



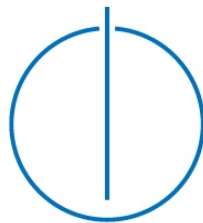
**Technische Universität
München**

Fakultät für Informatik

Master's Thesis in Informatik

Design and Implementation of an Ethereum-like Blockchain
Simulation Framework

Aditya Shrikant Deshpande





**Technische Universität
München**

Fakultät für Informatik

Master's Thesis in Informatik

Design and Implementation of an Ethereum-like Blockchain
Simulation Framework

Design und Implementierung eines Ethereum-ähnlichen
Blockchain-Simulationsframeworks

Author: Aditya Shrikant Deshpande

Supervisor: Prof. Dr. Hans-Arno Jacobsen

Advisor: MSc. Pezhman Nasirifard

Submission: 15.11.2018

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

München, 15.11.2018

(Aditya Shrikant Deshpande)

Abstract

Cryptocurrencies and Distributed Ledger technologies, such as Bitcoin, Ethereum and Hyperledger have received widespread attention over the past few years. With the increasing popularity of Ethereum, there has been an interest in understanding the internals of the system. The decentralized nature of Ethereum and other blockchain technologies makes it difficult to understand how the individual components work. A comprehensive investigation of properties of the network thus plays a significant role in understanding its workings and its further development. However, due to the heavy system requirements for deploying a full Ethereum blockchain and high running costs for simulating the blockchain network this becomes a challenging task. In this work, we propose eVIBES, an event-driven, parallel, message-oriented, broadcast-based configurable Ethereum simulation tool for simulating large-scale Ethereum networks. eVIBES with its configurable nature can help users to simulate large-scale Ethereum networks and analyze them on a personal computer. eVIBES can thus serve as a useful tool for professionals and beginners alike for gaining practical insights into the system properties and internal working of Ethereum blockchain by running simulations with varying parameters and then analyzing the system behavior.

Acknowledgment

I thank my thesis advisor, MSc. Pezhman Nasirifard for his support throughout the thesis work; helping improve the quality of thesis by sharing his knowledge and being available for discussions within a short notice. I thank him for giving me the freedom to take technological decisions concerning the thesis. I would also like to thank Prof. Dr. Hans-Arno Jacobsen for the opportunity to write a thesis at the chair of Business Information Systems at TU Munich.

Contents

List of Figures	5
List of Tables	6
Abbreviations	6
1 Introduction	7
1.1 Motivation	7
1.2 Problem Statement	8
1.3 Approach	9
1.4 Contribution	9
1.5 Organization	10
2 Background	11
2.1 Cryptography	11
2.1.1 Public-key Cryptography	11
2.1.2 Digital Signatures	12
2.1.3 Cryptography in Blockchain Systems	12
2.2 Peer-to-Peer Networks	12
2.2.1 Network Architecture	13
2.3 Nodes	14
2.4 Consensus Protocols	14
2.5 Distributed Ledger	14
2.6 Blockchain	15
2.6.1 Mining	15
2.7 Ethereum	16
2.7.1 Ethereum as a State Transition Machine	16
2.7.2 Value	18
2.7.3 World State	18
2.7.4 Transactions and Messages	19
2.7.5 The Block	19
2.7.6 Gas and Payment	21
2.7.7 Block Execution	22

2.7.8	Node Types	23
3	Related Work	24
3.1	Ganache	24
3.2	HIVE	25
3.3	P2P Network Emulators	26
3.3.1	Peersim	26
3.3.2	OPNET	26
3.4	VIBES	26
3.4.1	Architecture	27
3.4.2	Issues	27
4	Approach	29
4.1	Actor Model	29
4.2	Extending VIBES for Ethereum Blockchain	30
4.2.1	Enhancements to Overcome the Issues	30
4.3	Architectural Decisions	30
4.3.1	Enhancing the Existing VIBES Architecture	30
4.3.2	Decision	32
4.4	System Architecture	33
4.4.1	Orchestrator	33
4.4.2	Reducer	34
4.4.3	Ethereum Client	34
4.4.4	Streams	34
4.4.5	Database	36
4.5	System Design	36
4.5.1	Structural Design	36
4.5.2	System Behavioural Design	40
4.6	Prerequisites	45
4.6.1	Akka Actors	45
4.6.2	Akka Streams and Akka HTTP	47
4.7	Implementation	48
4.7.1	Node Hierarchy	48
4.7.2	Configuration Parameters	48
4.7.3	Dashboard Statistics	49
4.7.4	Simulation Initialization	50
4.7.5	Boot Node Selection for Client Initialization	51
4.7.6	Neighbourhood Creation in Boot Nodes	52
4.7.7	Proof of Work Simulation	53
4.7.8	Transaction Generation	53
4.7.9	Maintaining Account states	54
4.7.10	Tracking Executed Transactions	54
4.7.11	Priority Inbox	54

4.7.12	Tracking the Longest Subtree in Blockchain	55
4.7.13	State Management of the Simulated Client	55
4.8	Technology	57
4.8.1	Backend	57
4.8.2	Frontend	57
5	Evaluation	59
5.1	Correctness	59
5.1.1	Transaction Execution	59
5.1.2	Block Verification	61
5.1.3	Blockchain Verification	62
5.1.4	Conclusion	62
5.2	Scalability	63
5.2.1	Node Creation	63
5.2.2	Varying Number of Neighbors	64
5.2.3	Varying Number of Accounts	64
5.2.4	Conclusion	66
5.3	Flexibility	66
5.4	Extensibility	66
5.4.1	Side chains	67
5.4.2	Contracts	67
5.4.3	Conclusion	67
5.5	Powerful Visuals	67
6	Summary	69
6.1	Status	69
6.2	Conclusions	69
6.3	Future Work	70
6.3.1	Smart Contracts	70
6.3.2	Comparative Analysis Tools	70
6.3.3	Network Behavior Simulation	70
6.3.4	Network Topology	70
6.3.5	Detailed Analysis of a Node	71
6.3.6	Extending the Simulator for Testing Side-Chains	71
	Appendices	72
A	Appendix	73

List of Figures

2.1	Linking of blocks to form of a Blockchain [1]	15
2.2	Ethereum state transition function [1]	17
2.3	Visualization of a State Trie in the Ethereum Blockchain [2]	20
2.4	Steps involved in Ethereum Block Computation	22
3.1	A view of the User Interface - Ganache [3]	25
3.2	VIBES architecture [4]	27
4.1	Suggested Model System architecture	31
4.2	System architecture of eVIBES	33
4.3	Use case diagram for Orchestrator Actor	37
4.4	Use case diagram for Reducer Actor	38
4.5	Use case diagram for Node Actor	39
4.6	Use case diagram for EVM-Primary Actor	40
4.7	Use case diagram for TxPooler Actor	41
4.8	Use case diagram for Accounting Actor	41
4.9	Working of eVIBES- Behaviour Diagram	42
4.10	Node Discovery Flowchart	43
4.11	Transaction Execution Flowchart	44
4.12	Call stack of a single and multithreaded implementation [5]	45
4.13	Message flow between akka actors [6]	46
4.14	Actor supervision [7]	47
4.15	Actor Hierarchy in eVIBES	48
4.16	State transition of the Node actor	55
4.17	State transition of the Transaction pooler actor	56
4.18	State transition of the EVM Primary actor	56
5.1	Verifying the scalability of the simulator-varying number of nodes with 5 BOOT NODES	63
5.2	Verifying the scalability of the simulator-varying number of nodes with 10 BOOTNODES	64
5.3	Verifying the scalability of the simulator-varying number of neighbors. . . .	65
5.4	Verifying the scalability of the simulator-varying number of accounts. . . .	65

5.5	eVIBES Dashboard	68
5.6	eVIBES Welcome screen	68

List of Tables

2.1	Ether value	18
5.1	Transaction execution evaluation	60
5.2	Block Generation evaluation	61

Chapter 1

Introduction

Humans are at the cusp of redefining the existing financial system from the ground up. This debacle started with the creation of a new alternate economy on the internet by the introduction of Bitcoin [8]. The noteworthy part of this arrangement was that the Bitcoin was not offered by a central authority, but by a decentralized system which was patrolled algorithmically and consisted of a network of users. Blockchain - the technology behind the Bitcoin, is designed to achieve Byzantine fault tolerance [9] and can reach consensus without the need of trust between the networked users [10]. This new system eliminated the need of delegates involved in the traditional financial system and placed a decentralized web of connected systems in its place. Over the years new systems were introduced that enhanced the Bitcoin architecture. One of the leading blockchain technology among them is the Ethereum Blockchain [1]. The current market share of Ethereum is \$21,447,799,407 which ranks it second by the total worth, one place below Bitcoin (as of today) [11]. One of the fascinating features of Ethereum which has made it a popular Blockchain system is its Turing complete nature [12]. This ability allows the creation of smart contracts and decentralized applications using the Ethereum platform.

1.1 Motivation

The essence of the Blockchain systems architecture is Decentralization [13]. Each node in the network is self-governing and behaves independent of other nodes (Physical Decentralization) [13], yet the overall system state is the same, i.e., Logical centralization [13]. This logical centralization is achieved with the help of consensus algorithms. The physical decentralization being one of the crucial properties of the Ethereum blockchain

becomes a roadblock when it comes to studying the behavior of these systems. A study to understand the properties of the system can be done by creating a private network and running the nodes equivalent to the Ethereum mainnet. However simulation of this scale is an impossible feat without multiple machines having above average hardware. One way to analyze a decentralized, distributed system is by creating a simulator. A simulator built to mimic the workings of the system can give a good approximation about the system properties. In this thesis, we propose the development of such a system that simulates an Ethereum Blockchain-like system, which will help in studying the properties of the Ethereum blockchain.

1.2 Problem Statement

The Ethereum tools ecosystem is rich with multiple tools such as Ganache [3], Hive [14] and other peer-to-peer network tools such as PeerSim [15], OPNET [16], but all have different objectives when it comes to testing the blockchain. None of these tools share the vision to analyze the behavior of the system as a whole but act as tools to support the testing of Ethereum smart contracts or Decentralised applications aka Dapps. Existing tools offer a way to create simulations of the network by creating a private local network, but this too is limited in the number of maximum possible simulated nodes. Ganache [3] is a tool for creating a private Ethereum blockchain used for testing. However, it does not capture how the nodes work among each other and how they reach consensus.

On the other hand Hive [14] is an excellent tool for testing the clients in the Ethereum blockchain by creating docker images for the clients. However, simulating representative percentage of nodes present in the current blockchain system on a laptop or a personal computer is not possible. There does not exist a system, as per our research, that solves the motive of simulating the Ethereum Blockchain without creating full-fledged node instances, for testing the properties of the system.

As a part of this thesis, we propose the development of a system to simulate the working of Ethereum blockchain. Our goal is to develop eVIBES in a way that closely mimics the Transaction execution, Block generation and Node Discovery of an Ethereum client. We designed eVIBES in a way to keep track of all the account states and versions of Blockchain on individual simulated nodes thereby giving the user a more granular view and insight into the workings of the Blockchain.

The overall aim is to build a simulator that is,

- **Correct:** Correct here refers to the ability of the simulator to behave closely to the actual ethereum blockchain system.
- **Scalable:** Scalability is the comparison of the time taken to run the simulation when the amount of work increase over time.
- **Extensible:** Addition of new features should be easily possible in the system.
- **Flexible:** The system should be flexible to create Ethereum networks in multiple configurations.
- **and Powerful visuals:** The simulator should provide a good interface to interact and also a detailed level of analysis of the results.

With these aims, the eVIBES simulator would establish as a valuable tool for understanding and analyzing the properties of the Ethereum blockchain. It can prove to be a useful tool for protocol developers, Ethereum users, and academicians interested in the Ethereum Blockchain.

1.3 Approach

To address our aim, we started the system design by following the domain design principles [17] to ensure the compartmentalization of the features and building a system with a good separation of concerns. To ensure the scalability of the system, we selected technologies that would achieve our goals with minimum boilerplate code. Moreover following to the principles of Reactive manifesto [18] ensured we built a reactive and extensible system. Evaluation of the system helped to determine the correctness of the system and rendered the system useful.

1.4 Contribution

The research that followed during the thesis presented the need of an Ethereum blockchain simulator that is low cost and works on computers having average resources (Personal computers). eVIBES fills this void by offering a simulator that is flexible and offers the ability to build an ethereum-like blockchain with ease thereby enabling the study of the properties and the behavior of the ethereum blockchain on computers with

average resources. eVIBES also satisfies the need for understanding the state of individual blockchain network components by providing data of expected granularity.

1.5 Organization

Organization of the thesis is as follows: Chapter-2 starts by discussing the theoretical knowledge that lays the foundation to understand the system, i.e. eVIBES. Chapter-3 provides insights into the related works. In the related works, we discuss different tools in the Ethereum blockchain ecosystem and explain how these tools fit our aim. Chapter-4 elaborates the system design and the rationale behind all the design decisions. The chapter also details the workings of the system by explaining the underlying algorithms and their implementation. In Chapter 5 we explain the process of validating and verifying the system by performing a series of evaluations. In the last section, we provide the current status of the system and offer our observations, conclusion and possible future work.

Chapter 2

Background

The following section presents the topics that provide the necessary theoretical foundation to understand the thesis. We start with explaining the basic technologies like Cryptography, Peer-to-Peer networks and Distributed ledgers. These technologies form the building blocks towards understanding the Ethereum blockchain. We then deep dive in the workings of Ethereum as it forms the foundation of the work presented in the thesis.

2.1 Cryptography

Cryptography is the study of hiding/ securing information and propagation of the information securely in a way that only the intended recipient can access it. The central idea behind cryptography is to provide a secure way for information transfer in an unsecure medium thus preventing any malicious entity from accessing it [19]. Information is hidden at the source in a way which can only be decoded by the intended recipient. The information is encoded at source using a mathematical function(s). The encoded information is known as cipher. Cypher is then sent to the sink through an unsecured medium. The cipher gets decoded once it is received at the sink. This type of cryptography is called Symmetric-key cryptography. The name, "Symmetric-Key cryptography" comes from the fact that same key is used for encoding and decoding of the information [20].

2.1.1 Public-key Cryptography

Unlike Symmetric-key cryptography which uses one key for encryption and decryption, Public Key cryptography uses two keys, private and public [21]. A public key is shared

among all the stakeholders, and all the stakeholder parties have their own private keys. Encryption is done using both the public and the private keys thereby ensuring that the decoding is possible by someone who has the same public key and their own private key. Public key cryptography enables the creation of digital signatures which can be used to validate the integrity of the shared data. Digital signatures provide the necessary data integrity in the blockchain system.

2.1.2 Digital Signatures

Digital signatures are a mechanism to present the authenticity of the digital data mathematically [21].

They are one of the critical building blocks in the blockchain system.

Goals achieved by a digital signature are,

- Authentication: Verification of the document author is possible from the digital signature of a digitally signed document.
- Non- repudiation: The digital signature can be verified, and hence the author cannot deny the existence of such a document.
- Data integrity: A slight alteration in the document changes the digital signature of the document. Hence any alteration in the document can be verified thereby maintaining the data integrity of the document.

2.1.3 Cryptography in Blockchain Systems

Cryptography serves two main purposes in a blockchain based system,

- Ensuring the data integrity of the saved old records on the blockchain.
- A way to ensure the security, ability to verify the source (non-repudiation) and the authenticity of the sender of the transaction.

2.2 Peer-to-Peer Networks

A Peer-to-Peer network is a network formed by connecting two or more devices without the need of a central coordinator. All the devices in the network have the same level of privileges and participate equally in the network [22].

Unlike the traditional client-server model where there is a strict separation of the roles, consumer and the producer, In the peer to peer system, this separation is avoided as a device can be both a consumer and a producer.

2.2.1 Network Architecture

Although the nodes are connected directly on the application layer, this is not the case on the network layer. This network of nodes on the application layer is known as an overlay network. Data is exchanged using the regular network layer protocols however the application communicates based on the application layer protocol [23]. The overlay networks help define peer discovery and communication protocols which are independent of the network layer protocols i.e. the physical network topology.

Based on how the devices are connected in the overlay network, we can classify the network topology in three categories.

- Structured Networks
- Unstructured Networks
- Hybrid

Structured Networks

In structured networks, the overlay network organizes the nodes/ devices to a specific network topology that ensures effective communication.

Unstructured Networks

No structure is imposed in unstructured peer-to-peer networks. These networks are formed when nodes randomly connect to form connections.

Hybrid Networks

Hybrid networks are a combination of both structured and unstructured networks. One of the conventional models is to have an unstructured network and a structured client-server model that helps unstructured network to find peers.

2.3 Nodes

A node is a device on the network that forms the foundation of the blockchain network. Each node runs a copy of a blockchain client that performs block verification and generation among other tasks. Each node connects with other nodes in an unstructured manner thereby forming the entire blockchain network. Each node is autonomous and thus the system, architecturally and politically decentralized [13]. As each node hosts a similar copy of the ledger, it makes the network logically centralized.

Nodes being part of the blockchain system provide their valuable resources for validating blocks and transactions and collect the transaction fees for their contribution. This process is called mining.

2.4 Consensus Protocols

In a distributed system it is essential to achieve overall system reliability in the presence of other faulty participants. Consensus protocols help these systems to reach an agreement over the value of the data that is shared among all the entities in the network.

2.5 Distributed Ledger

A distributed ledger (also called a shared ledger, or Distributed Ledger Technology, DLT) is a consensus of replicated, shared, and synchronized digital data geographically spread across multiple sites, countries, or institutions [24]. A distributed ledger consists of multiple clients spread across the geography that host a copy of the database and communicate with each other using consensus protocols to maintain a consistent state of the database. Distributed ledger can be categorized into multiple types based on their architecture. They can require permission or be permission-less depending upon if anyone is allowed to validate transactions or if a fixed set of users are granted permission to access the data and validate it. They can also vary depending on the use of consensus algorithms.

2.6 Blockchain

Blockchain is a distributed public ledger of information. The key aspect that separates blockchain from other ledger technologies is its decentralized nature. There is no governing authority that manages the working of the blockchain instead it is hosted by multiple devices in the network. Each device holds a local copy of the current state of the ledger and updates it regularly to keep it consistent with the other copies present on the network.

