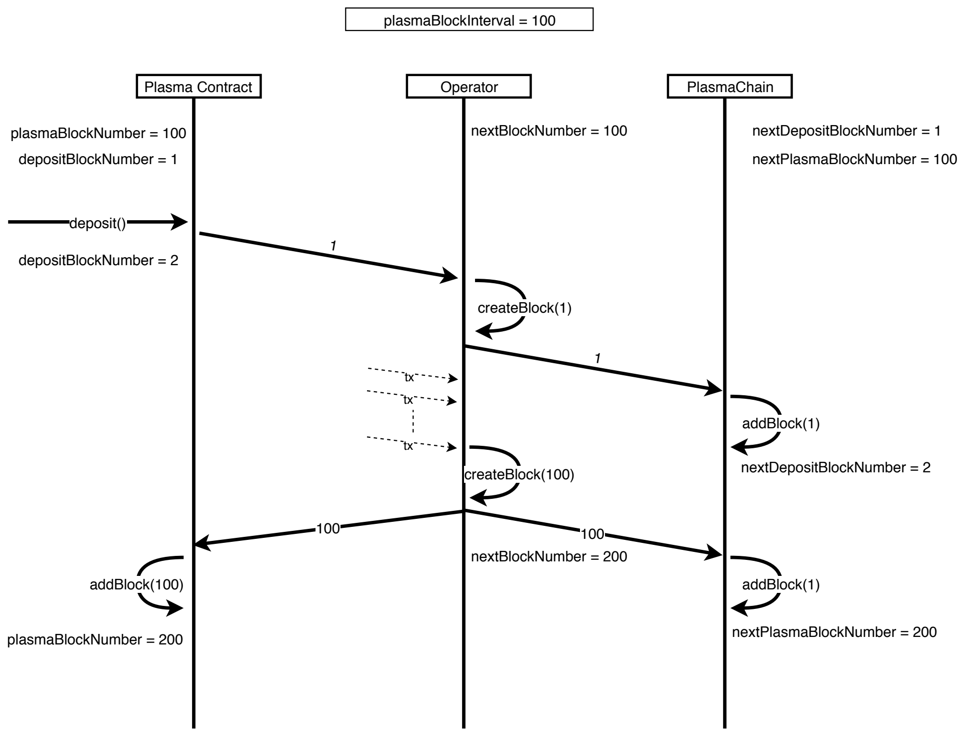


Figure 4.10: Block synchronization between Plasma contract and Operator (supported by a Plasma chain). The Plasma contract receives the first deposit transaction from some client. It creates a block with number *depositBlockNumber* and increments it for the next deposit transaction. The Operator's chain tracks the received deposit blocks as well with *nextDepositBlockNumber*. When an Operator receives the deposit block, the PlasmaChain validates the block via *nextDepositBlockNumber* and increments it after the block is added to the blockchain. Later on, an Operator creates a block with number *nextBlockNumber*. It propagates the block to the smart contract, which expects to see a block with number *plasmaBlockNumber*. If that is the correct block, it adds it to the blockchain and increments it by *plasmaBlockInterval*. The same applies for the PlasmaChain which increments *nextPlasmaBlockNumber*.

4.6.13 Ethereum Block creation

Transactions propagated in the Ethereum Blockchain are collected by all peers, but only an Ethereum Node can create blocks. Upon receiving a transaction, the Ethereum Node validates it based on its nonce (transactions must be received in order based on the nonce) and adds it to its FIFO queue. When the sum of all transactions' gasLimit reaches *blockGasLimit*, the node creates a block.

4.6.14 Ethereum Block mining

To prevent resource intensive computations for a Proof-Of-Work puzzle, we introduce a probability approach for adding new blocks to the Ethereum blockchain. When a node creates a block, it has 30% chance of mining this block. If many nodes create the same block, then the Ethereum Manager decides which of these nodes is allowed to add the block based on a First-come, First-Served principle. This is not a perfect way to reach a consensus in a decentralized system, as it relies on a central authority, this is currently sufficient for the goals of this thesis. Improving the consensus algorithm is considered as future task.

4.6.15 Node discovery

Upon deployment each Ethereum network peer publishes its address to the Peer Discovery Node. The Peer Discovery Node collects all addresses and determines the neighbours each peer will communicate with as it updates the neighbours tables every *neighboursUpdateRate*. The number of neighbours is defined by the configuration parameter *numberOfNeighbours*. Listing 4.7 shows the randomization algorithm that our simulator currently utilizes.

```
fun selectNewPeersFor(address: String): JsonArray {
    var allNodes = nodeAddresses.toMutableList()
    allNodes.remove(address)
    var neighbours = JsonArray()
    allNodes.shuffled().take(numberOfNeighbours).forEach { peer ->
      neighbours.add(peer)
    }
    return neighbours
  }
```

Listing 4.7: Randomization Algorithm for determining neighbours of a peer in the Network.

4.6.16 Technology Choice

As already introduced in section 4.3, our approach is based on Vert.x. Vert.x is a framework for building reactive, event-driven and non-blocking applications that can handle a great amount of concurrency using small number of kernel threads. Vert.x is polyglot, which means that developers can develop concurrent applications in multiple languages such as Java, JavaScript, Groovy, Ruby, Ceylon, Scala and Kotlin [6]. This characteristic helps us meet one of our requirements, namely giving freedom to developers to choose a language for extending our simulation system in a programming language they feel comfortable with.

For our project we decided to use Kotlin [56]. The decision was based on the advantages that Kotlin offers compared to Java. It solves some of the problems that Java faces such as null-references, raw types and checked exceptions. Furthermore, Kotlin reduces significantly boilerplate code in Java (Kotlin was developed with the idea to extend and improve Java) so that developers can write their code faster and in a more safe and reliable way. Despite that, Kotlin is fully compatible with Java, and libraries written in Java can be easily imported into Kotlin code. Moreover, Kotlin can deal with multi-threading in multiple ways, which was important to us due to the concurrent system we built.

To build the simulator fast and without manually managing and configuring all dependencies, we used the Gradle build system [57].

4.6.17 Web application

To display the results of a simulation, we built a web application in TypeScript [58] using Angular framework [59] in a Redux architectural style [31]. Our main purpose was to built a reactive web application that shows the results of a simulation in real-time and that deals with the internal state in a reliable and predictive way. To prove the distributed nature of the event bus of Vert.x as well as its polyglot abilities, we shared the event bus of the Plasma simulator (written in Kotlin) with the web application (written in TypeScript). In this way, the application listens for events that happen during simulation and displays them on the screen.

Due to the great number of events happening during a simulation, it was important to keep the state of the web application clean and consistent, and to manage it in such a way that unpredicted state changes are highly unlikely. This requirement was successfully met by implementing the Redux architecture style using NGRX [60]. NGRX provides reactive state management for Angular applications and consists of the following key concepts:

- Actions - an action describes special events that are dispatched from components and services.
- Reducer - state changes are handled by pure functions called reducers, that take current state, apply a dispatched action and compute a new state.
- Selectors - pure functions used to select and derive particular pieces of the state.
- Store - observable of state and observer of actions.

- Effects - side effect models for Store to reduce the state based on external interactions such as network requests, web-socket messages and time-based events.

The NGRX top-level architecture is illustrated on Figure 4.11.

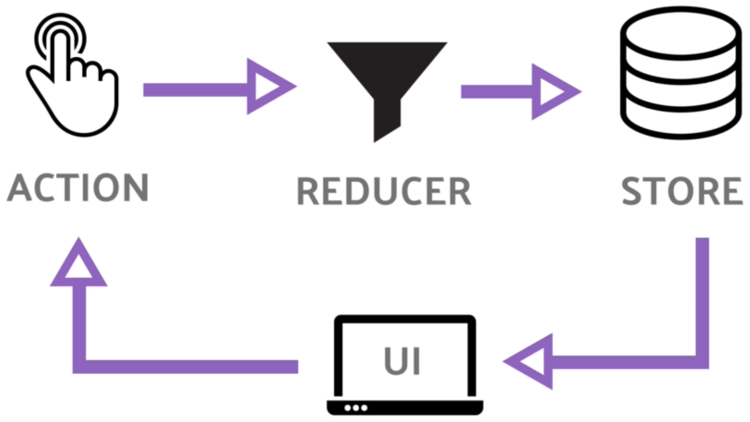


Figure 4.11: NGRX Top-Level Design

Chapter 5

Evaluation

In this chapter we will evaluate the simulator according to 7 criteria: Resource Utilization, Correctness, Flexibility, Extensibility, Scalability, Plasma Framework Compliance and Powerful Visuals as they were defined in Chapter 1. All tests have been conducted on MacBook Pro 2015, Intel Core i5-5257U @ 2.7GHz (3MB shared Level 3 cache), 8GB 1867 MHz DD3.

