Bachelor Thesis

CrowdBED: A Crowdsourcing system that leverages on Blockchain technology for rEliability and Decentralisation

Adamos Ttofari

University of Cyprus

Department of Computer Science

June 2020

University of Cyprus

Department of Computer Science

CrowdBED: A Crowdsourcing system that leverages onBlockchain technology for rEliability and Decentralisation

Adamos Ttofari

Advisor

Chryssis Georgiou

Diploma project has been submitted for partial fulfillment of the requierements of Informatics Degree acquisition from the University of Cyprus

June 2020

Acknowledgement

For the achievement of this project firstly, I would like to express my sincere gratitude to my advisor Assistant Professor Chryssis Georgiou. That gave me the opportunity to work on this unique project and the collaboration. During the year he push me towards the best result. Additionally I would like to give my gratitude to the CrowdBED team Dr. Evgenia Christoforou, Dr. Nicolas Nicolaou and Dr. Efstathios Stavrakis for providing advises during the meetings. Also I would like to thank Nicolas and Efstathios for providing a Virtual Machine for the development.

Special thanks to the Linux Foundation team for maintaining the Hyperledger projects and specifically the Sawtooth distributed ledger. Additionally I would like to thank the Sawtooth team for being available and helpful in the chat and specifically special gratitude to Arun for answering my questions regarding Sawtooth on StackOverflow. Furthermore I would like to thank the Computer Science Department for the courses and the knowledge provided that helped me during the implementation.

Finally I would like to thank my family and friends for being with me during my life and supporting me at every step.

Abstract

This thesis presents the implementation of CrowdBED, a prototype framework for crowdsourcing computational tasks on top of Blockchain technology to achieve reliability and decentralisation, thus achieving crowsourcing's full potential. Consider a case where a requester publishes a task with a deadline. Then a set of workers independently try to solve the task and submit their answers for a reward. The whole process is orchestrated by a smart contract on a blockchain network.

The implementation was done with Hyperledger Sawtooth distributed ledger framework, which is an open source project in the Hyperledger greenhouse, maintained by the Linux Foundation. Thus making it the first decentralised solution using a Hyperledger distributed ledger. CrowdBED has a flexible, modular and secure network design, while at the same time provides a language neutral interface for client implementation. It supports general-purpose deterministic computational tasks, meaning the requester just writes the computation script and provides the data for the workers to run and submit the result. Simultaneously having a permisionless access for the clients, hence achieving anonymity.

Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Motivation

Crowdsourcing environments promote various types of collaborative Internet-based activities and they have a societal, scientific and technological impact, changing forever the way humans and machines interact and collaborate for a given purpose. The partial availability of resources in personal computers and the access to the Internet, have led to the volunteer internet-based computing. Among the most popular examples is SETI@home [28] where its purpose was to analyze radio signals, searching for signs of extraterrestrial life. Additionally, computing platforms where users gain rewards for their contributions exist, for example Amazon Mechanical Turk [1].

Traditional crowdsourcing solutions are controlled by a centralized platform, managed by a believed-to-be trusted party, and connecting the requester of a crowdsourcing activity with the crowd. Although centralized platform solutions promise a smooth, relative trusted and reliable interaction of the requesters with the crowd, usually the usage of the platform implies large fees, imposes strict rules, but most importantly, it consists of a single point of

security and performance bottleneck. Thanks to recent technology advancements, blockchain technology has proven to be a transparent, scalable, decentralized solution, that provides strict security and performance guarantees. Thus, a distributed crowdsourcing approach could leverage the blockchain technology to alleviate the shortcomings of the centralized solutions, augmenting and promoting the catholic impact of crowdsourcing.

With the implementation of CrowdBED we aim to create a decentralised solution for crowdsourcing computational tasks. On the same time we aim for the validation to be requester independent, thus providing transparent reward distribution mechanism for the workers.

1.2 Contributions

CrowdBED is the first decentralised crowdsourcing solution using a Hyperledger Distributed Ledger. Specifically Hyperledger Sawtooth was used, thus providing a modular and byzantine fault tolerant network design. At the same time Sawtooth is a public blockchain, meaning it provides transparency for all the operation that took place on the network. Additionally CrowdBED provides a language neutral interaction interface for client implementation.

At the same time, giving a permissionless access for the clients, thus providing total anonymity. Plus because answers are deterministic, submissions are validated on the system, thus requesters can't steal the answers without giving reward to the workers. At the same time giving a simple interface for task creation, with only requirement to implement a computation script and provide the input for workers just to execute it and submit the output to CrowdBED.