Each of the device in the network is connected to few other devices in the network thereby forming a peer-to-peer network and a decentralized system. Devices communicate with each other using cryptographic mechanisms for secure communication. The central idea behind a blockchain is formation of blocks out of the information it receives. A collection of these blocks connected to each other using cryptographic mechanisms, consequently forming a chain is called as a blockchain. As shown in the 2.1 hash of the previous block is stored by the next block, forming a chain of blocks.

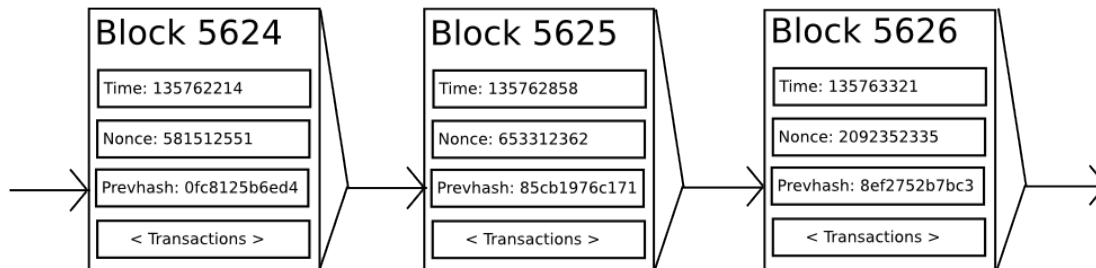


Figure 2.1: Linking of blocks to form of a Blockchain [1]

2.6.1 Mining

The process of creating blocks by computing the received transactions is known as mining. The generated blocks act as a permanent record asserting the execution of a transaction. Miner nodes in the network perform mining of the blocks. If the block is successfully mined then the miner node receives a prize for the work done.

2.7 Ethereum

As explained in the official documentation of Ethereum [1],

“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”

Ethereum has re-purposed the distributed ledger model that was proposed in the Bitcoin to model it as a virtual computer. It does this by giving the machine level opcodes the same level of certainty as Bitcoin transactions [1].

The Ethereum network uses its built-in currency, Ether, for providing a mechanism for efficiently exchanging digital assets and as a means to provide value to the nodes that perform the mining of the blocks. As mentioned in [25] the primary purpose of Ethereum is not a currency application. However, for execution of a transaction on every node and creation of blocks, there is an inherent cost that the network incurs. Apart from that, some malicious users might tend to exploit the network and abuse its resources. To avoid such abuse of the computing power, all the expenditures incurred are assigned a cost, and network cost unit, known as gas is defined to measure it. Everything that consumes resources has a specific gas value associated with it and to expend the resources in the network; one is required to exchange Ether for the amount of gas required to complete the task.

Ethereum forms a central part of the thesis, and hence we will discuss the Ethereum blockchain in detail in the next sections.

2.7.1 Ethereum as a State Transition Machine

Ethereum can be thought as state transition machine, with its initial state as a genesis state and with successive transactions changing the state thereby moving the system from the initial state to some final state. Transactions represent a way for the system to move from one state to other. Formally we can define a state transition as explained in the official Ethereum client specification [12] as follows,

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T) \quad (2.1)$$

where Υ is the Ethereum state transition function. T is a transaction and σ is the state

of the system.

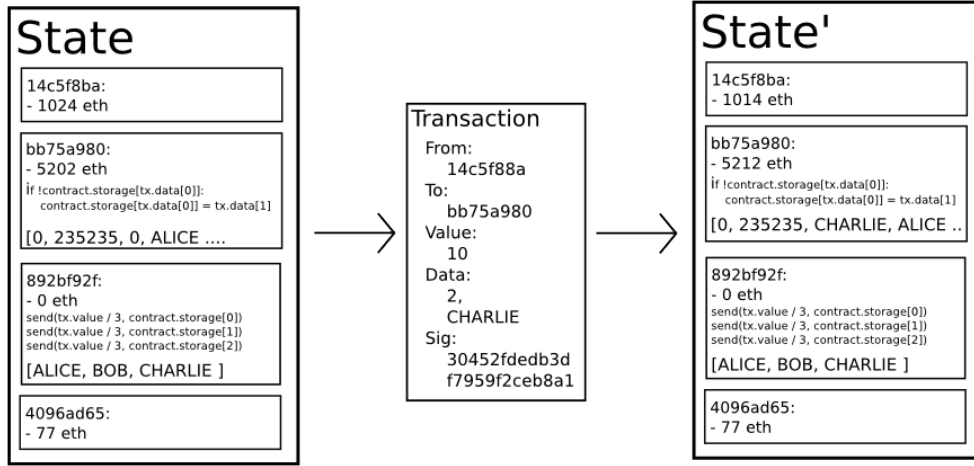


Figure 2.2: Ethereum state transition function [1]

The formula 2.1 is taken from the official Ethereum client specification [12]. Based on a gas limit, transactions are then collected and combined in a block. These blocks are then chained together using cryptographic mechanisms thus forming a blockchain. Creation of such blocks leads to incentivization of the miner. This process is a state transition function that adds value to the miner's account. In the later sections we discuss mining in detail.

Formally this can be stated as,

$$\sigma_{t+1} \equiv \Pi(\sigma_t, B) \quad (2.2)$$

$$B \equiv (... , (T_0, T_1, ...), ...) \quad (2.3)$$

$$\Pi(\sigma, B) \equiv \Omega(B, \Upsilon(\Upsilon(\sigma, T_0), T_1)...) \quad (2.4)$$

Where Ω is the reward function that rewards the miner once the block is finalized; B is this block. Block includes a series of transactions and other components; and Π is the state-transition function on the block-level.

All the formulas mentioned above 2.2, 2.3 and 2.4 are taken from the official Ethereum client specification [12]. Visually Figure 2.2 gives an impression of how a transaction enables the state transitions in a system. These transitions are at the heart of the

Multiplier	Name
10^0	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

Table 2.1: Ether value and sub-denominations [12]

Ethereum blockchain system, and we will discuss them further in the next sections.

2.7.2 Value

As mentioned above, In order to expend the resources of the network, a user has to exchange Ether which is the built-in currency of the Ethereum, for the amount of gas used or to put it simply, for the work done. This currency is also known as ETH and Wei is the smallest sub-denomination of Ether. Other sub-denominations of Ether are mentioned in the Table 2.1.

All the values in the thesis mentioned henceforth will all be computed in Wei as a unit for Ether. Also, the values generated in the eVIBES system are also in Wei.

2.7.3 World State

Unlike Bitcoin, Ethereum uses an account based ledger. Each distinct address represents a separate unique account in Ethereum. World state is referred to as a mapping between addresses (20 byte) and the accounts state. All the state transfers in the network operate on these account states. These account states are not maintained on the blockchain but are stored on the clients.

Ethereum accounts contain four fields,

- nonce: A counter ensuring that each transaction is processed only once.
- ether balance
- contract code, if present

- Account's storage (empty by default)

Accounts can be categorized into two types, externally owned accounts that are controlled by private keys and contract accounts that are controlled by their contract code. The origin of any transaction is an account that is always controlled by a private key.

2.7.4 Transactions and Messages

A transaction is a data packet containing a message that is sent from the externally owned accounts [1]. The sender signs all the transactions. There are two types of transactions, Wallet to smart contract and Wallet to Wallet. Wallet here referring to an externally owned account. Transactions contain the following data,

- Recipient
- Sender's signature
- Amount of Ether to transfer
- START GAS value: Maximum amount of gas allowance for the execution of the transaction
- GASPRICE value: Fees the sender would pay per expended gas unit.
- Optional data field

Messages are very similar to transactions except that they are only sent between contracts and do not exist on the blockchain. Messages are not mined unlike Transactions. Whenever a contract calls a method on another contract a message is sent. Whereas when an externally owned account calls a method on a contract, a transaction is sent.

2.7.5 The Block

A Block is a snapshot of the state of the Ethereum blockchain at a specific time-step. It is expressed as a three-tuple of (H, U, T) where H is the block header, T is the set of transactions and U a set of block headers that are known to have a parent equal to the present block's parents (Ommers blocks). Each block contains a hash of its parent thereby linking it to the previous block and creating a chain of blocks.

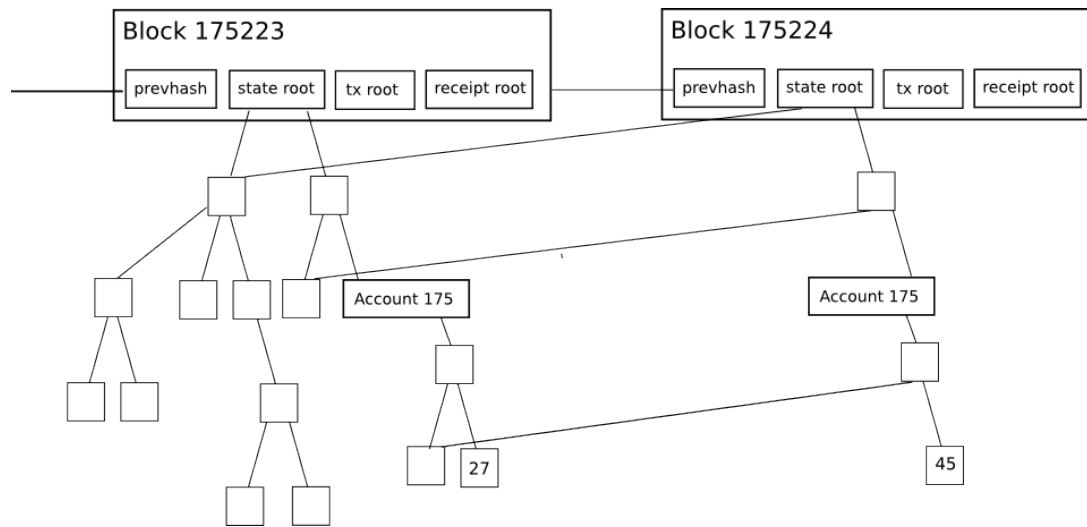


Figure 2.3: Visualization of a State Trie in the Ethereum Blockchain [2]

Structure of a Block in Ethereum Blockchain

As seen in the Figure 2.3 state root, tx root and receipt root are amongst other components of a block. Each of these is a value of the hash of the root of a tree. This tree for each component stores the values of all the instances of those specific components. Below, let us see the contents of each of these trees.

State Trie

Each client has a copy of a global state trie which it stores in a database. A state tree contains values of all the accounts that the client has encountered till date. The state trie is implemented using a special data structure known as Merkle Patricia tree [2] with path to access the state as a sha3 [26] hash of the account address and the value as a serialized version of the account. As already discussed above, account contains storage root as one of its component. This storage root is a hash value of another Merkle Patricia trie.

Storage Trie

All the contract data for an account is stored in the Storage trie. Each account is assigned a new instance of the storage tree.

Transaction Trie

Each block contains a transaction trie that stores all the transactions executed while mining the block. Once generated this trie is not updated again. The miner also assigns a transactionIndex which represents the order in which the transactions were executed while mining.

Transaction Receipts Trie

Transaction receipts trie keeps track of the receipts generated after the transaction execution. Similar to the Transaction trie each block contains one transaction trie and once generated, it is not updated again.

2.7.6 Gas and Payment

Owing to the Turing complete nature of Ethereum, all the computations are subject to a fee. The fee schedule is expressed as the mapping of the amount of gas utilized to a fixed Ether value. Thus for any computation done, there is a fixed agreed upon cost that one has to pay in terms of gas. A few flags are put in place to handle the tracking and usage of the gas to avoid abuse.

GasLimit

Every transaction has a GasLimit associate with it. A miner is not allowed to use gas more than the GasLimit mentioned in the transaction. Before execution of the transaction, ether value which is equivalent to the gas is deducted from the account of the sender. After the transaction execution the remaining ether value is transferred back to the sender's account. If while execution the gas usage exceeds the GasLimit then the transaction is considered invalid.

GasPrice

GasPrice is the price of one unit of gas in Ether. This price is mentioned in the transaction by the sender. Transactors are free to specify any GasPrice. The miners are also free to choose the transactions that yield them higher rewards.

2.7.7 Block Execution

In this section we discuss the process of Block creation in Ethereum as it forms one of the core parts in the implementation of the eVIBES system. Initially, all the transactions that are received by the client are collected in a pool of transactions. The miner can set a specific ordering to the transaction pool. Once the miner decides upon the set of transactions that he wishes to execute the client then selects a parent block from the local version of the blockchain and determines the Uncle blocks using the modified version of the GHOST protocol [27]. The client then executes all the selected transactions and updates the affected accounts accordingly in the global state trie. The transaction trie and the transaction receipts trie are updated. The process of Proof of Work computation starts and if successful then the miner's account is then incremented with the reward and the block is published to other nodes in the network starting from the neighboring nodes.

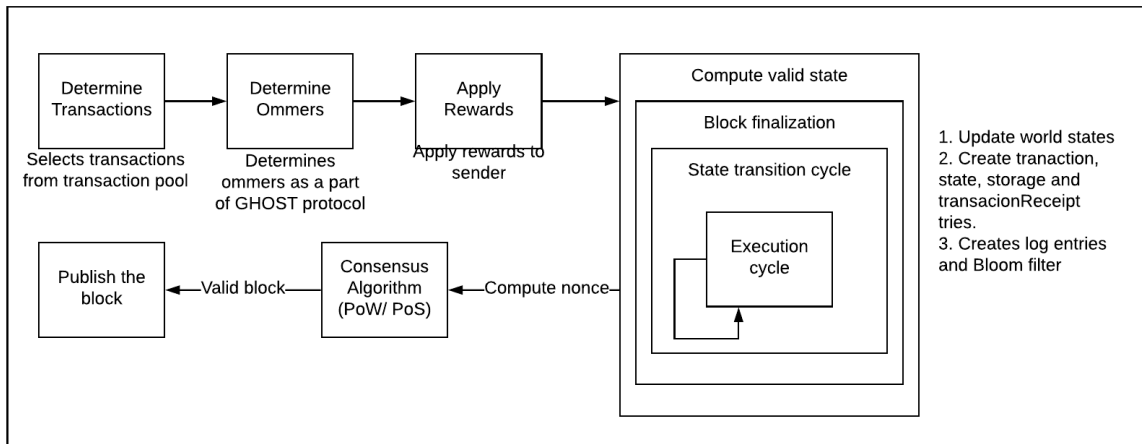


Figure 2.4: Steps involved in Ethereum Block Computation

Intrinsic Validity of a Transaction

Before executing any transaction, the Ethereum client ensures that the transaction has not been executed before. The Ethereum client performs a test to determine the validity of the transaction. The test is explained in the steps below as mentioned in the official ethereum client implementation specification [12],

- The Transaction is well-formed RLP [28] with no additional trailing bytes.
- Signature of the transaction is valid.

- the transaction has a valid nonce.
- The gas limit mentioned is not less than the intrinsic gas value used by the transaction.
- the account balance of the sender contains the amount more than the upfront cost for transaction execution.

If all the above conditions are satisfied, the transaction is said to be a valid transaction.

2.7.8 Node Types

Full Nodes

Full nodes form the foundation of the Ethereum blockchain network. They store the copy of the entire blockchain. All the transactions are sent to the full node which are then propagated to other nodes in the network. The full nodes are responsible for syncing their local blockchain copy with the network.

Light Nodes

Light nodes are called so as they do not have to store the entire blockchain. Instead, they connect to a full node for a copy of the blockchain.

Solo Miners

Solo miners try to mine the block on their own. In order to do so, these nodes keep a synced copy of the entire blockchain.

Mining Pools

Mining pools are a cluster of nodes that use their combined power to mine a block. Once the block is mined, the mining rewards are then distributed among the participating nodes.

Chapter 3

Related Work

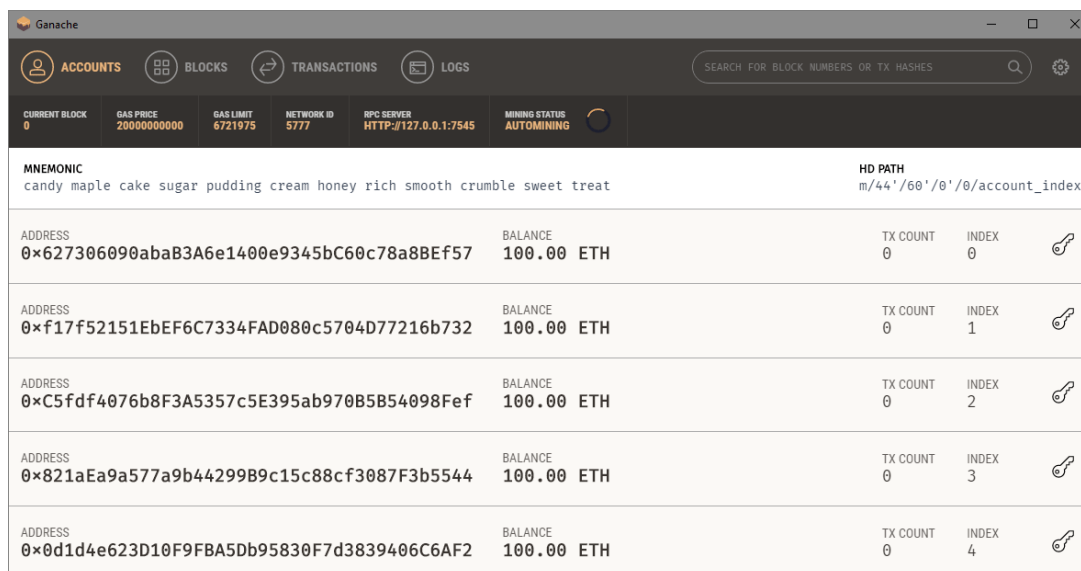
In this chapter, we discuss the related work that has been done towards ethereum blockchain simulation. We also discuss how eVIBES compares to them.

3.1 Ganache

Ganache [3] is the latest rebranding of the testRPC framework. It is mainly useful for creating blockchain for testing the contract code. Ganache offers the user with the ability to create accounts, Blocks, Transactions that can be used for testing, deploying the contract code and testing the application code for Daaps in ethereum.