5.1 Correctness

At the time of writing this thesis there is no application running a productive Plasma application. Due to this fact reliable reference results are hard to find which is why we are going to validate the correctness of the simulator based on empirical analysis and numerical proofs. We would like to validate the consistency between input parameters, expected behaviour and the simulated output.

5.1.1 Expected and simulated total balance of all Plasma clients within a single Plasma blockchain

Each Plasma client joins a Plasma blockchain with a specified initial amount of tokens. During a simulation, clients create a specified number of transactions to other clients in the same Plasma blockchain. At the beginning of the simulation we have:

$$totalNumberOfTokens = tokensPerClient * numberOfPlasmaClients$$

	Run 1	Run 2	Run 3	Run 4	Run 5
Client 1	86	98	85	161	58
Client 2	58	162	122	184	134
Client 3	81	117	137	196	165
Client 4	100	66	62	46	54
Client 5	128	82	72	91	116
Client 6	148	151	146	68	126
Client 7	78	99	124	78	88
Client 8	80	64	104	36	52
Client 9	107	107	120	73	160
Client 10	134	54	28	67	47
Total simulated	1000	1000	1000	1000	1000
Expected	1000	1000	1000	1000	1000

Table 5.1: Evaluation of total balance of all Plasma clients within a single Plasma blockchain

At the end of the simulation, the sum of all clients' balances should have the same value as $totalNumberOfTokens$. This is due to the fact, that our simulator makes use of an UTXO based transaction model and that new tokens are not generated within a Plasma chain.
For this experiment, we ran a simulation for 10 clients ($numberOfPlasmaClients$) within a single Plasma blockchain, each receiving 100 tokens ($tokensPerClient$) for a transaction generation interval of 100.
Table 5.1 shows the results.

5.1.2 Expected and simulated number of created blocks within a single Plasma blockchain

Recall that, a Plasma blockchain in our simulation employs a Proof of Authority consensus algorithm. Blocks are generated for specified number of transactions at a particular transaction generation interval. The blockchain length that a simulation emits depends

Tokens per Client	Run 1	Run 2	Run 3	Run 4	Run 5	Simulated	Expected
10	105	100	105	104	106	104	110
20	108	109	108	109	109	108.6	110
50	110	108	109	109	110	109.2	110
100	110	109	110	109	110	109.6	110

Table 5.2: Evaluation of Plasma blockchain length

also on gasLimit within an Ethereum block and gasLimit for a transaction as well as number of transaction per Plasma block. In case block gasLimit is less then the sum of the gasLimit of all deposit transactions, then some of the deposit transactions will be mined in a later block, which prevents some Plasma clients from creating transactions for the first couple of intervals. This feature could be quite useful for estimation and optimization of gas usage.
To avoid pending times for Plasma clients, we ran a simulation with an Ethereum block gasLimit that corresponds to the sum of all Plasma clients' deposit transactions gasLimits. That is a simulation with the following parameters:

- **Number of clients** : 10
- **Gas limit for transaction**: 30
- **Gas limit for an Ethereum block**: 300
- **Number of transactions per Plasma block**: 5
- **Transaction generation intervals**: 50

Based on these parameters we would expect to see 10 * 50 = 500 transactions distributed into 100 blocks (500 transactions / 5 transactions per block). Additionally, the Plasma blockchain must contain another 10 blocks that correspond to the deposit blocks coming from ethereum. In total, our expected Plasma blockchain length is 110. This number could vary, as at some time of the simulation, it could happen, that a client has no utxos, hence cannot generate a valid transaction. The simulation results are summarized in Table 5.2.

Tokens per Client	Run 1	Run 2	Run 3	Run 4	Run 5	Simulated	Expected
10	485	460	485	480	490	480	510
20	500	505	500	505	505	503	510
50	510	500	505	505	510	506	510
100	510	505	510	505	510	508	510

Table 5.3: Evaluation of total number of transaction in a single Plasma blockchain

5.1.3 Expected and simulated number of transactions in single Plasma blockchain

Using the previous configuration, we can easily determine the expected total number of transactions that a single Plasma blockchain contains. That is, the sum of the number of deposit blocks (each deposit block contains one transaction) and the number of transactions created by the nodes. Table 5.3 summarizes the results.

5.1.4 Expected and simulated number of deposit blocks received by an operator in a single Plasma blockchain

When a client is willing to join the blockchain, it sends a deposit transaction to the Plasma contract. For each deposit transaction the contract generates a block and adds it to its blockchain. When a transaction is mined in ethereum, the operator of the corresponding Plasma blockchain receives the deposit block.
Expected behaviour of the simulator would be that an operator receives all deposit blocks, that clients generate. That is, each of the Plasma clients, we define, send one deposit transaction every single simulation. Hence, number of deposit blocks should always correspond to the number of clients. Table 5.4 illustrates once again consistent results.

5.1.5 Total number of blocks in nested Plasma blockchains

To evaluate number of blocks that are produced in nested Plasma blockchains, we will simulate a single Plasma first-level blockchain with two Plasma second-level blockchain (children of the first-level chain).
The parameters that we set and are important for the evaluation are the following:

Number of clients	Run 1	Run 2	Run 3	Run 4	Run 5	Simulated	Expected
10	10	10	10	10	10	10	10
30	30	30	30	30	30	30	30
100	100	100	100	100	100	100	100

Table 5.4: Evaluation of number of deposit blocks

- **Number of clients** : 10
- **Gas limit for transaction**: 30
- **Gas limit for an Ethereum block**: 900
- **Number of transactions per Plasma block**: 5
- **Transaction generation intervals**: 50
- **Transaction generation rate**: 5
- **Second level (nested) blockchains**: 2

Based on this configuration, we can calculate the expected values as follows:

- Second-level (nested) Plasma blockchains - each of the nested chains has 10 clients, each creating a transaction every 5 seconds for 50 intervals. In total that makes $1 * 10 * 50 = 500$ transactions. These transactions split into blocks of 5 transactions is $500/5 = 100$. For each client, the operator of a second-level chain will receive a deposit block from its parent main Plasma chain, which is $1*10 = 10$ blocks. In total, we expect that 110 blocks will be mined on each second level Plasma blockchain.
- First-level (main) Plasma chain - analogously to the second level blockchains, the internal actors of the main Plasma chain should produce 110 blocks. Nonetheless, this is not the final number of blocks the first-level chain will produce, as second level chains report their blocks to a Plasma main chain. Hence, each of the blocks in a second-level blockchain correspond to one transaction in a main Plasma chain. Giving that, we simulate two second-level blockchain each of which is expected to yield 110 blocks, then an additional $2 * 110 = 220$ transactions will be submitted to the first-level Plasma blockchain. These transactions increase the number of blocks by $220/5 = 44$ blocks. Thus, in total we expect that $110 + 44 = 154$ blocks will be mined on the first-level Plasma blockchain.

	Run 1	Run 2	Run 3	Run 4	Run 5	Simulated	Expected
First second-level Plasma blockchain	154	153	153	154	153	153.4	154
Second second-level Plasma blockchain	110	109	110	110	110	109.8	110
First-level Plasma blockchain	110	109	109	110	110	109.6	110

Table 5.5: Evaluation of number of deposit blocks

It is important to note, that the simulations that we conducted were set up under the condition that all deposit blocks were mined in one Ethereum block. In reality this will hardly happen. Nevertheless, such ”perfect” condition demonstrates the correctness of the simulator regarding nested blockchains. If deposit blocks are mined in multiple blocks on the Ethereum chain then it will be extremely difficult to evaluate such nested chains' behaviour, as some accounts will start sending transactions in a Plasma chain significantly later than others. This behaviour that our simulator shows, can be at the same time quite helpful when analyzing the interactions between Ethereum and plasma. The results of this evaluation can be taken from Table 5.5.