1.3 Methodology

During the course of implementation Agile Software Development process took place. Meaning the implementation underwent various changes depending on the challenges appearing during the process and focused on the functionality of CrowdBED. Part of the methodology was the regular meetings and discussions with the members of the CrowdBED team that guided the major decisions about the system.

Initially a study took place on how different distributed ledgers work and deciding how can they can solve the problem. Specifically, three different Hyperledger distributed ledgers

were studied Fabric [22], Iroha [8] and Sawtooth [10]. After running a test network and completing the tutorial for every ledger, we examined the advantages and the disadvantages of every ledger, Hyperledger Sawtooth was the choise for the CrowdBED implementation.

After that, the main focus was on the implementation of the Transaction Family because of the modularity of Sawtooth that provides the separation between application logic and network. Meaning that development environment was used. At the same time a client prototype was implemented as Command Line Interface.

When the first version was ready then the implementation was deployed to a network simulator with Docker containers [32] to test the correctness of the implementation. After that any additions to the system were tested firstly on the developer mode of Sawtooth with one node and then on the network simulator.

Finally after the finalization of the implementation EC2 t2.micro Virual Machines were deployed in the Amazon Web Services for testing CrowdBED correctness on a real network environment.

1.4 Thesis Structure

The thesis consists of eight chapters:

- **Chapter One**: is an brief introduction about CrowdBED motivation, contribution, objective and methodology.
- **Chapter Two**: Previous knowledge is discussed, in order gain understanding of basic concepts that are used in the CrowdBED implementation.
- **Chapter Three**: Hyperledger Sawtooth is examined thoroughly, in order to get understanding how CrowdBED network works.
- **Chapter Four**: How data is structured in CrowdBED. For every data type, the fields are explained in detail and how they contribute to the crowsourcing paradigm in CrowdBED.
- **Chapter Five**: This chapter discusses the logic behind CrowdBED and how client requests are handled in the system. To support the basic functions of crowdsourcing.

- **Chapter Six**: In this chapter client semantics are discussed in order to achieve a smooth communication between client and the CrowdBED network.
- **Chapter Seven**: In this chapter we discuss the implementation of CrowdBED.
- **Chapter Eight**: Conclusions are made and future work is discussed.

Chapter 2

Background Knowledge

Contents

Before discussing how CrowdBED works it is important to understand some concepts that were very important in the implementation of the system.

2.1 Crowsourcing

With the availability of computational resources and access to the broadband Internet. Internet based computing has began. With various projects that helped to contribute to scientific problems from searching for alien life (SETI@home [28]) to fighting the COVID-19 pandemic (Folding@home [5]) executed in personal computers when they are idle. Providing a flexible and scalable solution when normally data centers would take place for these executions. Moreover computational platfroms where workers who compute for reward exist, for example Amazon Mechanical Turk [1].

CrowdBED follows the Master-Worker paradigm, where a master or requester publishes a task on a platform. The interested workers then get the task and solve it on their personal machines and submit their answers to the platform and get the expected reward if they submit the correct answer. In this paradigm, malicious workers could report wrong answers and malicious masters that report that the answers are incorrect and steal them [38].

2.2 Cryptography

Cryptography is the process that two parties want to communicate in secret and modify their messages in a predefined process that only they can undo. It is used in secure communications for thousands of years [36].

Traditional cryptography used symmetric key cryptography where the sender and the receiver used the same key for encryption and decryption. The main challenge is how the sender can send the secret key in a public environment like the Internet.

2.2.1 Public Key Cryptography

Currently in modern age, Public Key cryptography is used for secure communications. In order to communicate each entity has a pair of mathematically related keys, called public and private key. Private key is only known to the entity and the Public key can be published

to everyone. Encryption of a message happens with one of the keys, and decryption happens only with the other key [36].

2.2.2 Hash Functions

Hash functions are one-way deterministic functions applied to a sequence of data creating a unique digest of fixed size. The basic requirements are the following [36]:

- **Preimage resistance**: Having a digest $h(x)$it should be impossible to find the data sequence x.
- **Second Preimage resistance**: Knowing the digest $h(x)$ and the data sequence $x$it should be impossible to find a different data sequence y that $h(x) = h(y)$.
- **Collision resistance**: It should be impossible to find any pair of different data sequences x and y that $h(x) = h(y)$.

2.2.3 Digital Signatures

A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents [36]. A valid signature gives a very strong reason that the message was created by the sender and was not altered in transit. The process is the following:

- The sender applies hash function to the message. Then encrypts the digest with the private key, thus creating the signature. Finally sends the original message and the signature.
- The receiver gets the message and the signature. Decrypts the signature with the sender's public key and checks the message's digest with the decrypted digest.