Taking a brief look at the UI offered by the Ganache framework as seen in the figure 3.1 the ganache UI can be divided into four parts,

- Accounts: Account related information.
- Blocks: Information on each mined block and the corresponding gas used and transactions executed.
- Transactions: List of transactions executed on the blockchain.
- Logs: Generated logs.



The screenshot shows the Ganache application window. At the top, there are tabs for ACCOUNTS, BLOCKS, TRANSACTIONS, and LOGS. Below the tabs, there is a search bar and a status bar with various metrics: CURRENT BLOCK (0), GAS PRICE (2000000000), GAS LIMIT (6721975), NETWORK ID (5777), RPC SERVER (HTTP://127.0.0.1:7545), and MINING STATUS (AUTOMINING). The main content area displays a table of accounts with the following data:

MNEMONIC		HD PATH	
candy maple cake sugar pudding cream honey rich smooth crumble sweet treat		m/44'/60'/0'/0'/0/account_index	
ADDRESS	BALANCE	TX COUNT	INDEX
0x627306090abaB3A6e1400e9345bC60c78a8BEf57	100.00 ETH	0	0
0xf17f52151EbEf6C7334FAD080c5704D77216b732	100.00 ETH	0	1
0xC5fdf4076b8F3A5357c5E395ab970B5854098Fef	100.00 ETH	0	2
0x821aEa9a577a9b44299B9c15c88cf3087F3b5544	100.00 ETH	0	3
0x0d1d4e623D10F9FBA5Db95830F7d3839406C6AF2	100.00 ETH	0	4

Figure 3.1: A view of the User Interface - Ganache [3]

The primary aim of ganache varies strongly with that of eVIBES. Ganache does not offer a way to create a blockchain network that can be analyzed to study the behavior of the network by tweaking system parameters. As mentioned earlier, Ganache offers a way to test contract code whereas the primary aim of eVIBES is to offer a framework for simulating scenarios on the simulated ethereum blockchain and help the user understand the properties of the system. So although the Ganache framework simulates a blockchain which makes it similar to the eVIBES, the purpose behind the use of the simulated blockchain are different.

Ganache served as an inspiration for the UI design behind the eVIBES simulator.

3.2 HIVE

HIVE is an end to end ethereum test harness. The main purpose of HIVE is to offer a way to test the ethereum client implementations against the ethereum official client specifications [12]. Although unit testing of the clients is fairly straightforward, much pre-setup is involved when it comes to testing how the clients behave when interacting with other clients [14]. HIVE offers a way to perform a black box testing of the client implementation.

For testing the client implementations, HIVE offers a way to simulate client networks and test/analyze the interactions between clients to study the working of the client under test.

This way HIVE offers a peek into how the clients behave while interacting but it does this without creating a blockchain that can be analyzed. It offers a simulation for the node behavior.

3.3 P2P Network Emulators

With an aim to simulate the behavior of Blockchain we came across some Peer-to-Peer network emulators like PeerSim [15], OPNET [16] and TestNet [29]. Although all these simulators fail to offer a granular view into how the blockchain is maintained in each node, we took a look at them to understand the architectural choices made to develop these emulators.

3.3.1 Peersim

PeerSim is a simulation tool for simulating P2P networks. Instead of simulating the behavior of the blockchain clients, we can treat them as P2P nodes and study the interactions between them. PeerSim offers a way to implement the protocols that govern the working of the p2p network by offering a JAVA API making the simulation flexible.

3.3.2 OPNET

OPNET [16] network simulator offers the ability to simulate any networks but with a limitation of a select number of protocols. Workings of the underlying protocols are fixed and cannot be changed. The simulator does not offer ways to implement new protocols.

3.4 VIBES

VIBES [4] stands for Visualization of Interactive, Blockchain, Extended simulations, is a simulation framework designed to simulate blockchain networks on a computer with average resources. VIBES offers a way to simulate the blockchain network at an event level. VIBES served as a starting point to the work done in eVIBES, and hence internals of VIBES are discussed here.

3.4.1 Architecture

VIBES offers a user interface that can be used to configure the simulation to build blockchain networks in multiple configurations. Figure 3.2 gives a view of the VIBES system architecture. Coordinator in the simulator controls the working of the simulation. Each blockchain client is represented as a Node in the simulation. The Reducer collects all the results generated during the simulation and then passes them to the coordinator at the end of the simulation.

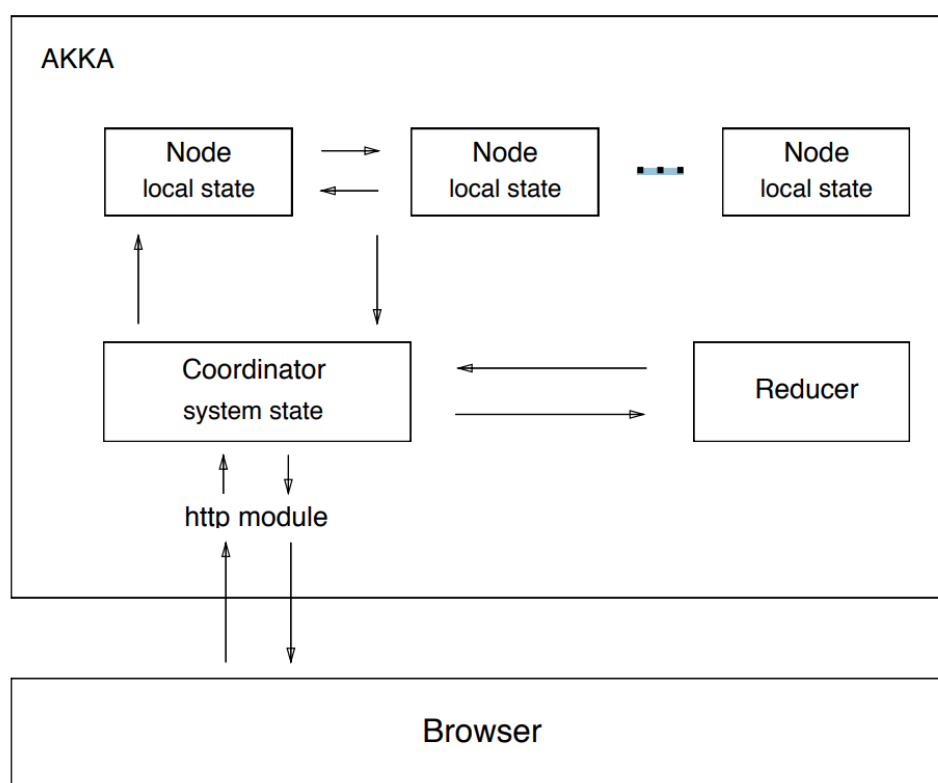


Figure 3.2: VIBES architecture [4]

3.4.2 Issues

VIBES offers a solution to simulate the working of the Bitcoin blockchain [8] and does not capture the internal state of nodes. As mentioned earlier VIBES offers a way to simulate the events in the system it ignores the implementation of block formation, transaction execution in the simulation. eVIBES aims to create a simulation that would offer a way to understand the internals of the node (Ethereum client) as well as how these nodes

interact with each other. Below are some of the issues we found for extending VIBES to satisfy the aims mentioned above.

Coordinator controls the direction of the blockchain:

In standard blockchain network, there is no central authority, a consensus is reached amongst nodes and then propagated. In VIBES the coordinator decides on the consensus and hence the direction of the blockchain. This makes it difficult to simulate ethereum as unlike bitcoin, ethereum depends on Ommers block verification as a part of security in GHOST protocol. These Ommers nodes are hard to simulate in this architecture.

Side-chains are not possible:

As the coordinator controls the flow of the blockchain, this inherently makes the blockchain global. In order to simulate an side-chain we need a new coordinator in VIBES. The coordinator created in VIBES is also responsible for the working of the simulation. Multiple responsibilities render the existing blockchain faulty for executing off-chain features.

Unable to simulate network level protocols:

In the current implementation, there is no provision to simulate network level protocols used by nodes to communicate with each other.

Fast-forward implementation is rendered not useful in Ethereum:

Unlike Bitcoin, Blocks in ethereum are created based on a GasLimit and not in a constant BlockTime. The idea of fast-forwarding in the simulation mainly tries to overcome the 10-minute block time in Bitcoin. The concept of Co-ordinator acting as an authority in consensus is rendered useless in Ethereum, as the block creation on each node can happen simultaneously and independently.

Chapter 4

Approach

In this chapter, we talk about the design and the implementation details for the eVIBES simulation platform. We start by discussing the Actor model which serves as a prerequisite in the implementation of eVIBES. We, later on, discuss enhancements in VIBES and the design and implementation details of the eVIBES system.

4.1 Actor Model

In Computer science, an Actor model can be explained as a mathematical model of concurrent computations [30]. Similar to the object-oriented paradigm where everything is modeled as objects, here Actors are the universal primitive used for modeling a system. Actors can communicate with each other using messages. In response to a message that an Actor receives it can,

- create more actors (finite amount)
- map a message received to a specific action, i.e., perform a specific behavior on a message.
- send messages (finite) to other actors.

the Actor model of computation forms a core part of the eVIBES implementation.

4.2 Extending VIBES for Ethereum Blockchain

As discussed in the previous chapter implementation of an Ethereum simulator using VIBES posed issues. Below we discuss the enhancements required in the system that would overcome discussed issues.

4.2.1 Enhancements to Overcome the Issues

New actor for managing network level protocols

An entity to simulate network level protocols would make the architecture modular and allow for addition of new protocols in the future.

Nodes communicate with each other

With each node communicating with neighbor nodes we can create a simulation of a network that is similar to a peer-to-peer network.

Splitting the Master/ Coordinator Actor

Splitting allows us to separate the concerns of handling ethereum related messages as well as simulation control messages.

4.3 Architectural Decisions

In this section, we discuss the architectural decisions taken towards the development of the eVIBES simulation framework.

4.3.1 Enhancing the Existing VIBES Architecture

As VIBES was a starting point for the development of the simulation we initially planned to enhance the VIBES system to implement the Ethereum blockchain. Below we discuss the changes required to the VIBES framework.

Changes

Considering the issues and the ethereum network architecture, the architecture as shown in the Figure 4.1 offers a way to solve all the issues discussed in Chapter 2. Let us discuss these changes one by one.

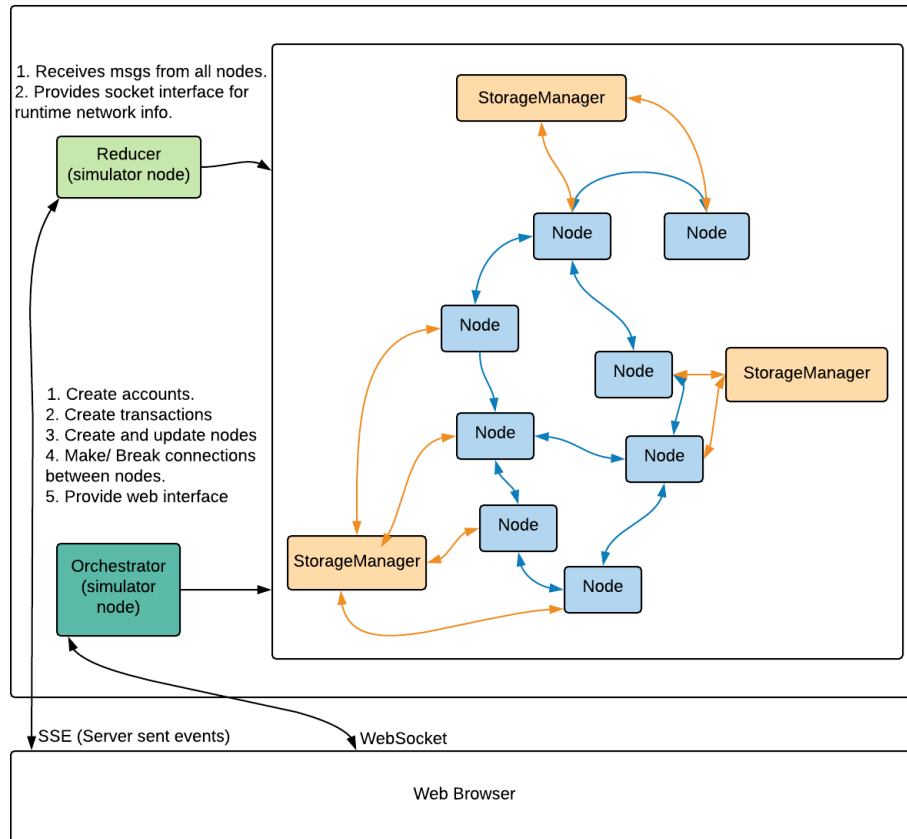


Figure 4.1: Suggested Model System architecture

- **Simulation related blocks:** These are the blocks that act as helpers in the simulation. Orchestrator (Similar to Master Actor) as the name suggests orchestrates the entire simulation by initializing the entire network. The orchestrator is also responsible for creating new accounts (once during the initialization) and transactions (periodically). Network monitor tracks the health of the blockchain network and in case of exceptions intimates the Orchestrator to take proper action. StorageManager is responsible for providing a means for the Nodes to communicate with the database. All the requests to the StorageManager go through a load balancer to distribute the load. In the Ethereum Node, the responsibility of database

interaction lies with each node, but doing this might reduce the speed of the execution. A block executing the functionality of the Ethereum Virtual machine would be a part of the simulation blocks as well.

- **Network Related blocks:** Although not shown in the diagram, these blocks are responsible for the simulation of communication protocols. This separation of concerns makes it possible to change the Network related code without affecting the working of the blockchain specific code. Some of the protocols simulated in these blocks include a Node Discovery Protocol [31]
- **Blocks simulating Blockchain functionality:** As in the Ethereum blockchain network, the main and the only component of the network is a Node. The simulation tries to mimic the working and functionality of the Ethereum blockchain and hence has a Node as an entity which simulates the actual client on the Ethereum blockchain network.

4.3.2 Decision

Considering the issues and the enhancements that would go towards extending the VIBES simulator, below we discuss the advantages offered by the enhancements and explain the decisions taken towards the architecture of the eVIBES system.

Advantages

The suggested enhancements offer advantages over the existing VIBES model. Below are some of those advantages,

- Separation of concerns from Master Actor makes the system more flexible.
- Independence to simulate communication protocols.
- Ability to create multiple blockchains (Off-chains) in parallel.
- Nodes communicating with each other for consensus makes the system similar to the existing real-world ethereum system.
- Ability to manipulate network parameters on the fly.

As many changes suggested would have led to breaking changes in the code for the VIBES bitcoin blockchain. We decided to implement a new system altogether.

4.4 System Architecture

Owing to the decision of creating a new system from scratch, Figure 4.2 gives an overview of the new system architecture for eVIBES. In this section, we discuss all the components of the system architecture in detail.

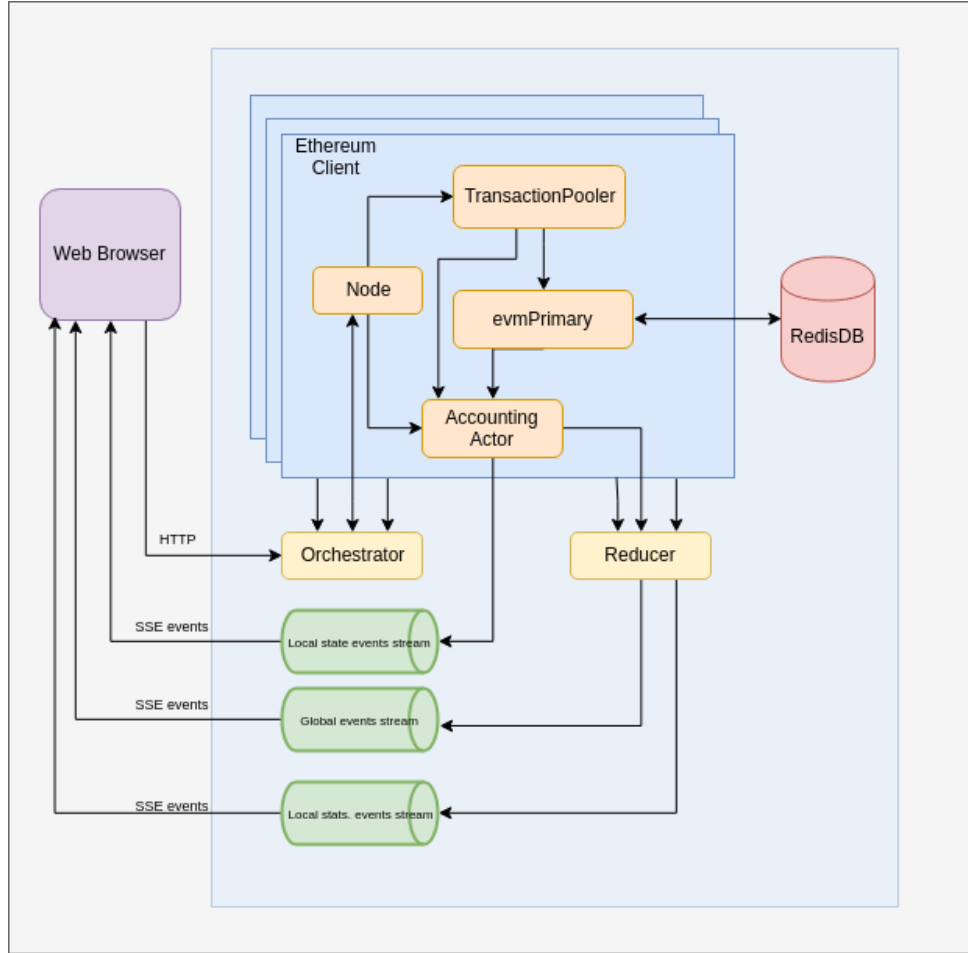


Figure 4.2: System architecture of eVIBES

4.4.1 Orchestrator

The Orchestrator node is the primary component that controls the simulation. It is responsible for receiving the simulation configuration from the users and setting up the simulation. The tasks performed by the Orchestrator include the creation of all Ethereum nodes, accounts and periodical transactions generation. The user can start/stop

and control the simulation parameters during its execution by communicating with the Orchestrator. Orchestrator, as seen in the System architecture diagram, is a singleton object. In order to extend the simulation to run multiple simulations, one has to spawn multiple orchestrator actors.