5.1.6 Ethereum blockchain

Even though it was not a purpose of the simulator to simulate fully fledged Ethereum blockchain, we added only such functionality that is of importance for analyzing Plasma chains.
In this subsection we evaluate block generation.

Block generation

Blocks in Ethereum are mined and later added when a predefined gasLimit is reached. Adding of blocks to the Ethereum blockchain is essential for Plasma chains as this is the only way Ethereum clients can become part of plasma, thus Plasma clients, and also to withdraw their tokens from plasma.
To evaluate blocks generation in Ethereum we consider two cases

Run 1	Run 2	Run 3	Run 4	Run 5	Simulated	Expected
5	6	6	5	6	5.6	6

Table 5.6: Evaluation of block generation in Ethereum without user transactions

Block generation without user transactions

Block generation without user transactions is the ability of the simulator to function without generating transactions within Ethereum, excluding Plasma. For this case we define the following parameters:

- **Number of clients** : 30
- **Gas limit for transaction**: 10
- **Gas limit for an Ethereum block**: 300
- **Number of transactions per Plasma block**: 10
- **Transaction generation intervals**: 50

These parameters allow us to define expected number of blocks after a single simulation. To calculate the expected number of blocks, we first need to find the number of transactions that are propagated to Ethereum. The number of transactions is the sum of all *deposit* transactions added to the total number of *submitBlock* transactions. Deposit transactions are the same amount as the number of clients, thus 30. Transactions for submitting blocks is the number of blocks created in a Plasma chain. The number of blocks a Plasma chain should generate is $50 * 30/10 = 150$. Hence, the total number of transactions is $150+30 = 180$. Further, each transaction submitted to the main chain contains a gasLimit of 10, which helps us to calculate that the total number of gas is $180 * 10 = 1800$. The block gasLimit we defined is 300. Based all these simple calculations, we can define the expected number of blocks in Ethereum as $1800/300 = 6$. The results of the simulations we ran are shown on Table 5.6.

Block generation with user transactions

In addition to the previous configuration we enabled the generation of user transactions inside Ethereum, which will increase the number of blocks. The additional parameters are:

Run 1	Run 2	Run 3	Run 4	Run 5	Simulated	Expected
39	39	39	39	39	39	40

Table 5.7: Evaluation of block generation in Ethereum with user transactions

- **Gas limit for user transaction**: 10
- **Number of Ethereum user accounts**: 20

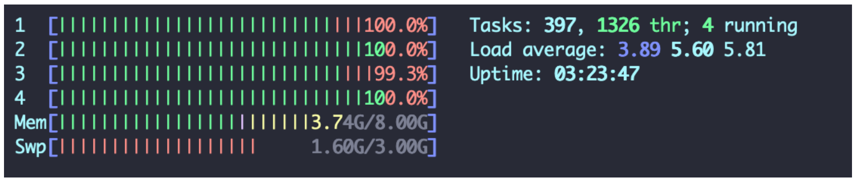
As we preserve the transaction generation interval of 50, we can calculate the expected additional number of transactions: $10 * 20 * 50 = 10000$ transactions, which leads to $10000/300 = 33.33 \approx 34$ blocks. The total number of blocks per simulation is then $6 + 34 = 40$. Table 5.7 shows again consistent results.

5.1.7 Conclusion

Despite many complex interactions within each simulation, the simulator demonstrates consistent results regarding its correctness when it is ran with appropriate configuration. By appropriate configuration we mean tuning the parameters in such a way, that Plasma and Ethereum blockchains interact in a proper way.
This makes the simulator a reliable tool for testing and adjusting Ethereum Plasma blockchains.

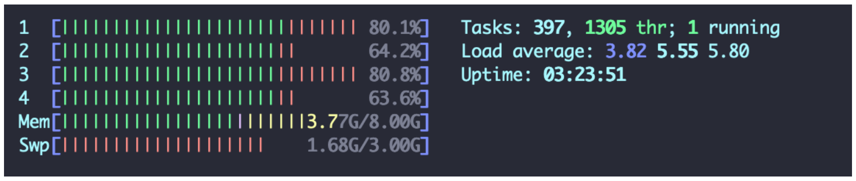
5.2 Resource Utilization

Resource Utilization refers to simulator's ability to use computing resources in an efficient manner. One of our initial requirements is to enable multi-threaded simulation on a single machine, whereas most threads are nodes in a simulated P2P network. Our empirical analysis showed that our simulator is able to accurately perform simulations with more than 200 nodes. In peak times of a simulation, when multiple clients are simultaneously generating transactions, CPU utilization reached its maximum (Figure 5.1). After that, we observed decline in CPU usage (Figure 5.2), since as explained in Chapter 4, transactions are generated at certain time intervals. The high resource utilization for complex simulations is an expected behaviour of the simulator due to the multi-threaded technology - vertx which was used for the implementation.



```
1  [||||||||||||||||||||||||||||||||100.0%]   Tasks: 397, 1326 thr; 4 running
2  [||||||||||||||||||||||||||||||||100.0%]   Load average: 3.89 5.60 5.81
3  [|||||||||||||||||||||||||||||||||99.3%]   Uptime: 03:23:47
4  [||||||||||||||||||||||||||||||||100.0%]
Mem[|||||||||||||||||||||||||||3.74G/8.00G]
Swp[||||||||||||||||||||      1.60G/3.00G]
```

Figure 5.1: CPU Utilization at parallel transaction generation and propagation.



```
1  [||||||||||||||||||||||||||||||    80.1%]   Tasks: 397, 1305 thr; 1 running
2  [||||||||||||||||||||||||          64.2%]   Load average: 3.82 5.55 5.80
3  [||||||||||||||||||||||||||||||    80.8%]   Uptime: 03:23:51
4  [||||||||||||||||||||||||          63.6%]
Mem[||||||||||||||||||||||||||3.77G/8.00G]
Swp[|||||||||||||||||||||     1.68G/3.00G]
```

Figure 5.2: CPU Utilization after transaction propagation.

5.3 Scalability

Scalability of the Plasma simulator relates to its ability to handle growing amount of work. To evaluate how the simulator behaves, when number of Plasma clients, Ethereum nodes and blocks increase, we will conduct 3 experiments.

5.3.1 Varying Plasma clients

We ran a simulation with 5 transactions per block on a Plasma chain, 30 Ethereum nodes and varying number of Plasma clients. Figure 5.3 shows that the number of Plasma blocks scales linearly, when the number of clients increases linearly.

5.3.2 Varying number of Ethereum nodes

For this experiment we linearly increased the number of Ethereum nodes and measured the average propagation time of blocks. By the number of nodes, we mean the sum of the dedicated Ethereum miners and the Plasma clients. Note that, the propagation time in our design depends only on the total number of peers in the network. As shown of Figure 5.4 the propagation time scales linearly.