2.3 Fault Tolerance

Fault tolerance refers to the ability of a system (computer, network, cloud cluster, etc.) to continue operating normally when one or more of its components fail [39]. The objective of creating a fault-tolerant system is to stop disruptions arising from a single point of failure,

ensuring the high availability and continuity of applications or systems. Every system should achieve the following goals:

- **Availability**: A system should be ready to be used at any time.
- **Reliability**: A system should continue to work normally without a failure for a very long time.
- **Safety**: Even after a failure the consequences should not be catastrophic.
- **Maintainability**: How easy a failed system can be repaired and return to normal functioning.

A common solution is state machine replication. A state machine, at any moment stores the state of the system. It receives inputs and by applying them in a sequential order ,updates the state and produces an output. In distributed systems the components simulate a state machine and at any moment all of them have the same state of the system. In simple terms a set of nodes process the same sequence of commands and are in the same state. Meaning that a client can interact with any of the components without finding any distinction between them [37].

2.3.1 Crash Fault Tolerance

A crash tolerant system is a system that operates efficiently while tolerating crash faults of its components [39]. The system, although some nodes crash, it continues to operate normally without interruptions or unexpected behaviour.

Currently a popular solution used in the Hyperledger implementations is Raft [35] for state machine replication. In Raft consensus the network passes through a series of rounds. In each round a leader is chosen and executes some commands where the rest of the nodes repeat the same commands. For the system to work properly at least the half of the nodes need to function properly.

2.3.2 Byzantine Fault Tolerance

Byzantine nodes made first appearance in the paper: The Byzantine Generals Problem [29]. A node is Byzantine if it exhibits arbitrary/malicious behavior (deviating by its specification)

and might intentionally cause problems to the system. The problem is defined as follows:

Imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that:

- All loyal generals decide upon the same plan of action.
- A small number of traitors cannot cause the loyal generals to adopt a bad plan.

The conclusion of the paper is that the generals cannot make the decision unless the number of generals is strictly greater than three times the number of traitors. Meaning that the problem can't be solved unless $N >= 3f + 1$ where N is the number of generals and f the number of traitors.

In simple terms the above problem states that even with a small percentage of malicious participants in a system such that the system can still work correctly without interruptions or unexpected behaviour. Byzantine Fault Tolerance was introduced by the PBFT paper [25]. A state machine replication algorithm to achieve Byzantine Fault Tolerance. This solution is used in Hyperledger Sawtooth to achieve consensus and will be discussed in Section 3.6.

2.4 Blockchain

What is Blockchain? Simply speaking it is a chain of blocks. But there is more than that, other may reference blockchain as an append only immutable ledger. Every block stores a lists of transactions cryptographically signed by the clients, and the hash of the previous block. The blockchain is stored in every peer's local storage. Making it hard to exploit because in order to modify a block's contents a malicious perticipant must recreate every block after it in order to create a valid blockchain and replace. [34]

Blockchain technology became popular with the introduction of Bitcoin by Satoshi Nakamoto [33]. It was the first cryptocurency to solve the double spending problem using a decentralized network without a central authority for example a bank. In order to add a block it uses the Proof of Work [26], a lottery based algorithm where a peer must guess a number that will make the block's hash have a specific amount of zeroes that is defined by

the network in order to have one block every 10 minutes. But due to that there is a possibility of a fork where multiple versions of the ledger appear. So in order to have one ledger the peers choose the longest chain after some time. The main flaw with bitcoin and proof of work is that it consumes a lot of power and has harmull impact to the environment.

But despite the flaws of bitcoin, blockchain technology became popular because of it's properties. Giving the start for many popular projects. Etherium [3] is a popular example because it maintains it's own cryprtocurency named Ether and adds the support of smart contracts, developer programs that are executed and handle the logic of transactions on the ledger. At this moment there are over 4000 cryptocurencies [2]. Linux Foundation started the Hyperledger [7] group for open source blockchain solutions. Finally even tech giants like Facebook understanding the importance of blockchain anoucent their own cryptocurency Libra [11], with that proving that blockchain will make an impact in the future.

Aditionally blockchain networks are so many and have many applications that are divided into categories. There is a division between the blockchains based on their data access and on the peer access. When we reference the data access there are two categories *public and private* ledgers. Where the public ledgers everyone can submit a transaction and read the blockchain contents. On other hand private blockchains cannot be viewd publicly and only authorised members can view the blockchain contents. Meanwhile peer access is divided into *permissioned and permisionless* blockchains. Where the permissionless ledgers allow everyone to add their machine to the blockchain network to maintain it, on the other hand in permissioned networks a system administrator must add the new members.