4.4.2 Reducer

The Reducer is responsible for generating the output of the simulation and presenting the data to the user. Similar to the Orchestrator, Reducer is a singleton. Reducer is a child actor of the Orchestrator.

4.4.3 Ethereum Client

The functionality of the Ethereum client is simulated by three actors, Node Actor, Tx Pooler Actor, EVM Primary actor. None of these are singleton objects. Each of the actors in the client will be spawned for every client required in the simulation.

4.4.4 Streams

As the direction of the data flow is from the Simulation to the browser, and none of the data flows in the opposite direction (except control signals), the Simulator makes use of Server sent events [32] streams instead of a 2-way socket connection. The simulation uses three streams to transfer data from the simulation to the web client.

Local State events

The local state events stream updates the user about the state of all the actors in the client. State diagrams presented in Figure: 4.16, Figure: 4.17 and Figure: 4.18 all detail the maintained states and the transitions performed by the child actors of a Node. We talk about these transitions in the later sections.

Local Data events

The local data events stream carries all the data generated by the individual client/Node Actor. It carries all the data as a JSON document. Below is the list of entities of data

that are transferred, per client. A new copy of the data is sent to the frontend for node discovery and block generation related change events.

- `clientId`: Id of the simulated client.
- `blockNum`: Number of the latest mined or verified block.
- `timestamp`: Timestamp generated before sending the event.
- `blockTime`: Time taken for the verification/ generation of the latest block.
- `avgBlockTime`: Average time taken for block verification/ generation.
- `difficulty`: Difficulty of the last mined block.
- `avgDifficulty`: The average difficulty of all the mined blocks.
- `gasLimit`: Gas limit of the last mined block.
- `avgGasLimit`: Average gas limit of all the mined blocks.
- `peers`: Latest number of peers connected to the client .
- `avgPeers`: Average number of peers connected from the start of the node till now.
- `pendingTx`: Latest number of pending transactions in the transaction pooler actor.
- `avgPendingTx`: Average pending transactions in the transaction pooler over time.
- `poolGasAcc`: Latest value of the summation of the gas. limit of all the transactions in the transaction pool.

Global Data events

The global data events stream carries the average data computed by considering the latest data from all the clients in the simulation. Similar to the other streams, global data streams carries data as a JSON document. Data in these document is a subset of the stats that are transferred by the Local data steam. The transferred stats are Latest BlockNumber, AverageBlockTime, AverageDifficulty, AverageTransactionCost, AverageGasSpending, AverageGasLimit, AverageUncleCount, AveragePeers, AveragePendingTransactions, AveragePropagationTime.

4.4.5 Database

Generally, in an Ethereum Client, Merkle Patricia trie [2] is used to keep track of the system state. A copy of a global Merkle Patricia tree is kept in the memory to speed up the execution in the ethereum client implementations. A key-value database enables the creation of the trie. To get rid of the implementation complexity of the trie data structure we decided to use a key-value database instead directly. RedisDB [33] being the database with the highest throughput and high data ingestion rate was our choice. Data from all the clients is stored in a single RedisDB instance. Stored data includes Transaction data, Account state transition data, i.e. change in account state from each transaction, Block data, Accounts states and client data like the location data.

4.5 System Design

Now that we have an understanding of the blocks involved in building the eVIBES system, we turn our attention to address the structural design and the behavioral design of the components involved in the system.

4.5.1 Structural Design

Structural design focusses on the static aspects of the system such as objects, attributes, and their relations. Below we discuss the structural design aspects of the eVIBES system.

Use case Diagram: Orchestrator

Orchestrator acts as a control center for the simulator. All the control signals are collected here and then transferred to the simulation. Signals such as START and STOP are received by a GET request from the browser client. Apart from the control signals, Orchestrator is also responsible for initializing the BOOT NODES and the FULL NODES before the start of the simulation. We will talk about FULL NODE and BOOT NODE in the implementation section. Orchestrator also creates transactions periodically depending upon the input transaction rate and the number of transactions in a batch. As seen in the Use case diagram for Orchestrator Figure: 4.3, the creation of BOOT NODES involves all the steps required for the creation of FULL NODES. Hence it will not be wrong to say that BOOT NODES are a special type of FULL NODES.

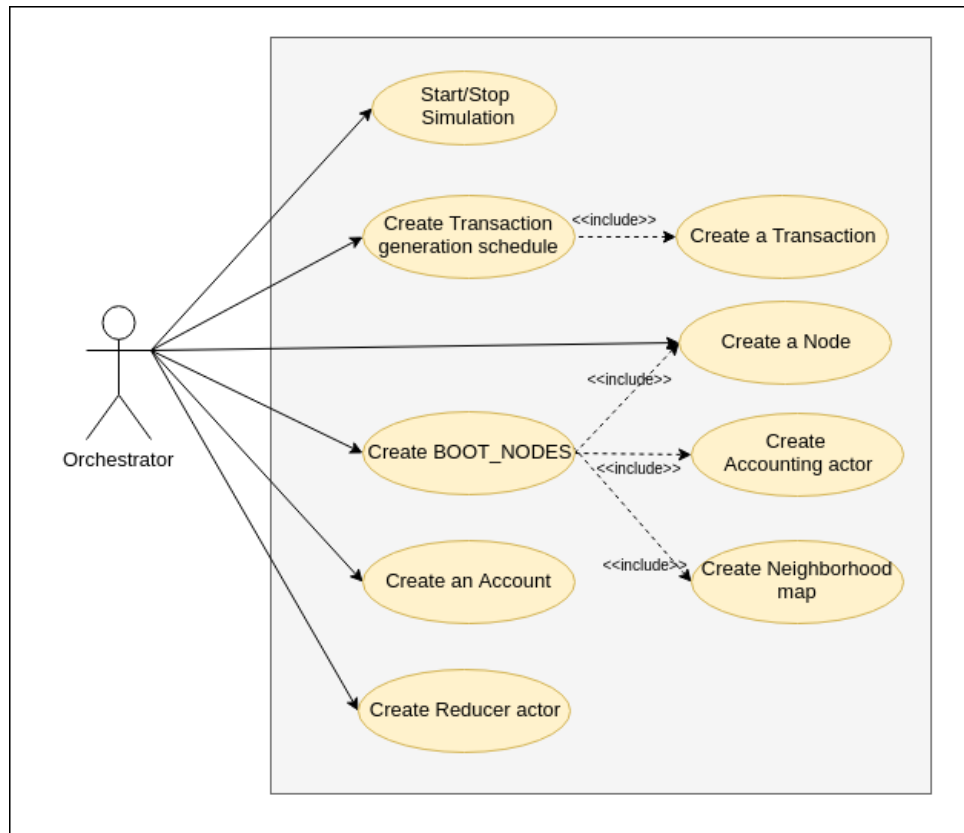


Figure 4.3: Use case diagram for Orchestrator Actor

Use case Diagram : Reducer

Reducer Actor is responsible for collecting all the global statistics of the simulation and forwarding it to the web client. It does this by creating a Server Sent Events stream and passes the global statistics to the streams in regular intervals. Apart from that, the reducer also keeps track of all the latest local statistics of the node. These local latest objects are referred to as Shadows in the Reducer use case diagram. Reducer using these Shadows computes the global statistics (averages) for the entire simulation. We will talk about the statistics computed by the Reducer in the Implementation section. All the use cases handled by the Reducer are shown in the Figure: 4.4

Use case Diagram : Node

Node actor simulates the working of an Ethereum client as per the official ethereum client specification [12]. All the use cases related to the Node are visualized in the Use case diagram in the Figure: 4.5 Apart from initializing other worker actors like TxPooler,

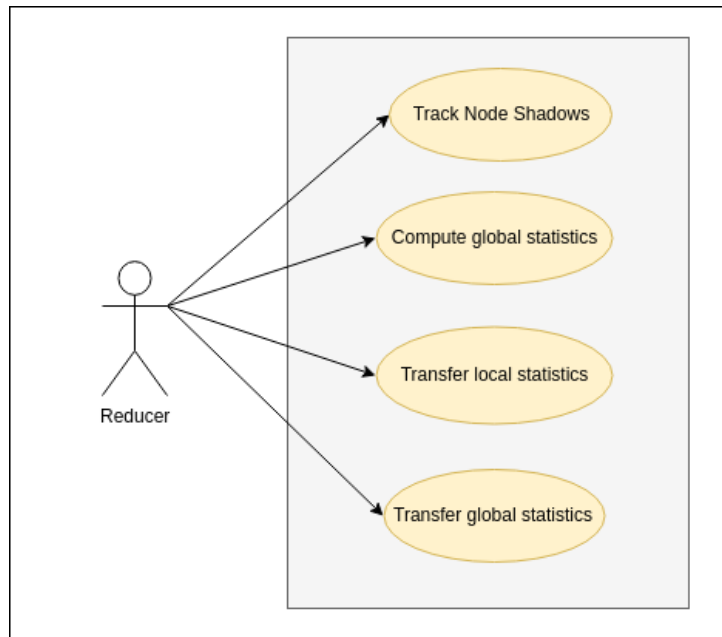


Figure 4.4: Use case diagram for Reducer Actor

EVM-Primary and the Accounting actor, it is responsible for handling the propagation of Transactions and Blocks. It is also responsible for neighbour discovery and maintaining a routing table of all the reachable nodes.

Use case Diagram : EVM-Primary

EVM-Primary, as the name suggests mimics the EVM of the Ethereum client. It is responsible for transaction execution / verification, block creation/ verification. Discussion of the steps involved in transaction creation and block creation is elaborated further in the implementation section. A detailed view of all the use cases implemented by the EVM-Primary can be viewed in Figure: 4.6

Use case Diagram : TxPooler

Transaction Pooler actor is a child actor of Node. Its primary purpose is to maintain a pool of transactions ordered by the GasLimit in a descending order thereby maximizing the profits of the miner. Every time the combined GasLimit of the transactions in the pool exceeds the given GasLimit, a list of transactions is created from the pool and sent to the EVM-Primary for block creation. In order to simulate the randomness in achieving the Proof of Work, the simulation has introduced a probability function that mimics the

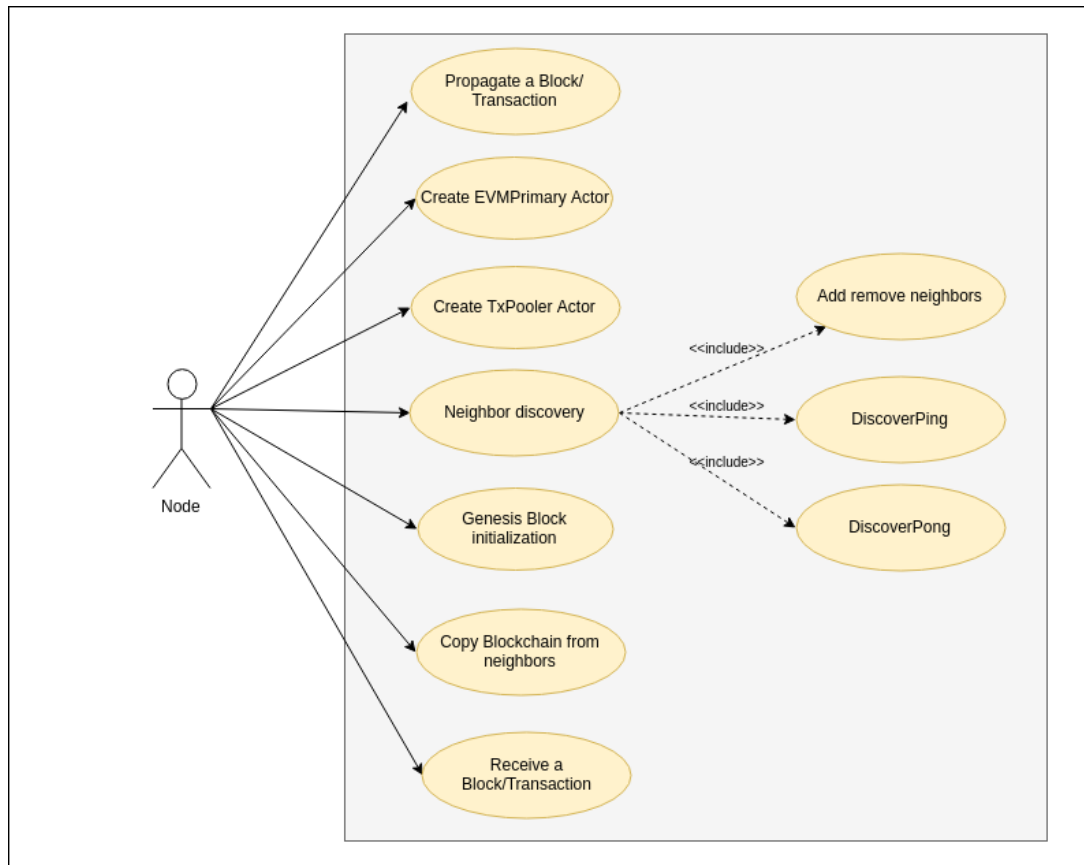


Figure 4.5: Use case diagram for Node Actor

ability of a node to mine a block successfully. If the result of the probability function is true then the computed transaction list is sent to the EVM-Primary for block creation. Apart from that the Transaction Pooler periodically sends the state of the pool to the Accounting actor. A view of all the use cases of a Tx-Pooler actor are visually stated in the Figure 4.7

Use case Diagram : Accounting Actor

Accounting actor is a child actor of the Node. Figure 4.8 visualizes the use cases handled by the Accounting actor. It is responsible for keeping track of all the statistics related to a Node. All the statistics generated by a Node are sent to the Accounting actor for processing. The accounting actor collates all the statistics and then sends them to the Reducer. State-related statistics are sent directly to the Web client.

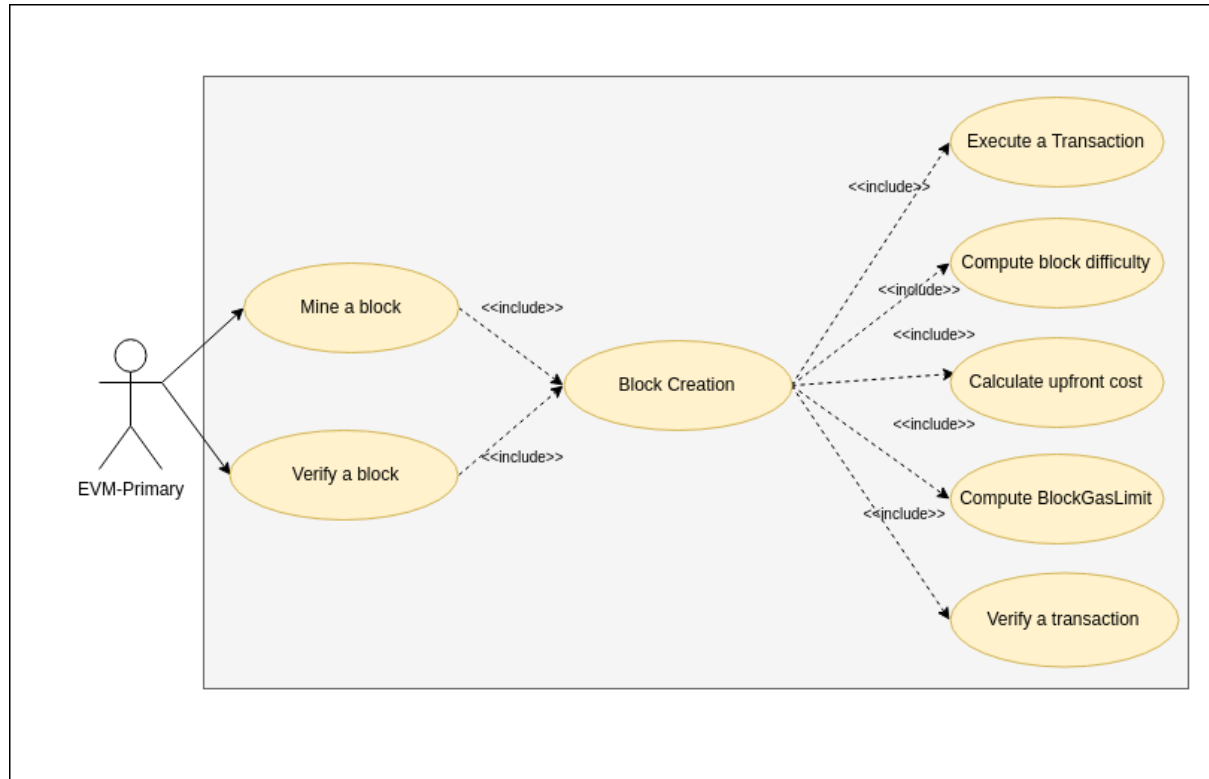


Figure 4.6: Use case diagram for EVM-Primary Actor

4.5.2 System Behavioural Design

System behavior design explains the dynamic properties of the system. This section explains the design details of all the algorithms used in the eVIBES.