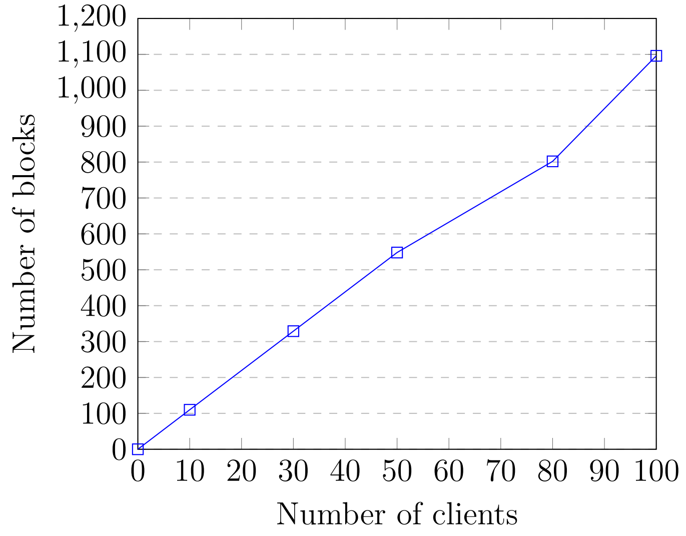


Figure 5.3: Verification of scalability for increasing number of clients

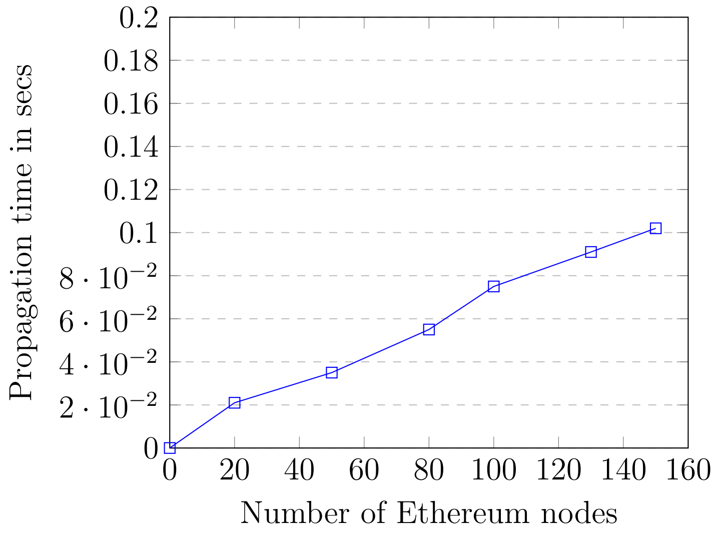


Figure 5.4: Verification of average propagation time in Ethereum network

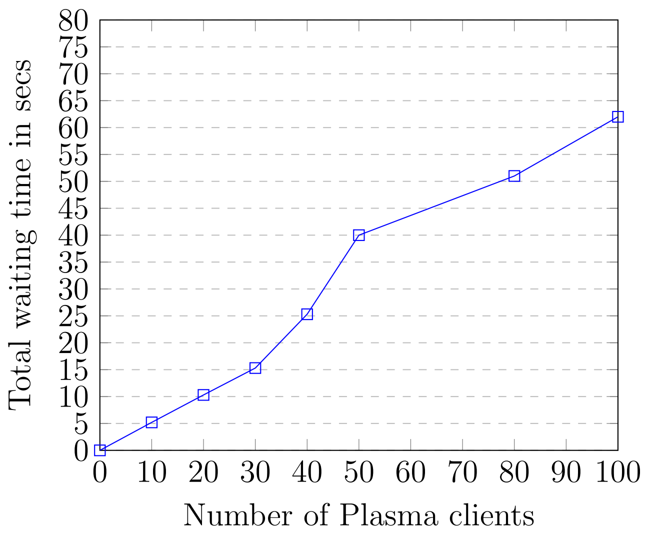


Figure 5.5: Deposit transactions waiting time scales linearly

5.3.3 Deposit transactions waiting time

To start generating transactions, Plasma clients need to first see their deposit transaction in a Plasma block. Deposit transactions are submitted to a Plasma blockchain when the corresponding transactions is included in a block on the Ethereum blockchain. The duration of this operation depends on factors such as gasLimit per Ethereum block, gasLimit per Ethereum transaction and number of clients and we define it as deposit waiting time.
In this experiment we kept the gasLimit of a block at a fixed value of 150 and gasLimit of transaction at 30. The number of clients was increased gradually an we measured the total waiting time of all clients to see their deposit transactions on a Plasma blockchain. The total waiting time is the time from starting the simulation up until the point in time when all deposit blocks are submitted to the Plasma blockchain. Figure 5.5 indicates that the waiting time of deposit transactions scales linearly.

5.3.4 Conclusion

To showcase the scalability capabilities of the system we conducted 3 experiments as we measured the number of blocks in a Plasma blockchain when we increase the Plasma clients as well as the waiting time for deposit blocks. Moreover, we evaluated the block propagation time in Ethereum by increasing the number of Ethereum nodes. The results demonstrate the ability of the simulator to scale linearly with increasing amount of work.

5.4 Flexibility

To be able to perform different scenarios on a system requires a particular degree of flexibility. In general, the quantification of flexibility could be seen as a subjective task as it is quite difficult to identify whether a system is flexible enough. Despite that, we designed our simulator in such a way that it enables experimenting with different parameters that tune the performance of the simulator. The expansion of number of parameters leads to exponential increase of the complexity of the system. For configuring the Plasma simulator, we introduce the following configuration parameters:

- *numberOfEthereumNodes* - number of Ethereum nodes (miners)
- *numberOfPlasmaClients* - number of Plasma clients that each Plasma blockchain contains
- *tokensPerClient* - number of tokens each Plasma client receives at the beginning of a simulation
- *transactionsPerPlasmaBlock* - number of transactions included in a single block on a Plasma blockchain
- *plasmaChildren* - number of Plasma second-level blockchains which report to a first-level Plasma blockchain (parent)
- *numberOfNeighbours* - number of neighbours each node in the simulated P2P-Network is connected to
- *neighboursUpdateRate* - number of seconds after which neighbours of each peer are recomputed and updated
- *blockGasLimit* - gasLimit of an Ethereum block
- *plasmaTXGasLimit* - gasLimit of each transaction created from Plasma participants and submitted to Ethereum network
- *plasmaTXGasPrice* - gasPrice for each transaction created from Plasma participants and submitted to Ethereum network
- *enableExternalTransactions* - a flag to determine whether the simulator should generate externally owned transactions (user transactions) within Ethereum network
- *externalTxGasLimit* - gasLimit of each user transaction
- *externalTxGasPrice* - gasPrice for each user transaction

- *externalTransactionGenerationRate* - number of seconds after which transactions are automatically generated
- *numberOfEthereumExternalAccounts* - number of accounts for which user transactions will be generated
- *amountPerEthereumExternalAccount* - number of tokens each externally owner account receives (used for creation of user transactions)
- *transactionGenerationRate* - number of seconds after which transactions are automatically generated within Plasma blockchain (this rate is true for all Plasma blockchains)
- *numberOfTransactionGenerationIntervals* - number of intervals at which transactions are generated. After completion of the last interval, the simulation ends
- *plasmaBlockInterval* - interval that determines how many deposit blocks can a Plasma contract create between two blocks of transactions generated by an Operator

These parameters are central in order to simulate a complex system such as the Ethereum Plasma blockchain. They give great opportunity of users to experiment with multiple blockchains but come at the cost of missing some details, hence not receiving the expected results. Validation for different combinations of parameters is a good option for future work.

5.5 Extensibility

The simulation system was designed with consideration for future increments. It was developed following the Separation Of Concerns (SoC) process [61] and Single Responsibility Principle (SRP) [62] in attempt to minimize coupling and maximize cohesion.
To demonstrate the extensibility of the simulator, we will showcase two examples for extending functionalities of the simulator.

5.5.1 Support for multiple nested Plasma chains

Currently the simulator supports 2 levels of nested Plasma chains. In theory, the Plasma chains can be nested forever, hence achieving even better scalability. In order to realize whether that is feasible in practice, one could extend the simulator to support n number of nested Plasma blockchains. This could be achieved by implementing additional logic to the Operator that will help all operators (multiple instances of the same verticle) follow closely the parent blockchains.

5.5.2 Implementation of Proof-Of-Stake

Proof of Stake algorithm can be embedded both in the Ethereum blockchain as well as the Plasma blockchain. This can be done by introducing new verticle to one (or both) subsystems that will collect the semi-randomly choose the next block miner based on how much tokens have the nodes deposited as a collateral and a piece of randomness in order to prevent centralization. Currently, there exist multiple miner nodes on the Ethereum blockchain, which can be used as validators. On the other hand, Plasma blockchain(s) currently have one miner (due to using Proof-of-Authority) - Operator. So, in order to implement Proof-Of-Stake consensus in Plasma one should also introduce additional nodes, which should function as validators. Plasma Clients should ideally still be operating.