Permissionless networks have high security, open environment and provide anonymity. Notable examples are Bitcoin and Etherium. Popular reasons to use a public solution is to have public transparency, true decentralization, immutability and great business to consumer solution. But to achieve that, permissionless blockchains take a lot of time to add a block in order to make sure everything is correct. For example Proof of Work takes almost 10 minutes to add a block. On other hand permissioned ledgers are efficient, usually private, stable and are a great bussiness to bussiness solutions. Notable example is Hyperledger Fabric.

Depending on the problem the appropriate choice is made, in the case of a decentralised currency that is not dependand on a central authority like a bank a public ledger is the choice, meanwhile the case of supply chain, a private ledger between the supply companies is more better choice.

2.5 Hyperledger Project

Hyperledger is a community that started from Linux Foundation. Its main purpose is to develop a suite of stable frameworks, tools and libraries for enterprise-grade blockchain deployments. It serves as a home to many distributed ledger frameworks like Fabric, Iroha and Sawtooth, as well as tools that can help blockchain deployment like Avalon and Aries.

2.5.1 Hyperledger Fabric

Hyperledger Fabric [22] is a open source private enterprise blockchain framework maintained mainly by IBM. It is the first to run distributed applications called Chaincodes with the use of general use laguages like Java, Golang or Javascript without the use of a cryptocurrency or depending on the system. It has a permisioned design and is crash fault tolerant using the Raft consensus algorithm. It is the most popular and maintained project in the Hyperledger Community.

In brief, the the architecture of Fabric is the following. The nodes in the network are divided into two categories Peers and Orderers. Orderers are responsible for getting endorsed transactions groupping them in blocks and adding them to the ledger. Meanwhile peers contain the chaincodes and a ledger copy, they are the ones that endorse transaction proposals.

When an application proposes a transaction it sends it to the peers for endorsements. Then the peers execute the transaction locally and depending on the result endorses the proposal and sends it back. The application receives the endorsements and when the endorsment policy is met the application sends it to the ordering service. The orderers collect endorsed transactions and add them to blocks. They reach consensus with the use of Raft and sends the block to peers to update the global state.

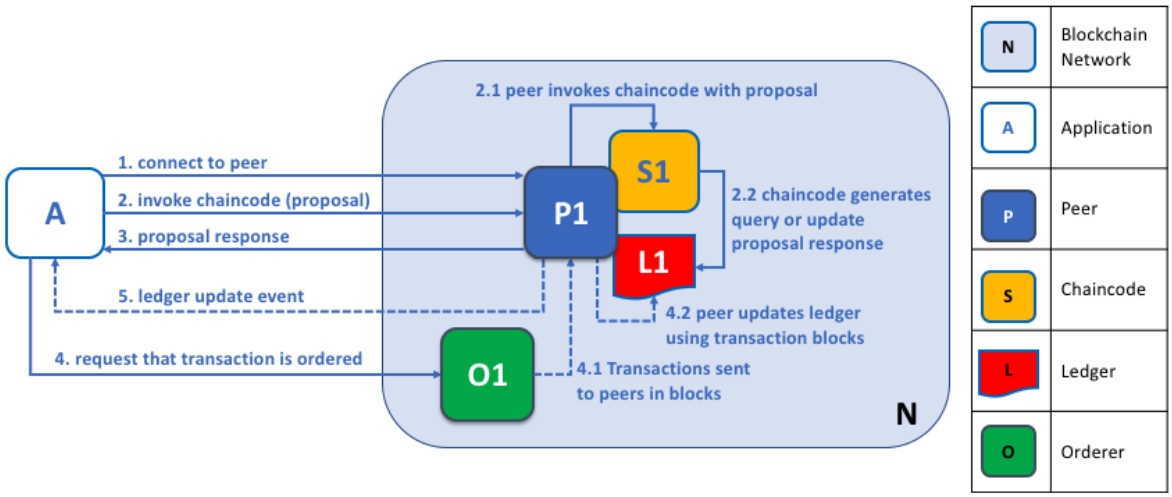

Figure 2.1: Hyperledger Fabric Overview [4]

At the current stage, Hyperledger Fabric is crash fault tolerant and requires every participant to be trusted. Leaving the network vulnerable to a malicious orderers or peers. Additionally Fabric has the ability to have private data on the ledger. Meaning even as a decentralized blockchain framework it has to be managed by a believed-to-be-trusted party. Because of that Fabric wasn't the choice for CrowdBED's implementation.