Simulator Initialization

Simulator Initialization starts with the creation of an Orchestrator Actor by initializing it with the input parameters. Orchestrator then handles the initialization phase by creating a fixed number of BOOT NODES. The number of BOOT NODES is specified in the settings if not specifically mentioned by the user of the simulation. The concept of BOOT NODES originates from the implementation of the ethereum clients. For every new client that wants to connect to the ethereum blockchain, the implementation of the client offers a few hardcoded entries in the client code. The client then uses these entries to copy the blockchain and the neighbor information from these nodes and updates its local copy of the blockchain. The simulation follows a similar approach for adding new clients to the simulation. Hence the Orchestrator node starts BOOT NODES (number mentioned in the

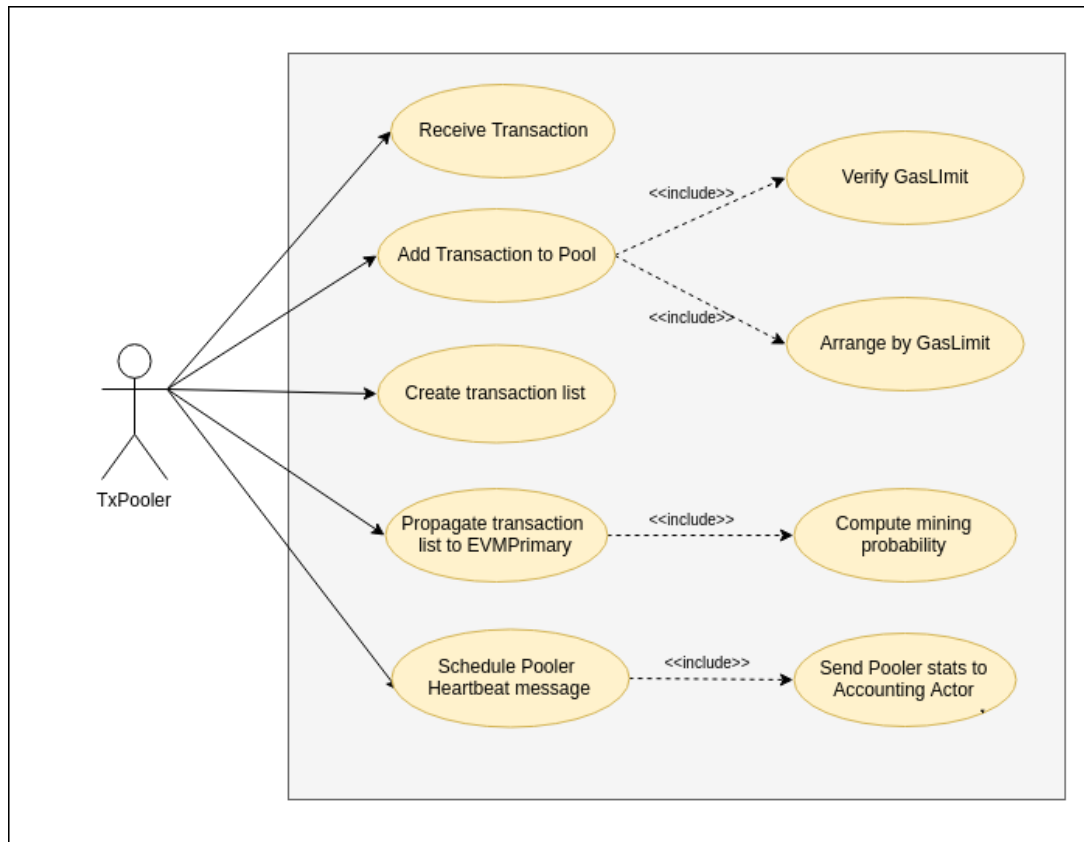


Figure 4.7: Use case diagram for TxPooler Actor

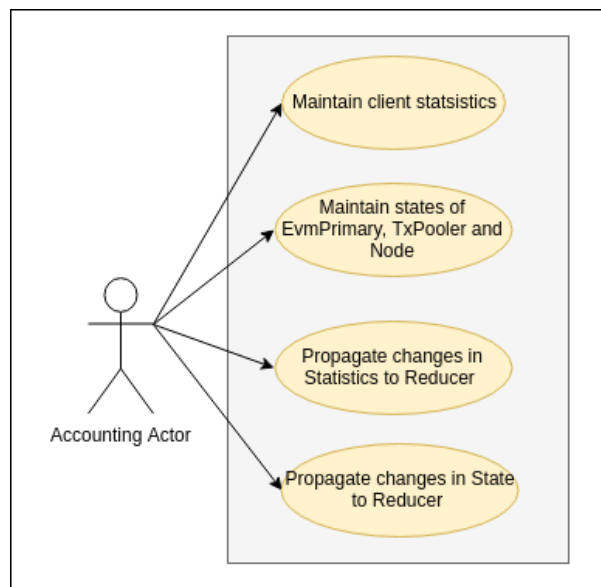


Figure 4.8: Use case diagram for Accounting Actor

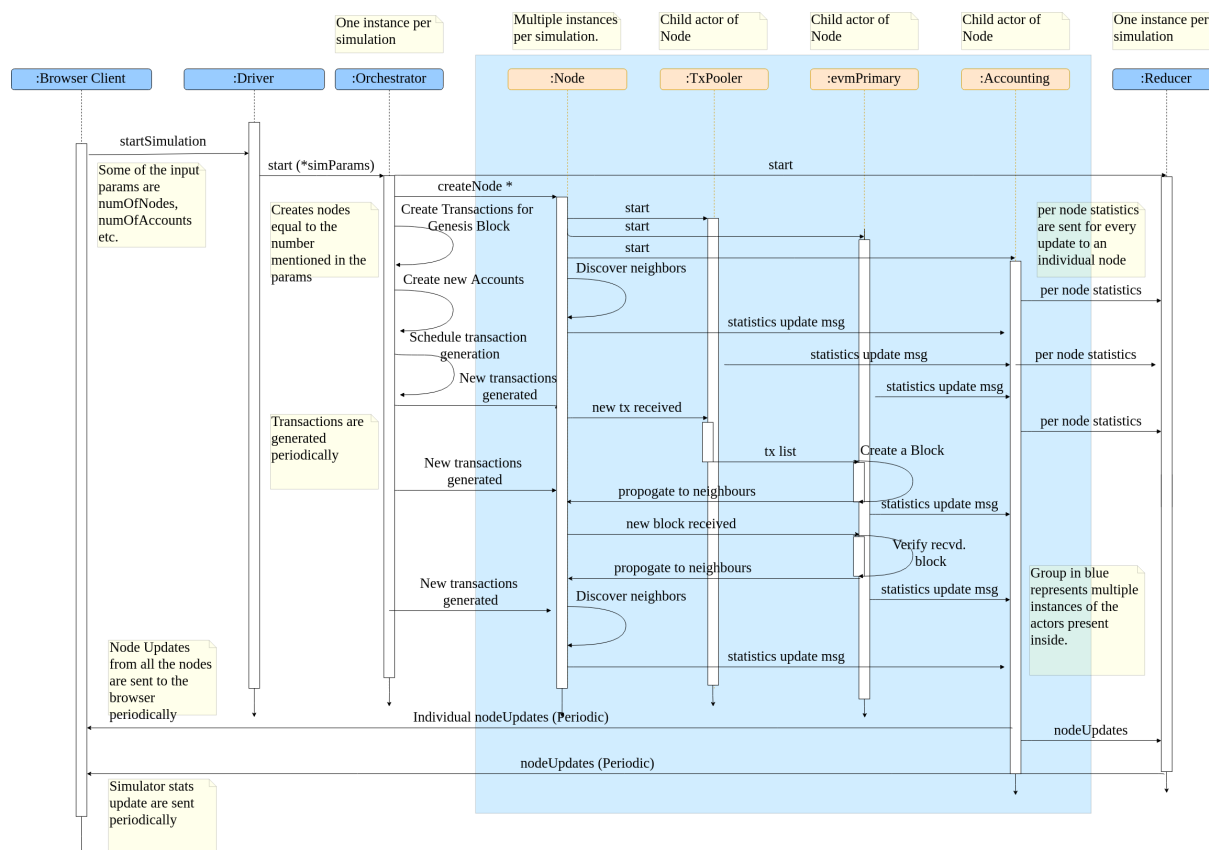


Figure 4.9: Working of eVIBES- Behaviour Diagram

settings) and initializes them with the genesis block, consequently forming the first clients in the simulated blockchain. Once the BOOT NODES are initialized the Orchestrator creates a random adjacency map simulating the connections graph between the BOOT NODES.

As seen in the Figure 4.9 simulation initialization involves scheduling transaction generation with the given transaction rate and batch size. Creation of accounts is a part of simulation initialization as well. The number of accounts is a part of the input settings. Also, periodic Discover messages that form the basis of the neighbor discovery algorithm are scheduled in the initialization phase.

Node Discovery

Figure 4.10 shows the steps involved in Node discovery. Node discovery is not applicable to the BOOT NODES in the simulation. It is assumed that the initial adjacency connections stay until the simulation stops. Node actor is responsible for maintaining a routing table

that lists all the reachable nodes from the client.

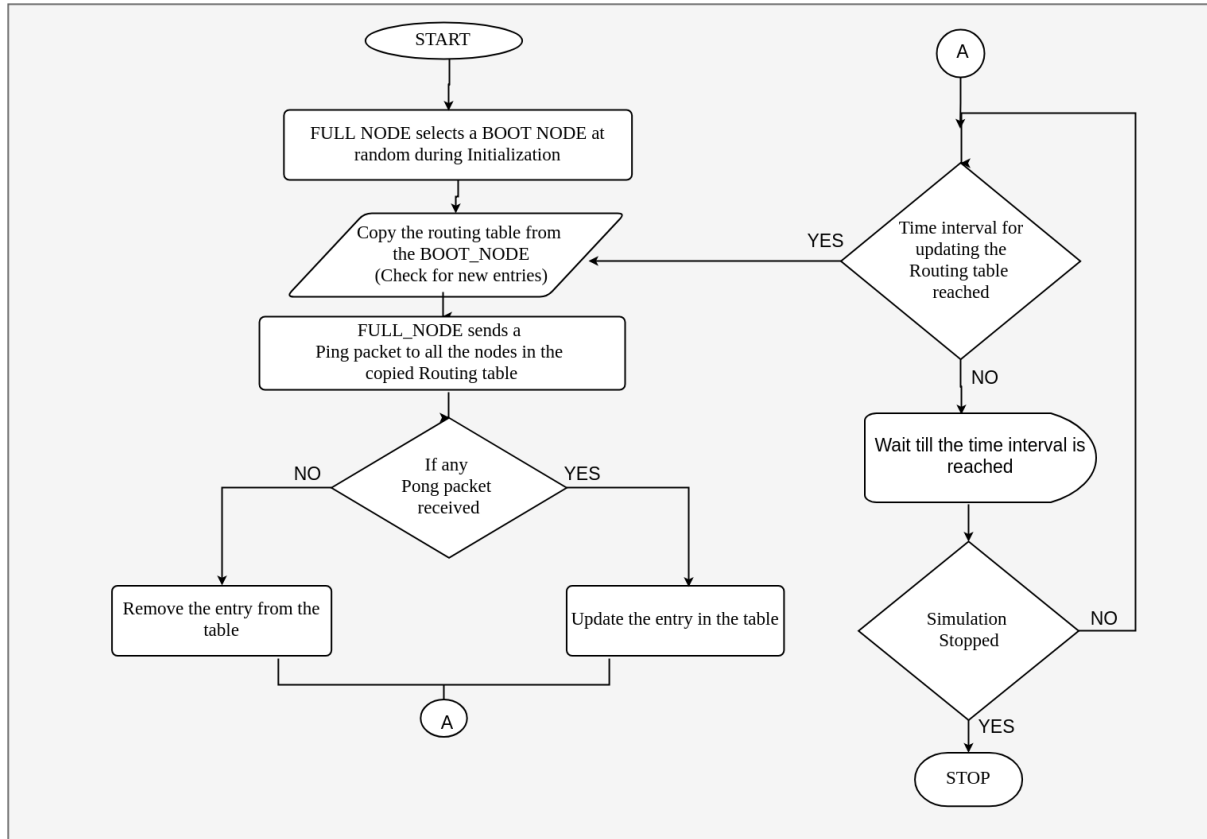


Figure 4.10: Node Discovery Flowchart

Transaction Executuon

This is a sub-part of Block creation and entails a two step process, Transaction verification and execution. The simulated algorithm for transaction execution tries to mimic the actual execution algorithm implemented in the Ethereum clients. However some liberty is taken to fit in changes towards the simulation. A Flowchart in the Figure 4.11 shows the steps involved in the transaction verification and trasaction execution.

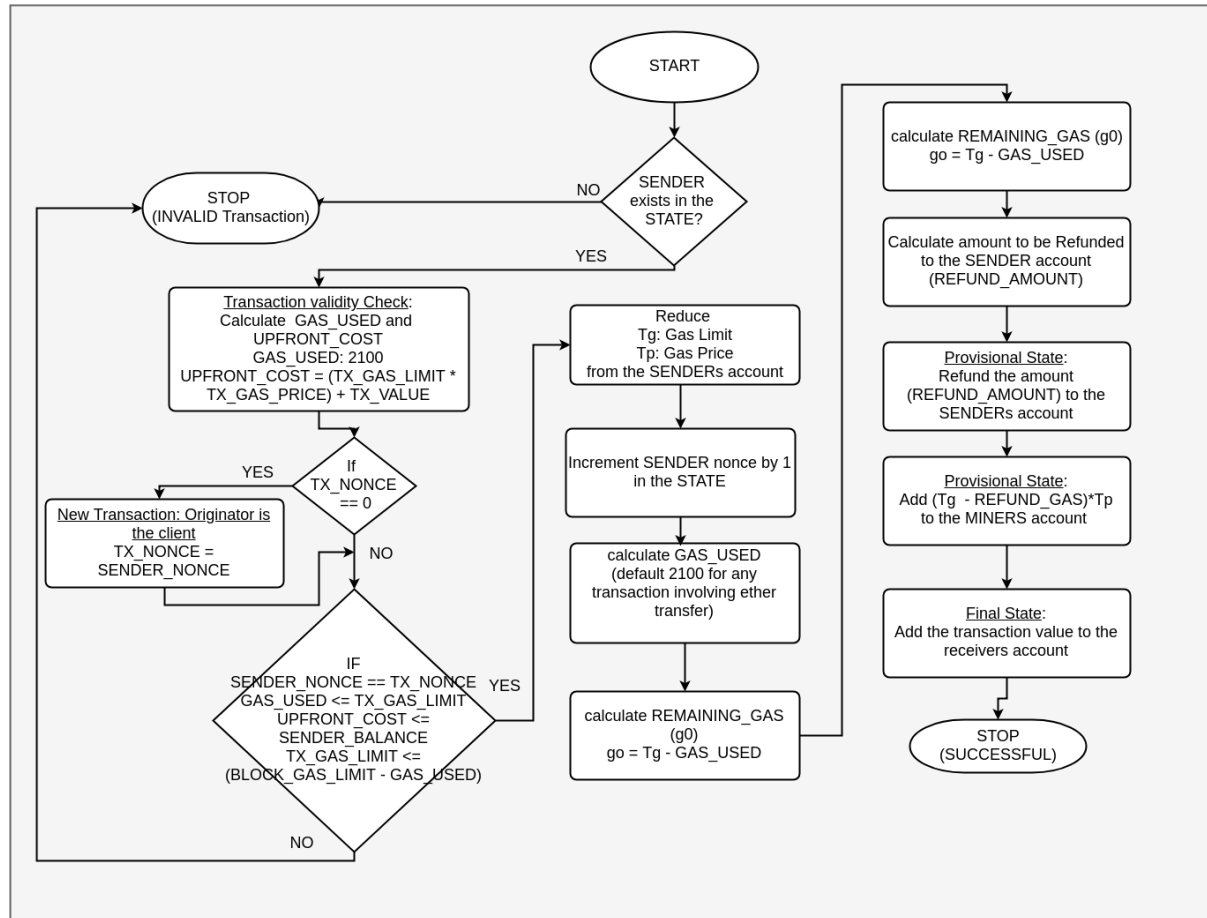


Figure 4.11: Transaction Execution Flowchart

Journey of a Transaction

Every transaction originates in the Orchestrator where it is assigned a source client and a random sender and receiver address. For now, the value of each transaction is static, but in the future, this can be changed. Once created it is sent to the source client. Source client then acts as the originator of the transaction. The transaction is then placed in the transaction pool of the originator client. Once the transaction is selected for block creation, it is bundled with other transactions and passed on to the EVM. For block creation, every transaction is first verified. If well-formed, the transaction is considered for execution. Once all the transactions in the list are executed a Block is formed, or in Blockchain terms, a Block is mined. This mined block is then propagated to its neighbors. The nonce value in the originating transactions is updated, and the transaction is propagated to the neighboring nodes.

A similar process is followed by clients that are not originator clients when they receive a

transaction.

4.6 Prerequisites

Before diving into the implementation details, this section gives an overview of the underlying technologies used to build the eVIBES system.

4.6.1 Akka Actors

In this thesis, the akka actor framework is used to model Actors. Akka is an open source library that is built for designing scalable and resilient systems [34].

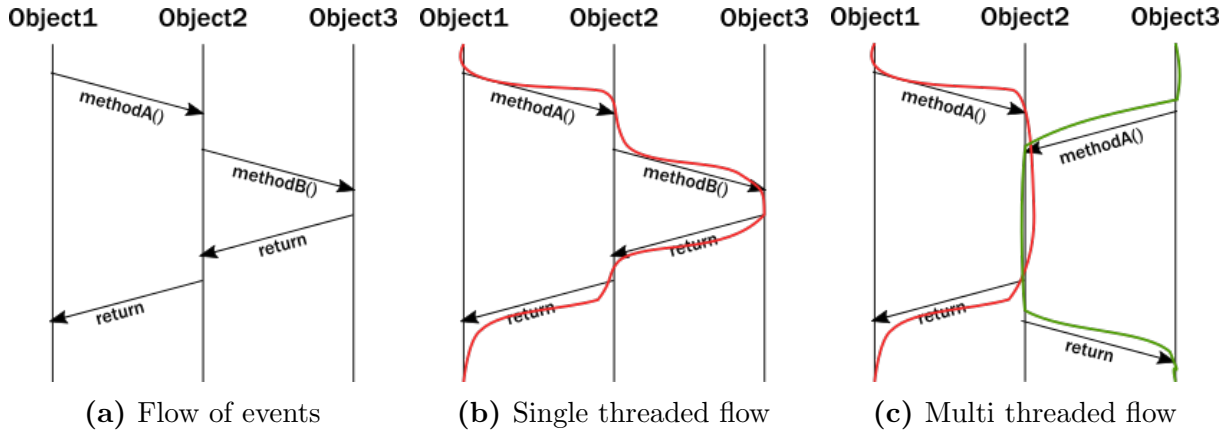


Figure 4.12: Call stack of a single and multithreaded implementation [5]

In an object-oriented programming model, where every modeled entity is an object, each object encapsulates its state and is hidden from the outside world. All the objects offer methods to access their state and prevent direct state change. These curated set of methods ensure that state of the system changes by a few methods maintaining the system invariant. Figure 4.12a shows the method call from one object to other. We see how an object has to call a method to access the state of other objects. Figure 4.12a hides how the entire flow of events unfolds. In order to understand the flow of events Figure 4.12b shows how the message exchange happens in a single threaded execution environment. All the invariants are maintained as the sequence is run by a single thread. When the same sequence is executed in a multi-threaded environment as shown in Figure 4.12c we see an

overlap between sequences executed by two threads. Encapsulation does not provide a way to handle this scenario and usually arbitrarily resolves this issue without the guarantee of preserving the invariants. A common approach to solve this problem is to use threads and locks that control the access to common sections of the code. This added complexity for synchronization increases the cost of execution.