5.5.3 Conclusion

As demonstrated in the extension proposals, adding new features to the current system will almost inevitably be related to introducing new nodes and some degree of synchronisation between these nodes. Due to the technology we used for the implementation of our simulator, extending it with such ideas in mind should be highly facilitated.
To sum it up, the design of the system enables its extensibility. This has been shown by the two proposed examples.

5.6 Plasma Framework Compliance

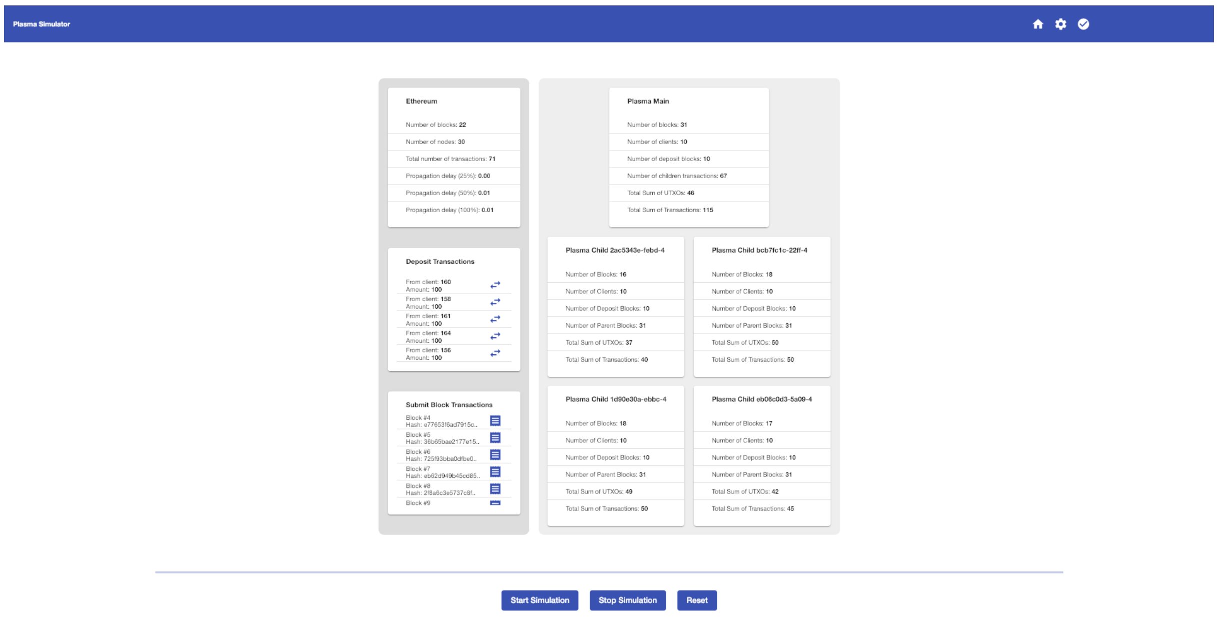
The idea to implement the Plasma Simulator is based on the Plasma Framework white paper [4]. To identify how compliant to the original paper our simulation system is, we compared the proposed features in the paper with the implemented features in our system. Table 5.8 summarizes the results.

	Plasma Framework	Plasma Simulator
Plasma Smart Contract	X	X
Deposits	X	X
Block Submission (Periodic commitments to the Root Chain)	X	X
Simple Withdrawals (Exits)	X	X
Mass withdrawals	X	n.a.
Challenge Exit	X	n.a.
UTXO Model	X	X
Plasma Proof-Of-Stake	X	n.a.
Plasma Proof-Of-Authority	X	X
Blockchain in Blockchains	X	X

Table 5.8: Comparison between features presented in Plasma White Paper and features implemented in the Plasma Simulator.

5.7 Powerful Visuals

To get a better understanding of the dynamics of the simulator we developed a simple user interface that displays relevant data to the user coming from different edges of the simulation. By different edges we mean the Ethereum simulation and the Plasma simulation. In addition, data for second-level Plasma chains is summarized and shows to the user. Furthermore, we added a configuration page, where the user can change the configuration parameters without touching the source code of the system. Screenshots of the web application are demonstrated on Figure 5.6 and Figure 5.7.



Ethereum

Number of blocks: **22**
Number of nodes: **30**
Total number of transactions: **71**
Propagation delay (25%): **0.00**
Propagation delay (50%): **0.01**
Propagation delay (100%): **0.01**

Deposit Transactions

From client: **160** Amount: **100**
From client: **158** Amount: **100**
From client: **161** Amount: **100**
From client: **164** Amount: **100**
From client: **156** Amount: **100**

Submit Block Transactions

Block #4 Hash: e77653f6ad7915c..
Block #5 Hash: 36b65bae2177e15..
Block #6 Hash: 725f93bba0dfbe0..
Block #7 Hash: eb62d949b45cd85..
Block #8 Hash: 2f8a6c3e5737c8f..
Block #9

Plasma Main

Number of blocks: **31**
Number of clients: **10**
Number of deposit blocks: **10**
Number of children transactions: **67**
Total Sum of UTXOs: **46**
Total Sum of Transactions: **115**

Plasma Child 2ac5343e-febd-4

Number of Blocks: **16**
Number of Clients: **10**
Number of Deposit Blocks: **10**
Number of Parent Blocks: **31**
Total Sum of UTXOs: **37**
Total Sum of Transactions: **40**

Plasma Child bcb7fc1c-22ff-4

Number of Blocks: **18**
Number of Clients: **10**
Number of Deposit Blocks: **10**
Number of Parent Blocks: **31**
Total Sum of UTXOs: **50**
Total Sum of Transactions: **50**

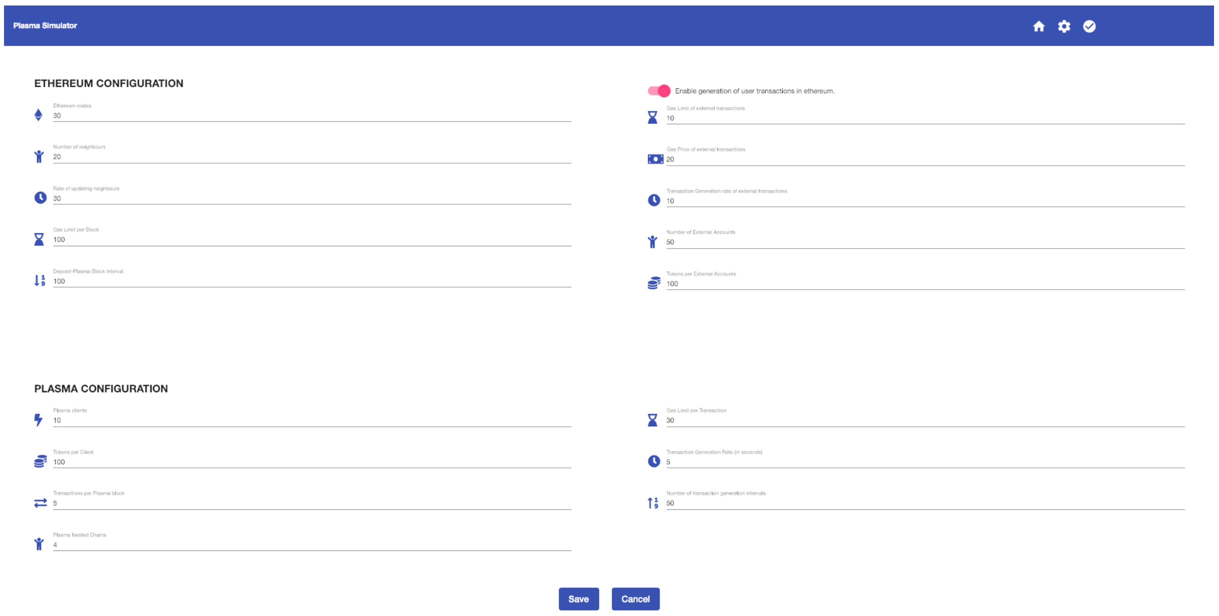
Plasma Child 1d90e30a-ebbc-4

Number of Blocks: **18**
Number of Clients: **10**
Number of Deposit Blocks: **10**
Number of Parent Blocks: **31**
Total Sum of UTXOs: **49**
Total Sum of Transactions: **50**

Plasma Child eb06c0d3-5a09-4

Number of Blocks: **17**
Number of Clients: **10**
Number of Deposit Blocks: **10**
Number of Parent Blocks: **31**
Total Sum of UTXOs: **42**
Total Sum of Transactions: **45**

Figure 5.6: Plasma Simulator Main Page



ETHEREUM CONFIGURATION

Ethereum nodes
30

Number of neighbours
20

Rate of updating neighbours
30

Gas Limit per Block
100

Deposit-Plasma Block Interval
100

Enable generation of user transactions in ethereum.