Actors provide a way to achieve concurrency in this system without the use of operating system primitives like locks, threads or critical section. As discussed previously each process is modeled as an actor with its state. Actors share their state using messages. No entity has access to the internal state of the Actors. As seen in Figure 4.13, each actor shares its state using messages. At any point in time, an Actor can only process one message thus preserving the system invariants without additional synchronization mechanisms.

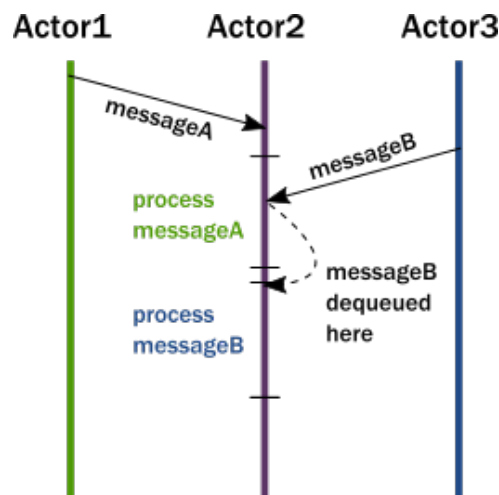


Figure 4.13: Message flow between akka actors [6]

Actor Supervision

As the actors do not share the same call stack there are different ways in which the errors are handled [7].

- When delegated task on the target actor fails : In this case, an error message is sent to the source explaining the failure.
- Service itself encounters a fault: Actors are arranged in a hierarchy as shown in the 4.14. Whenever a child actor fails, the parent actor for that child is notified and this failure is propagated till it reaches the root node or a node which handles the error. A parent node can then decide the actions to be taken on the error condition.

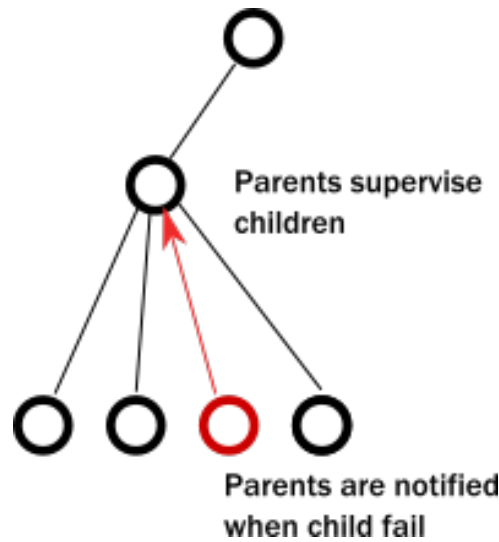


Figure 4.14: Actor supervision [7]

4.6.2 Akka Streams and Akka HTTP

Akka Streams are an implementation of the Reactive streams specification. Reactive streams provide a standard for asynchronous stream processing with back pressure [35]. In an asynchronous connection, the speed of data generation and data consumption might vary. If both these speeds are same, there is no problem to handle. Also, in the case where the Producer is slow, and the consumer is fast, asynchronous communication can work just fine. A problem arises when the speed of the Producer is more than the speed of the consumer. In such cases, the consumer gets overworked and sometimes drops packets or even crashes. To handle these cases, the concept of backpressure in streams can prove useful.

Using backpressure, the consumer can specify the number of units that it can handle, the producer will only send the specified number of units as mentioned by the consumer, thereby creating a backpressure.

In our implementation, we use three akka streams using backpressure to transfer the simulation data from the server to the web browser client.

Server Sent Events

eVIBES uses Server Sent Events as a way to propagate events/data from the simulation to the web browser. As stated in the akka documentation, Server-Sent Events (SSE) is a lightweight protocol that can be used for pushing notifications from an HTTP server to

a client [36]. In contrast to WebSockets where the data flow is bidirectional, the direction of data in SSE is unidirectional.

4.7 Implementation

Equipped with the knowledge of the design and the prerequisites this section discusses the implementation details of the eVIBES system.

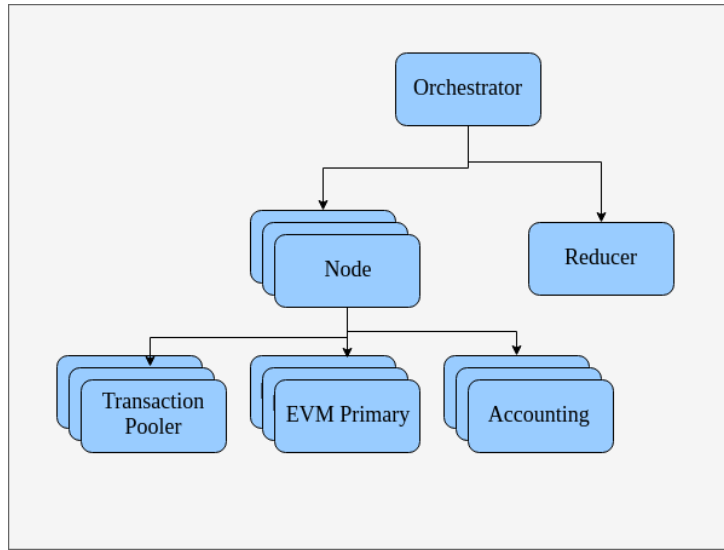


Figure 4.15: Actor Hierarchy in eVIBES

4.7.1 Node Hierarchy

Following the feature of actor supervision, Actors in eVIBES follow a certain node hierarchy. As shown in the Figure: 4.15 Orchestrator is the parent actor for all the actors. Each simulation has one Orchestrator actor which supervises all the actors in the simulation. Orchestrator creates one Reducer actor and multiple Node actors. These Node actors then create multiple actors as seen in the Figure: 4.15.

4.7.2 Configuration Parameters

The user can configure the following simulation parameters before the start of the simulation: Number of nodes, number of accounts, the number of transactions, the rate

of transaction generation, the range of gas limit for miner nodes. Besides, the user can override default values for the genesis block before the execution starts. The simulation outputs the following metrics: the total execution time, the total number of transactions processed, the throughput (transactions per second), the block propagation delay, the cost per transaction, a log of all transactions. Other metrics such as the change in difficulty over time, the change in block duration over time, the change in gas limit over time are presented at the end of the simulation. The system allows the inputs to be changed while the simulation is running. eVIBES allows manipulation of parameters like the transaction rate, the miner gas limit, and the network latency during the simulation execution. This control helps with studying the effects of these parameters on the system. eVIBES models the valid block generations (PoW in Ethereum network), as a probability score. This feature leads to multiple nodes generating valid blocks around the same time for similar transactions, as is the case in Ethereum blockchain. eVIBES stores these orphan blocks as Uncle/Ommers blocks in the blockchain and rewards the miners of those blocks based on the GHOST protocol [27].

4.7.3 Dashboard Statistics

eVIBES offers a web interface for interacting with the simulation which consists of a setup screen and a simulation execution screen. The setup screen allows the user to configure the simulation parameters, such as the number of nodes, the transaction rate, the number of accounts, the number of neighbors per node. The user can also add some settings that become a part of the genesis block in the simulation. These settings include the difficulty, the block gas limit, and the gas price. Once all the required settings provided, the user can start the simulation which then redirects the user to the simulation execution screen. The simulation screen offers a complete view of the simulation outcome in real-time. The outcome can be categorized into three categories:

- Average statistics of all nodes in the simulation
- Average statistics plotted over time
- Statistics for Individual nodes

Average statistics of the simulation

These statistics include, but not limited to, the total number of blocks created, the total active nodes in the simulation, the difficulty. Other statistics which are controlled by the

user, such as the transaction generation rate, the gas limit, are also displayed.

Average statistics plotted over time

To better understand the dynamic properties of the system, it is necessary to study the changes in the system over time. The system offers a feature for studying metrics over time and across simulations, such as the time required for block creation, the gas limit, the number of transactions.

Individual Statistics of Nodes in the Simulation

Apart from the average statistics of the entire network, eVIBES generates individual statistics for all the nodes as well.

4.7.4 Simulation Initialization

Initialization of the simulation starts with the start of the Orchestrator actor. The code snippet in Listing: 4.1 shows the steps involved in the initialization. The code creates two types of nodes, BOOT NODES, and FULL NODES. As explained earlier, BOOT NODES are a particular type of FULL NODE that are initialized with a fixed neighbor map, and their local copy of blockchain is initialized with a genesis block. Initialization of FULL NODES involves selecting a BOOT NODE at random and updating the local copy of blockchain from the version present in the BOOT NODE. A FULL NODE also fetches the routing table from the BOOT NODE.

```
/* startSimulation is a function inside the Orchestrator actor
 * responsible for creation of all the nodes and accounts in the
 * simulation. The function also schedules transaction execution.
 * */
def startSimulation(settings: Setting.type) = {
  // Create a reducer actor
  val reducer = context.system.actorOf(Props(
    new Reducer(globalStatsQueue, localStatsQueue)), "reducer")

  //Create and initialize BOOT NODES in the simulation
  val bootNodes = initializeSimulator(settings, reducer)
```



```

// Create and initialize FULL NODES
val allNodes = startNodes(settings.nodesNum, bootNodes._1,
                           reducer, eventQueue, settings, bootNodes._1)

// Create accounts for miner nodes. Accounts from both BOOT NODE
// and FULL NODE can be a miner
val minerAccounts = createMinerAccounts(bootNodes._2 ++ allNodes._2,
                                         allNodes._1)

//Create accounts equal to the number of accounts in the input
val accounts = createAccountsInNode(settings.accountsNum, allNodes._1)

// Using the accounts and the nodes generated above schedule transaction
// generation
scheduleTxCreation(settings, accounts ++ minerAccounts, allNodes._1)
}

```

Listing 4.1: Simulation initialization code

4.7.5 Boot Node Selection for Client Initialization

Failure scenarios for nodes are not simulated in eVIBES. Hence it becomes difficult to mimic the workings of the ethereum clients. This Uncertainty of the network failures is introduced in the simulation by adding randomness to the tasks that expect some degree of network failures. In the actual ethereum blockchain, when a client is initialized, it pings a hardcoded list of nodes a.k.a. BOOT NODES that would offer a copy of Blockchain to the client. In some cases, not all the BOOT NODES would be available, and hence the client would be initialized with the BOOT NODE that is available at that instance. In eVIBES, we use the code mentioned in the Listing 4.2 to select a BOOT NODE at random for initialization.

```

/* startNode is responsible for creation of FULL NODES in the simulation
* */
def startNodes(noOfNodes: Int,
               nodesMap: mutable.HashMap[String, ActorRef],

```

```

reducer: ActorRef,
eventQueue: SourceQueueWithComplete[EventJson],
setting: Setting.type,
bootNodeMap: mutable.HashMap[String, ActorRef]):
(mutable.HashMap[String, ActorRef], ListBuffer[Account]) = {

var accList = new ListBuffer[Account]
for (i <- 0 until noOfNodes) {
  //Create nodes equal to the number specified in the input params
  val nodetp = createNode("FULLNODE", reducer, eventQueue, setting)
  nodesMap.put(nodetp._1, nodetp._2)
  accList += nodetp._3
  val bootNodeKeys = bootNodeMap.keys.toList
  //Pick one BOOT NODE at random from a list of BOOT NODES
  val indx = Random.nextInt(bootNodeKeys.length)
  //Initialize the FULL NODE from the BOOT NODE
  nodetp._2 ! InitializeFullNode(
    bootNodeMap.get(bootNodeKeys(indx)).get)
}
new Tuple2[mutable.HashMap[String, ActorRef],
  ListBuffer[Account]](nodesMap, accList)
}

```

Listing 4.2: Code snippet for creation of FULL NODEs

4.7.6 Neighbourhood Creation in Boot Nodes

Neighborhood for BOOT NODES is created by randomly selecting a value from the range of minimum allowed neighbors and maximum allowed neighbors, which is an input parameter. A neighborhood in BOOT NODES is static and does not change till the simulation stops. Adding dynamic properties to the BOOT NODE neighborhood can be considered as a part of the future work.

4.7.7 Proof of Work Simulation

One of the most costly process in the Ethereum client is the Proof of Work computation. As the PoW computation was of no purpose to the simulation we instead replaced it by a probability function that gives a success probability of 0.3. Once a block is generated, there is a 30% chance that it will be added to the blockchain. The probability function is applied in the TxPooler Actor when the combined gasLimit of set of transactions in the pool reaches the block gas limit.

4.7.8 Transaction Generation

```

/*scheduleTxCreation is responsible to schedule periodic generation
 * of transactions.
 * */
def scheduleTxCreation(settings: Setting.type,
                      accounts: ListBuffer[Account],
                      nodeMap: mutable.HashMap[String, ActorRef]):
  Cancellable = {
    var nodeKeys = nodeMap.keys.toList
    // System scheduler to schedule transaction generation periodically
    context.system.scheduler.schedule(25 second, 10 second,
      new Runnable {
        override def run(): Unit = {
          // Transactions are generated here based on the batch size
          val txList = createTransactions(settings.txBatch, accounts)
          for (tx <- txList) {
            // Transaction is randomly assigned to a originator node
            val node = nodeMap.get(nodeKeys
              (Random.nextInt(nodeKeys.length))).get
            // Transaction propogation to the originator node
            node ! NewTx(tx)
          }
        }
      })
  }

```

Listing 4.3: Code snippet for Transaction generation

As shown in the code snippet in Listing: 4.3, generation of a transaction is based on the transaction batch size. The interval between two transaction generation cycles is currently hardcoded but can, later on, be converted as an input parameter.

4.7.9 Maintaining Account states

In Ethereum blockchain, each client keeps track of the account states by updating a global state trie whenever a new block is added to the local copy of the blockchain. Blockchain can be viewed as a tree with Genesis block as a root node and multiple versions of blockchains being the children nodes in the tree. Each block is a state transition function that changes the account states. It thus becomes mandatory to keep a copy of updated accounts for each block in the blockchain. State trie in the actual ethereum implementation resolves this requirement. In eVIBES we create an entry in the database with the key as a combination of client id, block id and value as a list of account addresses of all the updated accounts. This arrangement in eVIBES keeps track of all the account state changes of each block in the blockchain for every client.

4.7.10 Tracking Executed Transactions

Similar to the Account states, each block in the ethereum client stores a transaction root which is the hash of the root of the transaction trie. To avoid double execution of a transaction, every transaction cross verified against the transaction trie. eVIBES develops a mechanism that saves all the transactions executed as a key, value pair in the database. key being the client id, block id and the value is the list of all the executed transactions during the mining.

4.7.11 Priority Inbox

While simulating a message-oriented system, there comes a need to prioritize some messages over others. Say simulation control messages hold a higher priority than transaction propagation message. To handle this requirement eVIBES offers a priority inbox for each of the actors. This way a priority can be assigned to any message that is exchanged in the eVIBES system.

4.7.12 Tracking the Longest Subtree in Blockchain

In the ethereum blockchain when a new block is generated by a client, it is attached to the longest subtree in the local blockchain copy [27]. This is done greedily and is also called as GHOST protocol. In eVIBES a data structure known as GHOST_DepthSet is used to implement the GHOST protocol. GHOST_DepthSet keeps track of all the longest subtrees in the blockchain. The data structure is updated after every new block is added to the blockchain. Selection of the longest subtree in the blockchain for adding generated block is made using GHOST_DepthSet in eVIBES.

4.7.13 State Management of the Simulated Client

Once started, each actor that is the part of the simulated ethereum client, transitions from one state to other. In the below section we briefly discuss all the state transition of these actors.

Node Actor

As Node actor is mainly responsible for receiving and propogating messages, it usually stays in the ACCEPT-CONN state. However after every X seconds it transitions to a DISCOVER state for updating the routing table. Figure 4.16 shows all the transitions of the Node actor.

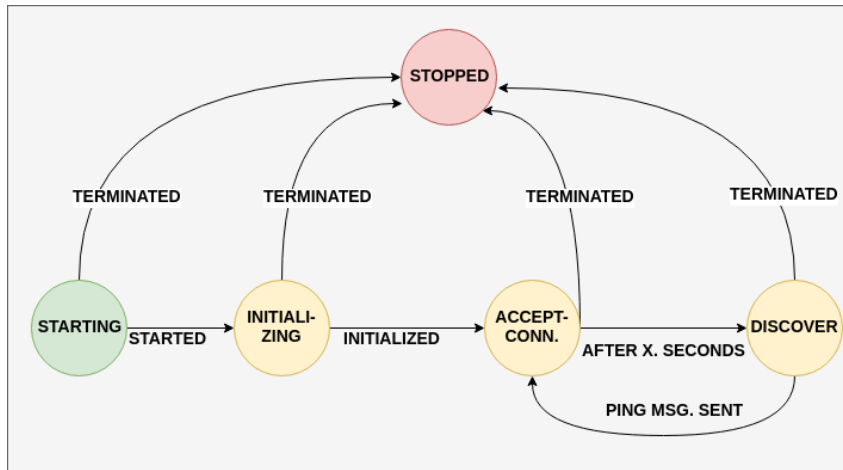


Figure 4.16: State transition of the Node actor

Transaction Pooler Actor

After initializing transaction pooler is mostly stays in ACCEPT-TX but if the transaction pool is full it transitions to REJECT-TX. Detailed view of all the transitions of the Transaction pooler actor can be seen in Figure 4.18

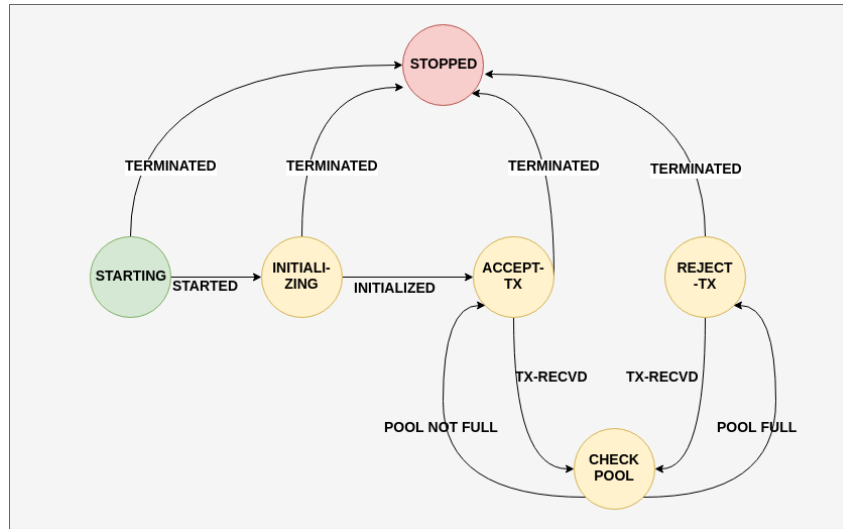


Figure 4.17: State transition of the Transaction pooler actor

EVM Primary Actor

Detailed view of the EVM Primary actor transitions can be seen in the Figure 4.18.

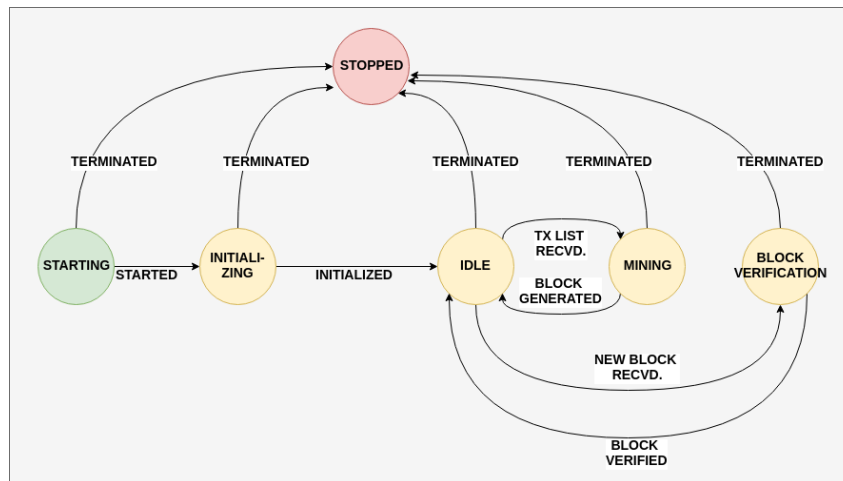


Figure 4.18: State transition of the EVM Primary actor

4.8 Technology

This section discusses the tools and technologies used for building the eVIBES simulator.

4.8.1 Backend

Inspired by the work in VIBES [4] and the proven advantages of using Scala and Akka framework for a similar use case, we decided to follow the same technology stack for developing the eVIBES system. Apart from that Scala as a JVM language offers functional primitives and less verbose programming primitives that help in building a robust, readable and scalable application.

The Akka Framework [34] is used to model the Actor system. The thesis also used parts of the Akka Streams[37] and Akka HTTP [38] for implementing the Server side events.

Akka framework offers the concurrency model that is required to simulate a large number of entities. We designed the system to run on a single machine, but in the future, if the need arises to run it on multiple machines, we would be able to do that efficiently because of the support of the Akka framework.

Storing state of each client is an integral part of the eVIBES thesis. For large simulations, the simulator would generate data at very high speeds. In considerations to these high speeds, we were looking for a database with high throughput, less boilerplate code and has a high data ingestion rate. Offering the highest throughput and a high ingestion rate, we decided to use RedisDB [33] as our database. RedisDB also offers a lot of off the shelf implementations of features such as Analytics, Inbuilt data-structures. These features can be helpful in the future for extending the working of the simulator without changing a lot of the code. Other contenders for the database were the LevelDB [39] and the RocksDB [40], but both lacked a good connector for Scala and hence were discarded.

4.8.2 Frontend

The thesis aimed at building a reactive dashboard for displaying the statistics at near real-time. The simulation generates large volumes of data per minute. Also, we had multiple frontend components that need near real-time updates. Updating the entire DOM everytime data for a single change in a component would have been a very costly process. The virtual-dom update approach offered by the ReactJS proved to be of immense help.

With react [41], updating the changed component without updating the entire DOM helped us gain a lot of speed in displaying the data.

For our requirements, using a Frontend framework would have proved a costly affair as well. Hence we dropped the idea of using Angular [42] for our choice of frontend technology.

To follow a single UI design paradigm, we used the Blueprint.js [43] library for building the UI components. The use of the Prebuilt UI components helped us maintain a consistent look and feel on the dashboard. Also, using the prebuilt components ensured that we would not spend a lot of time creating basic components.

Chapter 5

Evaluation

In this chapter, we evaluate the working of the eVIBES simulator using the parameters of Correctness, Scalability, Extensibility, Flexibility and Powerful visuals. All the tests have been conducted on an HP Probook 640 with an Intel® Core™ i7-7500U CPU @ 2.70GHz 4 processors, 7.7GiB DDR3 RAM, Intel® HD Graphics 620 (Kaby Lake GT2) Graphics card, Disk space of 70GB(SSD) and Ubuntu 18.4.1 LTS - 64-bit operating system.

5.1 Correctness

Correctness of the simulation is its ability to behave like the system which is simulated, in this case, Ethereum Blockchain. In this subsection, we consider all the fundamental aspects, algorithms in the ethereum blockchain client considering the Yellow Paper as the official documentation [12] and validate the simulation algorithms against the formal requirements.

5.1.1 Transaction Execution

Transaction execution algorithm forms a key part in the ethereum blockchain simulation. A faulty transaction execution algorithm can result in incorrect block creation and incorrect account states. As explained in figure 4.11, transaction execution is a series of steps that start with the accounts of Sender, Receiver, Miner and a Transaction and update the states of these accounts. An update which is synchronous to the transaction parameters considers the transaction execution as successful. In the evaluation, we consider the transactions generated at different points in time over the execution period

of the simulation and try to empirically compare the expected state changes with the changes made by the simulation.

In eVIBES, all the transactions carry the same value(5 units) and same gas limit; we will not mention the contents of each transaction.

States	Accounts	TX-1	TX-2	TX-3	TX-4
Before Tx Execution	<i>Sender</i>	2200000.0	2178995.0	2157990.0	2220995.0
	<i>Receiver</i>	2200000.0	2200000.0	2200000.0	2200000.0
	<i>Miner</i>	2284000.0	2221000.0	2242000.0	2199995.0
After Tx Execution : eVIBES	<i>Sender</i>	2178995.0	2157990.0	2136985.0	2220990.0
	<i>Receiver</i>	2200005.0	2200005.0	2200005.0	2200005.0
	<i>Miner</i>	2305000.0	2242000.0	2263000.0	2220995.0
After Tx Execution : Manual	<i>Sender</i>	2178995.0	2157990.0	2136985.0	2220990.0
	<i>Receiver</i>	2200005.0	2200005.0	2200005.0	2200005.0
	<i>Miner</i>	2305000.0	2242000.0	2263000.0	2220995.0

Table 5.1: Transaction execution evaluation

Table 5.1 shows four transactions picked up at random and tries to compare the expected system state and actual system states after transaction execution. As seen from the above table, all the values in the eVIBES state are same as the manually computed values. To understand what these numbers mean, let us consider values for TX-1. Using the values of Tx-GasLimit = 22000 ; Tx-GasPrice = 1.0 ; Remaining Gas = 1000 ; Refund Amount = 1000 and Miner Reward = 21000 and the algorithm from Figure 4.11 the correctness of the transaction execution implementation can be verified easily.

5.1.2 Block Verification

Blocks in the Ethereum blockchain can be generated either by mining or by receiving blocks from other nodes and verifying them. Once verified the blocks are added to the blockchain. Similarly, in the simulation, the blocks are generated using mining. A major difference between mined blocks and the verified blocks is the parent node selection. To verify the correctness of the algorithms, we compare blocks in random nodes in the simulator and verify their computed properties with expected properties. Properties under consideration are Difficulty, Block gas limit, and gas used. Later on, we also verify the correctness of the GHOST subtree selection algorithm.

Generated Blocks

Generated block or mined block is a new block, not present in any versions of the blockchain. To validate the correctness of the generated block we pick three generated blocks at random from different nodes and compare their contents with the expected contents.

	Accounts	Block-1	Block-2	Block-3
Parent Block	<i>Difficulty</i>	17179869184	17188257792	17221853204
	<i>Block gas limit</i>	63000.0	62938.477	62692.98
	<i>Gas used</i>	0.0	21000.0	21000.0
Child Block : eVIBES	<i>Difficulty</i>	17188257792	17196650496	17230262312
	<i>Block gas limit</i>	62938.477	62877.01	62631.758
	<i>Gas used</i>	21000.0	21000.0	21000.0
Child Block : Manual	<i>Difficulty</i>	17188257792	17196650496	17230262312
	<i>Block gas limit</i>	62938.477	62877.01	62631.758
	<i>Gas used</i>	21000.0	21000.0	21000.0

Table 5.2: Block Generation evaluation

All the values generated by the eVIBES are same as the values computed manually thus verifying the correctness of the block generation algorithm.

Propagated Blocks

As explained earlier, Propagated blocks are the ones that are mined in a different node and are sent to other nodes for verification. A node in the ethereum simulator checks for the valid block components and then executes all the transaction to update its state. As none of the parameters/properties under consideration are changed, we verify via a code walkthrough that none of the block properties are changed during the state update.

Parent Node Selection

For generated blocks as well as for propagated blocks properties of the blocks are verified against the properties of their parent blocks. Selection of a valid and correct parent block thus becomes an important aspect in Block verification. As explained in the previous section, the eVIBES uses an algorithm similar to GHOST [27] for subtree selection. Verification of the algorithm involved an empirical analysis of five blocks at random and looking at all the subtrees that were available during the block creation and the depth of the selected subtree. The depth of the selected subtree should be maximum.

Empirical analysis showed that every time a subtree with the maximum depth was selected. A code walkthrough verifies that the GHOST.DepthSet data structure maintains a list of nodes that have the maximum depth. This way eVIBES ensures greedy heaviest subtree selection at all points during the simulation execution.

5.1.3 Blockchain Verification

As explained in the previous chapter, the blockchain is a decentralized system and follows a logical centralization. To verify the logical centralization in the eVIBES simulator, we considered few randomly selected blocks and verified if those blocks become a part of the blockchain in other nodes. Our empirical analysis proved that blocks generated at one node are propagated to multiple nodes and are successfully verified.

5.1.4 Conclusion

Verification from above sub-sections asserts the Correctness of the implemented algorithms and the working of the eVIBES can be validated against the working of an Ethereum Client.

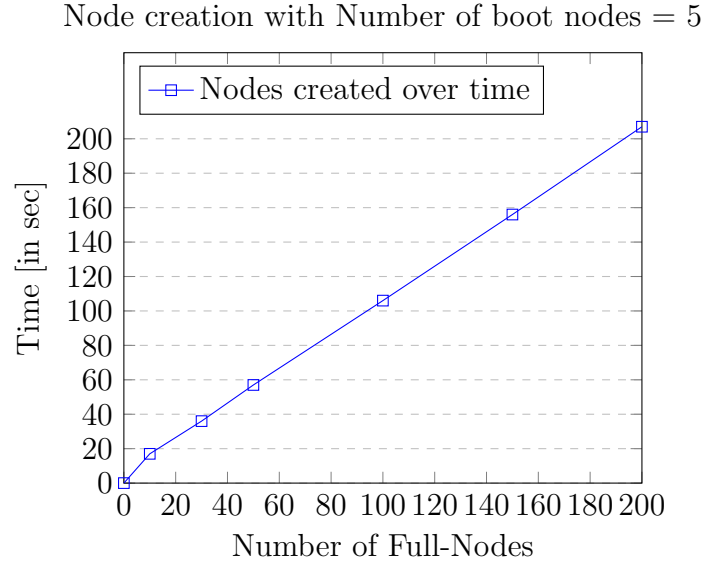


Figure 5.1: Verifying the scalability of the simulator-varying number of nodes with 5 BOOT NODES

5.2 Scalability

Scalability is the ability of the simulator to maintain the performance with increasing work, in the case of eVIBES, increasing number of Nodes, Neighbours, Transactions and Accounts. All these parameters directly affect the working of eVIBES.

5.2.1 Node Creation

To test the scalability of the eVIBES when the number of nodes are increased, we calculate the time taken for initialization of nodes (varying number) keeping number of neighbours per node between 2-5 and testing the scalability with number of boot nodes =5 and 10

eVIBES demonstrates its ability to scale linearly as the number of nodes increases. The Figure 5.1 and 5.1 suggest that creation of boot nodes is a linear process and number of boot nodes does not affect the scalability of the system.

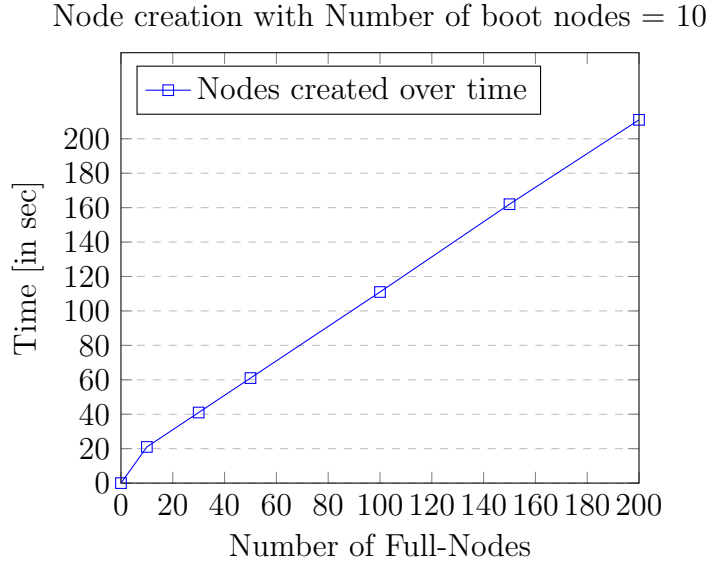


Figure 5.2: Verifying the scalability of the simulator-varying number of nodes with 10 BOOTNODES

5.2.2 Varying Number of Neighbors

Every node in the eVIBES simulator connects to N number of other nodes. We analyzed the effects of varying the number of neighbors to a node. As explained in the previous chapter, number of neighbors for a node is selected from a range of minimum and maximum number of allowed neighbors. This is an input parameter and is passed before the simulation starts. We test the scalability for three parameters settings 1-5, 5-10, 10-15 and 15-20 min-max allowed neighbors. We keep the number of nodes fixed to 100 and number of boot nodes to 3.

As seen in Figure 5.3 Number of Neighbors does not affect the scalability of the simulator.

5.2.3 Varying Number of Accounts

Accounts in eVIBES are created during initialization. As the account creation takes place on all the nodes, the number of accounts affects the scalability of the system. For a constant number of nodes 50 and fixed neighbors 1-5 and three boot nodes we will study the effects of account creation on the scalability of the system. Also, the genesis block contains one account only.

As seen from Figure 5.4 as the number of accounts increase, the time taken to initialize the simulation increases as well. Although the increase can be considered as linear or even

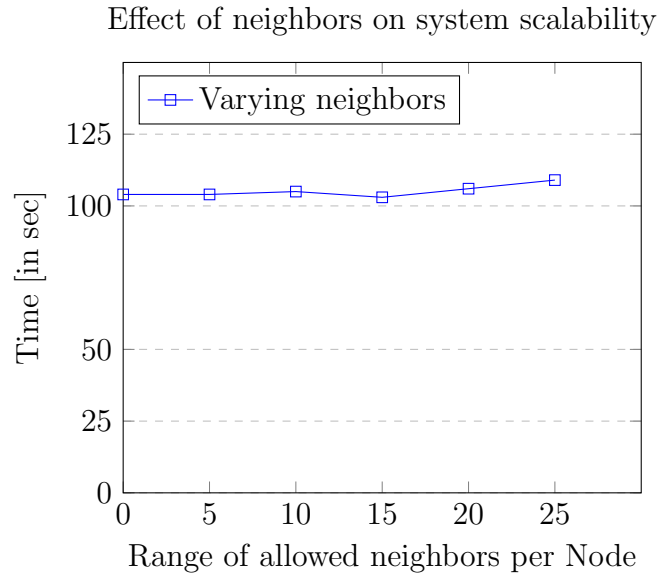


Figure 5.3: Verifying the scalability of the simulator-varying number of neighbors.

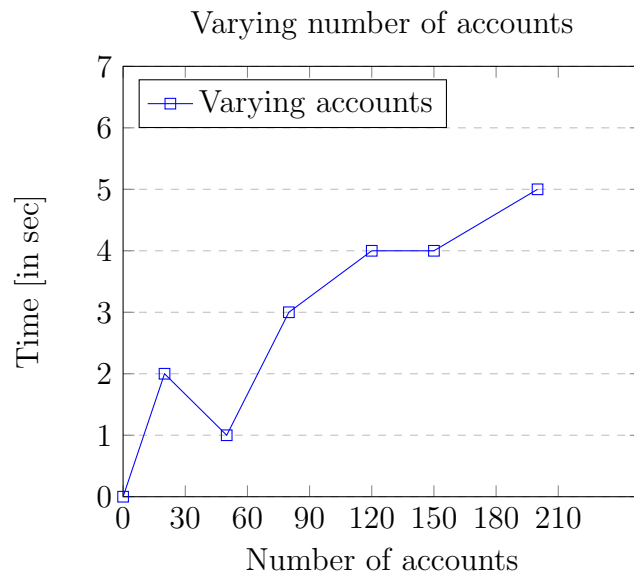


Figure 5.4: Verifying the scalability of the simulator-varying number of accounts.

constant as the time difference is extremely small.