Gas Limit of external transactions
10

Gas Price of external transactions
20

Transaction Generation rate of external transactions
10

Number of External Accounts
50

Tokens per External Accounts
100

PLASMA CONFIGURATION

Plasma clients
10

Tokens per Client
100

Transactions per Plasma block
5

Plasma Nested Chains
4

Gas Limit per Transaction
30

Transaction Generation Rate (in seconds)
5

Number of transaction generation intervals
50

Figure 5.7: Plasma Simulator Configuration Page

Chapter 6

Summary

In this thesis, we proposed, designed, implemented and evaluated a configurable, highly-scalable, event-driven, and reactive simulator for simulating the behaviour of Plasma - Scalable Autonomous Smart Contracts running on an Ethereum-like simulated blockchain. This is the first simulator that is able to simulate multiple blockchains communicating with each other over an eventbus using the resources of a moderately powerful single computer. In addition, we developed a reactive web application, that is continuously listening for outputs of the simulator and is displaying them in real time on the browser to the user.

6.1 Status

The main goal of this thesis was to build a reliable simulator for experimenting with Plasma blockchains. Our approach for constructing plasma chains is strongly based on the Plasma Framework white paper and successfully implements main features such as asset deposit, block submission and asset withdraw. Furthermore, we developed a way to simulate two levels of nested Plasma blockchains, which, to the best of our knowledge, is the first system that achieves to implement this Plasma concept. Additionally, our system simulates the most relevant for our approach features of Ethereum, which closes the computation cycle of Plasma. The evaluation of the system demonstrates, that the simulator produces reliable results and can be used as a starting point for thoroughly experimenting with Plasma.

6.2 Conclusions

There exist many different simulators that attempt to simulate various blockchains and some of their most important features. Such systems are quite useful when experimenting with one layer blockchains. Despite that, research focused on blockchains shows that off-chain constructs represent a promising approach for managing the blockchains' scaling problem. Most of these constructs are first and foremost theoretical and some communities attempt to implement proof-of-concepts (PoCs) for them. The individual features and characteristics of such PoCs are intensively discussed and developers reason about how exactly such systems should look like. This arises the necessity of simulation systems that are able to simulate two layer blockchains and to enable fast identification of good and bad design decisions. Hitherto such simulators did not exist which is the reason why we started working on this project. We developed a simulator that simulates two layers of blockchains and allows researchers to analyze the behaviour and the characteristics of Plasma side chains. Our design was focused on achieving granularity, so that our simulator can be easily modified and extended in order to simulate different features and approaches. We believe that this work would support researchers and all individuals interested in Plasma to better understand it and to improve its design.

6.3 Future Work

We encourage developers to extend and modify the simulator in order to reason about the variety of approaches that are yet to be developed. In this section we address different possibilities of future works.

6.3.1 Attacks

Achieving secure and unbreakable system is one of the keys for achieving trust. To identify how secure Plasma in reality is, attacks such as block-withholding attack should be ideally included as a part of the simulator.

6.3.2 Challenging exits

The simulator currently does not simulate exit challenges due to the nature of the simulator, namely its fairness. If various attacks are added to the simulator, it would be feasible to implement the *challengeExit* feature of Plasma.

6.3.3 Configuration validation

Our simulator is highly configurable, which makes it powerful but at the same time it represents a weakness, since a particular combination of parameters would make the simulation experiments useless. Thus, implementing a configuration validation would help prevent configuring the parameters in an unrealistic way.

6.3.4 Multiple Plasma Contracts

We currently support only one Plasma contract. Implementing and switching between different contracts would make the simulator more flexible and will allow comparison between different approaches.

6.3.5 Unlimited levels of Plasma chains

Plasma simulator supports two levels of plasma nested chains. Despite that, our system was designed in such a way, that it enables simple extension that would allow unlimited levels of Plasma nested chains. This would inherently mean that the web application should be also updated accordingly.

6.3.6 Plasma Transaction Generation

Creating transactions in Plasma is currently depending on messages sent from the Ethereum manager to all Plasma clients. This makes the simulation time deterministic but rather unrealistic as clients usually create transactions at different times. Adding a periodical timer to each of the Plasma clients, hence removing periodic messages from Ethereum manager, would make the simulation time less deterministic more realistic. The total simulation time would then increase, since the time when a client starts generating transactions will depend on the time when their deposit transactions are included in a block on Ethereum.

6.3.7 Decentralizing Ethereum

Mining blocks in Ethereum currently works on a First-come, First-served principle with a 30% probability that a node mines a block. Improving this probability distribution and letting nodes synchronize their blockchains in an decentralized manner is a great opportunity for future work and for achieving more realistic results on the Ethereum-like blockchain.

6.3.8 Decentralizing Plasma

Each Plasma blockchain in our setup is running a Proof-of-Authority consensus algorithm. As this is rather a centralized and simple approach, extending the simulator to select between different consensus algorithms such as Proof-of-Work and Proof-Of-Stake would be useful for determining best conditions within a Plasma chain.

6.3.9 Network protocols

The Plasma simulator utilizes a simple randomization approach for determining the neighbours a node in the P2P network is connected to. Implementing better neighbours discovery protocols would benefit the behaviour and the accuracy of the simulator.

6.3.10 Dynamic assets withdraw

In the current setup, all plasma clients withdraw their tokens at the end of the simulation. Extending that feature to a dynamic asset withdraw spread over the entire simulation will improve the conditions of Plasma.

6.3.11 Improve web application design

Our web application displays the most relevant variables, important for Plasma. Despite that, showing all UTXOs, all blocks and their transactions would improve the data analysis of the results. This is easy to implement since all this data is already sent to the application, but not yet displayed.

6.3.12 Experiment with interactions between simulators

As discussed in Chapter 3, eVIBES shows a great potential in simulating fully fledged Ethereum. Even though it is implemented in Scala based on the akka framework, an interaction between the Plasma simulator running on vertx and eVIBES could be attempted. A simple approach would be to replace our Ethereum-like simulation with the eVIBES simulation by integrating it into our system using vertx Reactive Streams. [63].

Appendices

Appendix A

Appendix

A.1 Source code

Available at https://github.com/georgiayloff/plasmasimulator

A.2 Source code of UTXOPool

```
class UTXOPool(var poolMap: MutableMap<UTXO, Transaction.Output> =
 ↪ mutableMapOf()) {

  constructor(pool: UTXOPool) : this(pool.poolMap.toMutableMap())

  fun addUTXO(utxo: UTXO, txOutput: Transaction.Output) {
    if(!poolMap.containsKey(utxo))
      poolMap.put(utxo, txOutput)
  }

  fun removeUTXO(utxo: UTXO) {
    poolMap.remove(utxo)
  }

  fun containsUTXO(utxo: UTXO) : Boolean {
    return poolMap.containsKey(utxo)
  }

  fun poolSize() : Int {
    return poolMap.size
  }

  fun getTxOutput(utxo: UTXO) : Transaction.Output? {
    return poolMap.get(utxo)
  }
}
```

A.3 Source code of Operator

```
class Operator: PlasmaParticipant() {
  var transactions = mutableListOf<Transaction>()
  var childChains = mutableListOf<String>()
  var childChainsMap = mutableMapOf<String, NestedChain>()
  var transactionsPerBlock = 0
  var plasmaBlockInterval = 10
  var nextBlockNumber = 10
  var startTime = System.currentTimeMillis()
  var numberOfDepositBlocks = 0
  var numberOfClients = 0
  var receivedDepositBlocks = mutableListOf<Int>()
  var totalChildrenDeposits = mutableListOf<String>()


  private companion object {
    private val LOG = LoggerFactory.getLogger(Operator::class.java)
  }

  override fun start(startFuture: Future<Void>?) {
    super.start(startFuture)
    LOG.info("Operator with $address deployed!")

    transactionsPerBlock = config().getInteger("transactionsPerBlock")
    plasmaBlockInterval = config().getInteger("plasmaBlockInterval")
    numberOfClients = config().getInteger("numberOfPlasmaClients")

    nextBlockNumber = plasmaBlockInterval

    if(config().containsKey("childrenPlasmaChainAddresses")) {
      childChains =
 ↪  config().getJsonArray("childrenPlasmaChainAddresses").
              list.toMutableList() as MutableList<String>
      childChains.forEach { address ->
        childChainsMap.put(address, NestedChain(address,
 ↪  plasmaBlockInterval))
      }
    }

    vertx.eventBus().consumer<Any>("${chain.chainAddress}/" +
            "${Address.PUBLISH_TRANSACTION.name}") { msg ->
      val newTransaction = Json.decodeValue(msg.body().toString(),
 ↪  Transaction::class.java)

      // nested transaction
      if(newTransaction.childChainTransaction) {
        println(newTransaction.childChainData)
      }

      if(chain.validateTransaction(newTransaction, plasmaPool))
```

```
          transactions.add(newTransaction)

        if(transactions.size >= transactionsPerBlock) {
          // create new block
          val newBlock =
↪   createBlock(transactions.take(transactionsPerBlock),
↪   nextBlockNumber, false)
          updateNextBlockNumber()
          if(applyBlock(newBlock))
            transactions =
↪   transactions.drop(transactionsPerBlock).toMutableList()
        }
      }

      vertx.eventBus().consumer<Any>("${chain.chainAddress}/" +
              "${Address.ETH_ANNOUNCE_DEPOSIT.name}") { msg ->
        val data = msg.body() as JsonObject

        val blockNumber = data.getInteger("blockNum")
        val address = data.getString("address")
        val amount = data.getInteger("amount")
        val chainAddress = data.getString("chainAddress")

        if(chainAddress != chain.chainAddress) {
          if(!totalChildrenDeposits.contains(address)) {
            totalChildrenDeposits.add(address)

          }
          // child chain deposit block
          var childDestAddress = chainAddress + "/" +
↪   Address.ETH_ANNOUNCE_DEPOSIT.name
          send(childDestAddress, data)
        }

        if(!receivedDepositBlocks.contains(blockNumber) && chainAddress
↪   == chain.chainAddress) {
          receivedDepositBlocks.add(blockNumber)
          numberOfDepositBlocks ++
          if(numberOfDepositBlocks == numberOfClients) {
            val stopTime = System.currentTimeMillis()
            val elapsedTime = (stopTime - startTime).toDouble() / 1000
            LOG.info("ALL DEPOSIT BLOCKS ARRIVED")
            LOG.info("ELAPSED TIME: $elapsedTime")
          }
          val tx = createTxForDepositBlock(address, amount)

          LOG.info("[$address] Operator received deposit $amount for
↪   $address ")

          val newBlock = createBlock(listOf(tx), blockNumber, true)
          applyBlock(newBlock, true)
        }
      }
    }

    fun createBlock(newTransactions: List<Transaction>, number: Int =
↪   -1, depositBlock: Boolean) : PlasmaBlock {
        val newBlock = PlasmaBlock(number = number,
                                   prevBlockNum = number - 1,
                                   transactions = newTransactions,
                                   depositBlock = depositBlock)

        if(depositBlock) { // deposit transaction block
          val depositTxOutput = newTransactions[0].outputs[0]
          newBlock.merkleRoot =
↪   HashUtils.hash(depositTxOutput.address.toByteArray() +
↪   depositTxOutput.amount.toByte())
          return newBlock
        }

        val blockRoot =
↪   MerkleTreeBuilder.getRoot(newBlock.transactions.toMutableList())
        newBlock.merkleRoot = blockRoot.digest

        return newBlock
    }

    fun applyBlock(block: PlasmaBlock, depositBlock: Boolean = false) :
↪   Boolean {
      if(!chain.validateBlock(block, plasmaPool)) {
        println("invalid block")
        return false
      }

      chain.addBlock(block, plasmaPool)
      if(!depositBlock && chain.parentChainAddress == null) {
```

```
      // deposit blocks come from plasma contract when a client
↪   deposits tokens
      // into the plasma chain, hence such blocks should not be
↪   submitted back
      // to the contract
      rootChainService.submitBlock(from = address, rootHash =
↪   block.merkleRoot, timestamp = block.timestamp)
    }
    //FileManager.writeNewFile(vertx, Json.encode(chain.blocks),
↪   "blockchain.json")
    removeUTXOsForBlock(block)
    createUTXOsForBlock(block)
    LOG.info("[$address] BLOCK ADDED TO BLOCKCHAIN. BLOCKS:
↪   ${chain.blocks.size}")
    LOG.info("[$address] TOTAL SUM OF UTXOs:
↪   ${calculateTotalBalance()}")
    val blockJson  = JsonObject(Json.encode(block))
    send("${chain.chainAddress}/${Address.PUBLISH_BLOCK.name}",
↪   blockJson)
    send(Address.NUMBER_OF_UTXOS.name,
↪   JsonObject().put("chainAddress",
↪   chain.chainAddress).put("numberOfUTXOs", plasmaPool.poolSize()))

    if(chain.parentChainAddress != null) {
      LOG.info("[$address] Create transaction to parent")
      // send the block to the parent chain as a transaction
      val tx = Transaction()
      tx.source = address
      tx.childChainTransaction = true
      tx.childChainData.put("blockNum", block.number.toString())
      tx.childChainData.put("merkleRoot",
↪   HashUtils.transform(block.merkleRoot))
      tx.childChainData.put("timestamp", block.timestamp.toString())

↪   send("${chain.parentChainAddress}/${Address.PUBLISH_TRANSACTION.name}",
↪   JsonObject(Json.encode(tx)))
    }
    // send the block to all child chains since the block could
↪   contain a transaction
    // that came from a child chain
    if(childChains.size > 0) {
      childChains.forEach{ chain ->
        send(chain, blockJson)
        }
      }
      return true
    }

    override fun stop(stopFuture: Future<Void>?) {
      LOG.info("Pending transactions ${transactions.size}")
      super.stop(stopFuture)
    }

    fun send(address: String, msg: Any?) {
      vertx.eventBus().publish(address, msg)
    }

    fun calculateTotalBalance(): Int {
      var total = 0
      plasmaPool.poolMap.forEach { (utxo, output) ->
          total += output.amount
      }
      return total
    }

    fun updateNextBlockNumber() {
      nextBlockNumber += plasmaBlockInterval
    }

    fun createTxForDepositBlock(address: String, amount: Int) :
 ↪    Transaction {
      val tx = Transaction()
      tx.source = plasmaContractAddress
      tx.depositTransaction = true
      tx.addOutput(address, amount)

      return tx
    }

}
```

A.4 Installation Instructions

A.4.1 Prerequisites

Before running the simulator, make sure you that:

- You have Java JDK or JRE version 7+ installed. To check, run *java -version*
- You have gradle installed. To check, run *gradle -v*. Installation instructions can be found on https://gradle.org/install/
- You have node and npm installed. To check, run *npm -v*. Installation instructions can be found on https://nodejs.org/en/download/

A.4.2 Run the simulator

Navigate to *plasmasimulator/simulator* and run *./gradlew clean run*

A.4.3 Run the web application

Navigate to *plasmasimulator/app* and run:

- *npm install*
- *ng serve*

The web application should be available at http://localhost:4200/

Bibliography

[1] Q. H. Vu, M. Lupu, and B. C. Ooi, *Peer-to-Peer Computing: Principles and Applications*. Springer Publishing Company, Incorporated, 1st ed., 2009.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." `https://bitcoin.org/bitcoin.pdf`.