5.2.4 Conclusion

Above study shows the performance of the eVIBES systems under different configurations. We observed that the system scales linearly with increasing number of nodes. The number of neighbors per node does not affect the scalability of the eVIBES system. A varying number of accounts also do not affect the scalability of the system.

5.3 Flexibility

Flexibility in the context of eVIBES is its ability to simulate blockchain with multiple configurations. eVIBES provides the user with a set of input parameters that can create an Ethereum blockchain network in different combinations. The input parameters that control the ethereum simulation are,

- Number of Boot nodes
- Number of Nodes
- Number of Accounts
- Input for transaction generation
- Neighbors per Node
- A configurable genesis block

These parameters are fundamental to building any blockchain and variation in their values changes the network. In eVIBES for the simplicity, we have introduced randomness in many scenarios which is generated by hardcoded probability distributions as explained in the Implementation section. In the future, the eVIBES could be extended by offering the user the ability to set the probabilities of different measures which are currently hardcoded further increasing the flexibility of the system.

5.4 Extensibility

eVIBES has been designed and implemented keeping in mind the extensibility of the system. Separation of concerns and division of responsibilities are key aims that were

followed during the entire process of implementation. To demonstrate the extensibility of the system let us consider adding support for new features in the eVIBES,

5.4.1 Side chains

Side chains are blockchains that are separated from the main blockchain. This is done to reduce the cost of transaction processing on the main blockchain. A contract between two parties is made over the main blockchain and then a side chain is created. All the transactions are then processed on the side chain. The side chain is merged with the main blockchain once the contract between the parties is satisfied. In eVIBES, each of the Client stores a local copy of the Blockchain. With the local copy, it is then possible to create a side chain of few nodes and then, later on, merge the state changes in the main blockchain.

5.4.2 Contracts

Contracts in Ethereum are code blocks that reside in the account. A contract is executed when a transaction is directed to the account with a contract. In eVIBES, all the EVM related tasks are handled by the EVM-Primary actor. In order to support contract execution, we can add a new actor that would perform contract related tasks and return the results to the EVM-Primary actor. The account is modeled as a class in eVIBES and to support the contract feature we would need to add a new parameter to store contracts in Account which can be done easily.

5.4.3 Conclusion

The scale of the problem is reduced to a great extent with the help of akka. Also, eVIBES mimics the actual Ethereum blockchain closely which proves as an advantage while extending the system to add new features.

5.5 Powerful Visuals

The eVIBES simulator offers a reactive dashboard with statistics of individual nodes as well as average statistics of the overall blockchain. The dashboard offers time-series graphs

of all the average statistics of the blockchain generated over time. The dashboard offers a high-level view into the workings of the simulation as well as a detailed view of all the nodes in the simulation.

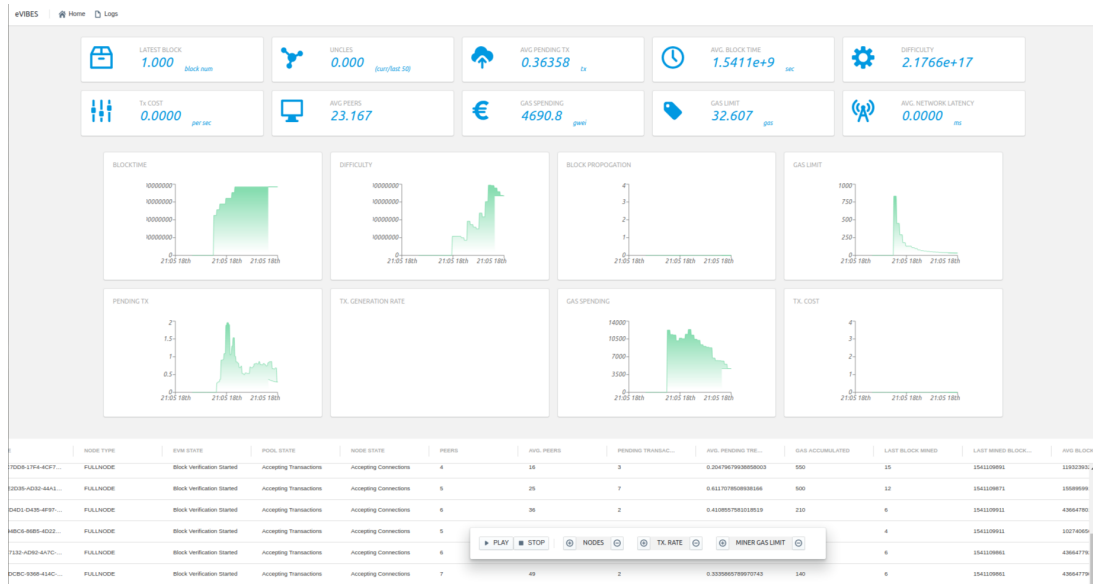


Figure 5.5: eVIBES Dashboard

Edit the configuration to start the eVIBES simulator

Boot node configuration

Number of Boot nodes (required)

 Initial number of nodes that will be initialized using the Genesis block

Transaction configuration

Number of Transaction (required)

 These are the total number of transactions that will be generated combined in all the batches

Rate of transactions (required)

 Rate at which transactions will be generated. i.e. Time for generating a batch of transactions

Post initialization configuration

Number of Nodes (required)

 These are the total number of nodes in the simulation

Min-Max Peers per Node (required)

 Number of peer connections per node

Genesis Block configuration

Parent Hash (default)

 Hash of the parent Block

Beneficiary (default)

 Beneficiary account address

Gas Limit (required)

 Initial Gas limit for genesis block

Number (required)

 Initial block number

Number of Initial Accounts (required)

 Initial number of Accounts that will be initialized using the Genesis block

Transactions in a batch (required)

 Based will be created based on the on the transaction rate.

Number of Accounts (required)

 Accounts that will be generated during the simulation

BlockGasLimit (required)

 Initial Block gas limit for the blockchain

Ommer Hash (default)

 Hash of the list of Ommer Blocks

Difficulty (required)

 Initial difficulty for genesis block

Gas Used (required)

 Initial Gas used for genesis block

Start the Simulation

Figure 5.6: eVIBES Welcome screen

Chapter 6

Summary

In the thesis, we have proposed, designed and implemented an event-driven, concurrent, message-oriented, broadcast-based configurable Ethereum simulation for simulating large-scale Ethereum networks. eVIBES is built on the foundations of domain driven design [17] and reactive manifesto [18] and closely simulates the behavior of the Ethereum blockchain. eVIBES implements transaction execution and block generation/mining in the simulation that mimics the working of other ethereum client implementations. With these features, eVIBES becomes one of the few Ethereum simulators designed for analyzing the behavior of the blockchain. To further facilitate a good understanding of the ethereum network, eVIBES offers a reactive dashboard with a detailed node level view as well as statistics related to the overall blockchain behavior.

6.1 Status

The thesis completes its goal of building a simulator that mimics the workings of an Ethereum blockchain network. The system built during the thesis is a basic system that enables behavior analysis of the ethereum blockchain network. The performed evaluation of eVIBES validates the behaviour of the implemented algorithms.

6.2 Conclusions

With the increasing interest in Ethereum, one would have assumed the existence of a useful tool ecosystem to reason about the behavior and workings of Ethereum blockchain.

However, our research proved otherwise. We found a few simulation tools, but none of them facilitated the study of the behavior of the ethereum blockchain network. Due to the lack of existing tools and related literature, the process of designing and developing eVIBES lead us to uncharted terrain. With the first version of the eVIBES verified and working we can now simulate ethereum-like blockchain networks and reason about the workings of the system under different configurations.

6.3 Future Work

In this section we discuss the areas that offer the possibility of future work.

6.3.1 Smart Contracts

Smart contracts from the basis of Ethereum blockchain. With all the pre-requisites in place, a natural extension to the eVIBES system is adding support for smart contracts.

6.3.2 Comparative Analysis Tools

Execution of eVIBES generates a lot of data about the network. Current implementation does not use all the generated data. A good extension of the system will be to collect all the data and build a comparative analysis tool similar to the one offered by etherchain [44]. A tool of this kind will further improve the visibility into the workings of the ethereum network.

6.3.3 Network Behavior Simulation

Currently, eVIBES does not simulate any of the network level protocols. Simulation of these protocols would improve the accuracy of the simulation and would increase the flexibility of the system.

6.3.4 Network Topology

In an actual Ethereum blockchain, the network topology keeps on changing due to the node failures, network partitions. Adding support for creating ethereum networks with a

particular topology can prove useful in studying the properties of Ethereum blockchain.

6.3.5 Detailed Analysis of a Node

eVIBES generates data for each node, including account state data and blockchain data. However, this data is not displayed on the dashboard. Current eVIBES dashboard does not offer a detailed view of these statistics. An extension to the eVIBES would be to create a detailed view of the data generated by individual nodes.

6.3.6 Extending the Simulator for Testing Side-Chains

As discussed in the evaluation section the current implementation of eVIBES each Node in the network stores a local copy of blockchain. This ability can be extended for adding side-chains support in the network.

Appendices

Appendix A

Appendix

A.1 Source code

The source code is available at <https://github.com/i13-msrg/evibes>

A.2 GHOST_DepthSet Implementation

```
/* GHOST_DepthSet is responsible for maintaining a list of blocks
 * in the blockchain that has the maximum depth.
 * GHOST_DepthSet is initialized after the node initialization and is
 * updated every time a new block is added to the blockchain.
 * */
class GHOST_DepthSet() {
    // HashMap of all the blocks with highest depth
    var depthSet = new mutable.HashSet[LightBlock]()
    var depth = 0
    var nodeMap = new mutable.HashMap[String, LightBlock]()

    // Responsible for adding new blocks to the depthSet
    def addLightBlock(lb: LightBlock) = {
        nodeMap.put(lb.blockId, lb)
        /*If the depth of the current block is greater than all
        * the blocks in depthSet, the depthSet is cleared and the
        * new block is added in the depth set.
        */
        if (lb.depth > depth) {
            depthSet.clear()
```

```

        depthSet.add(lb)
        depth = lb.depth
    }
    // For equal depth, add the new block to the depth set
    else if (lb.depth == depth) {depthSet.add(lb)}
}

// Responsible for retrieving leaf block that has the maximum depth.
def getLeafBlock() : LightBlock = {
    if (depthSet.size == 1) {
        val blk = depthSet.toList(0)
        nodeMap.getOrElse(blk.blockId, new LightBlock("0x0", "0x0", 0))
    }
    else if (depthSet.size == 0) {return new LightBlock("0x0", "0x0", 0)}
    else {
        // If the depthSet contains multiple blocks with same depth. Select
        // one block at random
        val blk = Random.shuffle(depthSet.toList).take(1)(0)
        nodeMap.getOrElse(blk.blockId, new LightBlock("0x0", "0x0", 0))
    }
}

// Given a block return the depth of the block
def getDepthOfBlock(id: String) = {
    val node = nodeMap.getOrElse(id, new LightBlock("0x0", "0x0", -1))
    node.depth
}
}

```

Listing A.1: Implementaion of GHOST_DepthSet

A.3 Installation Instructions

A.3.1 Installing Scala

Ensure that a Java 8 JDK is installed. If you don't have a version greater than 1.8, install the JDK.

After the successful installation of JDK install sbt for your machine,

- Mac: <https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Mac.html>
- Windows: <https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Windows.html>
- Linux: <https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Linux.html>

A.3.2 Installing Redis

Install Redis for your operating system and start the Redis process. Source: <https://redis.io/topics/quickstart>

A.3.3 Installing NPM

Install npm for your machine. Documentation can be found here: <https://docs.npmjs.com/downloading-and-installing-node-js-and-npm>

A.3.4 Installing Yarn

Install yarn from the source: <https://yarnpkg.com/lang/en/docs/install>. Navigate to the FRONTEND/ethereum folder and run *yarn install* for installing all the package dependencies.

A.3.5 Executing the code

Backend

Once all the steps are completed, download the source code from the github repository and access the VIBES_ETHEREUM folder. Run *sbt server/run* on the console. The

server starts at `localhost:8080`. Logs for the run are accessible as a .txt file at `VIBES_ETHEREUM/logs/akka.log`

Frontend

Navigate to the `FRONTEND/ethereum` folder and run `yarn start` to start the frontend at `localhost:3000`

Bibliography

- [1] “Ethereum : A next-generation smart contract and decentralized application platform.” <https://github.com/ethereum/wiki/wiki/White-Paper>. Accessed: 2018-11-12.
- [2] “Merkling in ethereum.” <https://blog.ethereum.org/2015/11/15/merkle-in-ethereum/>. Accessed: 2018-11-12.
- [3] “Truffle - ganache.” <http://truffleframework.com/ganache/>. Accessed: 2018-11-12.
- [4] 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*, pp. 19–20, ACM, 2017.
- [5] “Problems solved by the akka actor model.” <https://doc.akka.io/docs/akka/2.5.3/scala/guide/actors-intro.html>. Accessed: 2018-11-12.
- [6] “Akka actors intro.” <https://doc.akka.io/docs/akka/current/guide/actors-intro.html>. Accessed: 2018-11-12.
- [7] “Akka actor supervision.” <https://doc.akka.io/docs/akka/2.5/general/supervision.html>. Accessed: 2018-11-12.
- [8] “Bitcoin: A peer-to-peer electronic cash system,” 2008.
- [9] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, “Byzantine fault tolerance, from theory to reality,” in *Computer Safety, Reliability, and Security* (S. Anderson, M. Felici, and B. Littlewood, eds.), (Berlin, Heidelberg), pp. 235–248, Springer Berlin Heidelberg, 2003.
- [10] R. Beck, J. S. Czepluch, N. Lollike, and S. Malone, “Blockchain-the gateway to trust-free cryptographic transactions.,” in *ECIS*, p. ResearchPaper153, 2016.
- [11] “Coin market cap.” <https://coinmarketcap.com/>. Accessed: 2018-11-12.

- [12] “Ethereum: A secure decentralised generalised transaction ledger byzantium version adc4e61 - 2018-04-04,” 2015.
- [13] “The meaning of decentralization.” <https://medium.com/@VitalikButerin/the-meaning-of-decentralization-a0c92b76a274>. Accessed: 2018-11-12.
- [14] “Hive - ethereum end-to-end test harness.” <https://github.com/karalabe/hive>. Accessed: 2018-11-12.
- [15] A. Montresor and M. Jelasity, *PeerSim: A scalable P2P simulator*. 2009.
- [16] J. Banerjee, D. Goswami, and S. Nandi, “Opnet: A new paradigm for simulation of advanced communication systems,” pp. 319–328, 2014.
- [17] “Domain driven design.” http://dddcommunity.org/learning-ddd/what_is_ddd/. Accessed: 2018-11-12.
- [18] “The reactive manifesto.” <https://www.reactivemanifesto.org/>. Accessed: 2018-11-12.
- [19] R. Rivest, “Cryptography,” in *Handbook of Theoretical Computer Science*, edited by J. van Leeuwen, Elsevier Science Publishers B.V.
- [20] H. Delfs and H. Knebl, *Introduction to Cryptography: Principles and Applications (Information Security and Cryptography)*. Springer, 2007.
- [21] D. E. Newton, *Encyclopedia of Cryptology*. ABC-CLIO, 1997.
- [22] R. Steinmetz and K. Wehrle, *Peer-to-Peer Systems and Applications (Lecture Notes in Computer Science)*. Springer, 2005.
- [23] A. Galán-Jiménez, Jaime Gazo-Cervero in *Overlay Networks: Overview, Applications and Challenges*, International Journal of Computer Science and Network Security.
- [24] “Distributed ledger technology: beyond block chain (pdf) (report) by uk government, office for science.” https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf. Accessed: 2018-11-12.
- [25] “Ethereum beige paper.” <https://github.com/chronaeon/beigepaper/blob/master/beigepaper.pdf>. Accessed: 2018-11-12.
- [26] “KECCAK implementation overview.” <https://keccak.team/files/Keccak-implementation-3.2.pdf>. Accessed: 2018-11-12.
- [27] “Modified ghost protocol.” <https://github.com/ethereum/wiki/wiki/White-Paper#modified-ghost-implementation>. Accessed: 2018-11-12.

- [28] “Recursive length prefix.” <https://github.com/ethereum/wiki/wiki/RLP>. Accessed: 2018-11-12.
- [29] “Recursive length prefix.” <https://en.bitcoin.it/wiki/Testnet>. Accessed: 2018-11-12.
- [30] 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, pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.
- [31] “Node discovery protocol.” <https://github.com/ethereum/wiki/wiki/Node-discovery-protocol>. Accessed: 2018-11-12.
- [32] “Server sent events.” <https://www.w3.org/TR/eventsource/>. Accessed: 2018-11-12.
- [33] “Redis database.” <https://redis.io/>. Accessed: 2018-11-12.
- [34] “Akka.” <https://akka.io/>. Accessed: 2018-11-12.
- [35] “Reactive streams.” <http://www.reactive-streams.org/>. Accessed: 2018-11-12.
- [36] “Akka sse.” <https://doc.akka.io/docs/akka-http/current/sse-support.html>. Accessed: 2018-11-12.
- [37] “Akka streams.” <https://doc.akka.io/docs/akka/2.5/stream/>. Accessed: 2018-11-12.
- [38] “Akka http.” <https://doc.akka.io/docs/akka-http/current/>. Accessed: 2018-11-12.
- [39] “Leveldb.” <http://leveldb.org/>. Accessed: 2018-11-12.
- [40] “Rocksdb.” <https://rocksdb.org/>. Accessed: 2018-11-12.
- [41] “React.” <https://reactjs.org/>. Accessed: 2018-11-12.
- [42] “Angular.” <https://angular.io/>. Accessed: 2018-11-12.
- [43] “Blueprint.” <https://blueprintjs.com/>. Accessed: 2018-11-12.
- [44] “Etherchain.” <https://www.etherchain.org/>. Accessed: 2018-11-12.