[3] "Bitcoin developer guide." `https://bitcoin.org/en/developer-guide`. Accessed: 2019-01-10.

[4] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts." `https://plasma.io/plasma.pdf`, 8 2017.

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

[6] "Eclipse vert.x." `https://vertx.io/`. Accessed: 2019-01-18.

[7] A. Sangster, "The genesis of double entry bookkeeping," *The Accounting Review*, vol. 91, 01 2016.

[8] N. Szabo, "Smart contracts: Building blocks for digital markets." `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html`, 1996.

[9] V. Buterin, "A next-generation smart contract and decentralized application platform." `https://github.com/ethereum/wiki/wiki/White-Paper`, 2013.

[10] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols(extended abstract)," vol. In: Preneel B. (eds) Secure Information Networks. IFIP — The International Federation for Information Processing, vol 23. Springer, 1999.

[11] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake." `https://pdfs.semanticscholar.org/0db3/8d32069f3341d34c35085dc009a85ba13c13.pdf`, 8 2012.

[12] "Blockchain scaling." `https://medium.com/coinmonks/blockchain-scaling-30c9e1b7db1b`, 8 2018.

[13] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments." `https://lightning.network/lightning-network-paper.pdf`, 1 2016.

[14] e. a. Buterin, Vitalik, "On sharding blockchains." `https://github.com/ethereum/wiki/wiki/Sharding-FAQs`.

[15] B. Asolo, "Breaking down the blockchain scalability trilemma." `https://bitcoinist.com/breaking-down-the-scalability-trilemma/`, 6 2018.

[16] J. Coleman, "State channels." `https://www.jeffcoleman.ca/state-channels/`, 11 2015.

[17] J. Teutsch and C. Reitwießner, "A scalable verification solution for blockchains." `https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf`, 11 2017.

[18] V. Buterin, "Minimal viable plasma." `https://ethresear.ch/t/minimal-viable-plasma/426`, 1 2018.

[19] V. Buterin and e. a. Floersch, Karl, "Plasma cash: Plasma with much less per-user data checking." `https://ethresear.ch/t/plasma-cash-plasma-with-much-less-per-user-data-checking/1298`, 3 2018.

[20] e. a. Robinson, Dan, "Plasma debit: Arbitrary-denomination payments in plasma cash." `https://ethresear.ch/t/plasma-debit-arbitrary-denomination-payments-in-plasma-cash/2198`, 6 2018.

[21] "Plasma mvp." `https://github.com/omisego/plasma-mvp`.

[22] "Mvp of plasma cash." `https://github.com/slockit/plasma-cash-mvp`.

[23] "Plasma cash - erc721/erc20/eth supported." `https://github.com/loomnetwork/plasma-cash/tree/plasma-debit`.

[24] "Ganache." `https://truffleframework.com/ganache`. Accessed: 2019-01-18.

[25] "Hive - ethereum end-to-end test harness." `https://github.com/ethereum/hive`. Accessed: 2019-01-18.

[26] "Ropsten." `https://ropsten.etherscan.io/`. Accessed: 2019-01-18.

[27] "Kovan." `https://kovan.etherscan.io/`. Accessed: 2019-01-18.

[28] "Rinkeby." `https://www.rinkeby.io/`. Accessed: 2019-01-18.

[29] E. Evans, “Domain-driven design reference.” `https://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf`, 2015. Accessed: 2019-01-07.

[30] “The reactive manifesto.” `https://www.reactivemanifesto.org/`, 9 2014. Accessed: 2019-01-07.

[31] “Redux - a predictable state container for javascript apps..” `https://redux.js.org/`. Accessed: 2019-01-07.

[32] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[33] C. Eckert, *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. München: Oldenbourg, 8., aktualisierte und korr. aufl. ed., 2013. Verfasserangabe: von Claudia Eckert ; Quelldatenbank: FHBK-x ; Format:marcform: print ; Umfang: XIV, 1016 S : Ill., graph. Darst. ; 978-3-486-72138-6.

[34] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.

[35] G. Singh and Supriya, “Article: A study of encryption algorithms (rsa, des, 3des and aes) for information security,” *International Journal of Computer Applications*, vol. 67, pp. 33–38, April 2013. Full text available.

[36] “Rsa cryptosystem.” `https://primes.utm.edu/glossary/page.php?sort=RSA`. Accessed: 2019-01-09.

[37] J. Buchmann, “The digital signature algorithm (dsa).” `https://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1003_DSA.pdf`, 12 2001. Accessed: 2019-01-09.

[38] “The md5 message-digest algorithm.” `https://www.rfc-editor.org/rfc/pdfrfc/rfc1321.txt.pdf`, 4 1992. Accessed: 2019-01-08.

[39] “All about sha1, sha2 and sha256 hash algorithms.” `https://www.tbs-certificates.co.uk/FAQ/en/sha256.html`. Accessed: 2019-01-08.

[40] “Sha-3 standard: Permutation-based hash and extendable-output functions.” `https://ws680.nist.gov/publication/get_pdf.cfm?pub_id=919061`. Accessed: 2019-01-08.

[41] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Keccak,” in *Advances in Cryptology – EUROCRYPT 2013* (T. Johansson and P. Q. Nguyen, eds.), (Berlin, Heidelberg), pp. 313–314, Springer Berlin Heidelberg, 2013.

[42] “eth-hash.” `https://eth-hash.readthedocs.io/en/latest/`. Accessed: 2019-01-08.

[43] N. Bauerle, "What is a distributed ledger?." `https://www.coindesk.com/information/what-is-a-distributed-ledger`. Accessed: 2019-01-10.

[44] "What is bitcoin double spending?." `https://www.bitcoin.com/info/what-is-bitcoin-double-spending`. Accessed: 2019-01-10.

[45] "Merkle tree." `https://bitcoin.org/en/glossary/merkle-tree`. Accessed: 2019-01-11.

[46] "Merkle tree & merkle root explained." `https://www.mycryptopedia.com/merkle-tree-merkle-root-explained/`. Accessed: 2019-01-11.

[47] "What is an altcoin?." `https://bitcoinmagazine.com/guides/what-altcoin/`. Accessed: 2019-01-10.

[48] "Ethereum: A secure decentralised generalised transaction ledger byzantium version 69351d5 - 2018-12-10." `https://ethereum.github.io/yellowpaper/paper.pdf`, 2015.

[49] "Proof-of-authority chains." `https://wiki.parity.io/Proof-of-Authority-Chains`. Accessed: 2019-01-16.

[50] 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.

[51] "Akka." `https://akka.io/`. Accessed: 2019-01-18.

[52] "The scala programming language." `https://www.scala-lang.org/`. Accessed: 2019-01-18.

[53] A. Deshpande, P. Nasirifard, and H.-A. Jacobsen, "evibes: Configurable and interactive ethereum blockchain simulation framework," in *Proceedings of the 19th International Middleware Conference (Posters)*, Middleware '18, (New York, NY, USA), pp. 11–12, ACM, 2018.

[54] "Plasma mvp specification." `https://www.learnplasma.org/en/resources/#plasma-mvp-specification`. Accessed: 2019-01-18.

[55] M. Richards, *Software Architecture Patterns*. O'Reilly Media, Inc., 2015.

[56] "Kotlin." `https://kotlinlang.org/`. Accessed: 2019-01-18.

[57] "Gradle build tool." `https://gradle.org/`. Accessed: 2019-01-18.

[58] "Typescript - javascript that scales." `https://www.typescriptlang.org/`. Accessed: 2019-01-18.

[59] “Angular.” https://angular.io/. Accessed: 2019-01-18.

[60] “Ngrx - reactive state for angular.” https://ngrx.io/. Accessed: 2019-01-18.

[61] “Separation of concerns, explained.” https://blog.ndepend.com/separation-of-concerns-explained/. Accessed: 2019-01-16.

[62] “Single responsibility principle: A recipe for great code.” https://www.toptal.com/software/single-responsibility-principle. Accessed: 2019-01-16.

[63] “Vert.x reactive streams integration.” https://vertx.io/docs/vertx-reactive-streams/java/. Accessed: 2019-01-18.