2.5.2 Hyperledger Iroha

Hyperledger Iroha [8] is an other distributed ledger framework maintained by Soramitsu. It is popular in Japan with many use cases. It features simple deployment, variety of libraries, role-based access control, modular design and assets/identity management. Additionally it supports multi signature transactions. It is a private blockchain with permisioned design for data on the ledger. Features a novel consensus algorithm called YAC. Currently it doesn't support any smart contracts and is mainly used for asset management.

Briefly the base architecture of Iroha is the following. Initially the transaction is proposed from the client through the Torii a gRPC gateway for accepting transactions. After that the transaction goes to the TSP (Multisignature Transactions Processor) to be exchanged with other peers for signatures that are missing. After that the transaction is sent to the oprdering service for a stateless validation of the transaction. After that the transaction is verified and updates the world state view in a simulator and added to a block. Then the block is send to network in order to be endorsed by the YAC algorithm. When the block is accepted the block

is stored in the blockstore and updates the world state. In the meantime sychronizer makes sure the blockchain is up to date.

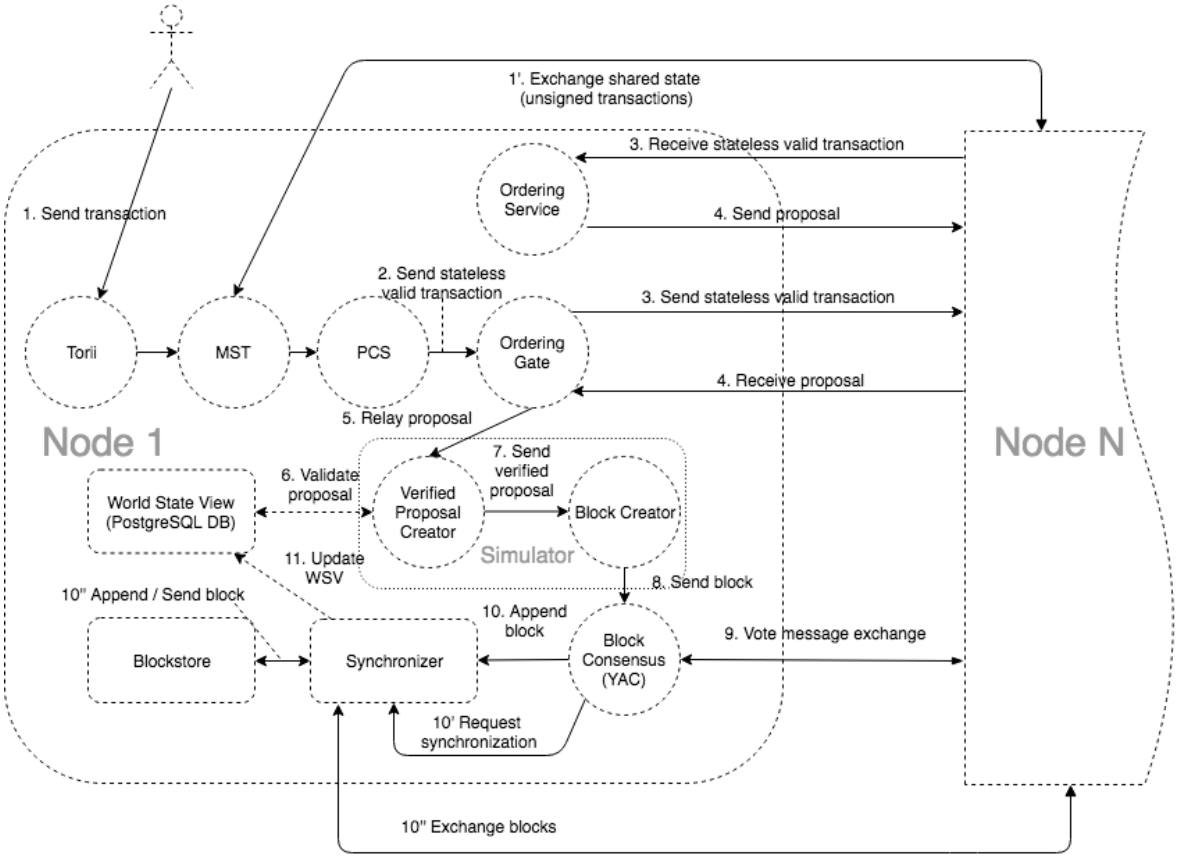

Figure 2.2: Hyperledger Iroha Overview [9]

At the current stage Hyperledger Iroha is a private blockchain software, is crash fault tolerant and doesn't support smart contracts. The reason Iroha isn't the framework of CrowdBED's implementation is because at a task's deadline validation must take place that needs a smart contract to support this operation.

2.5.3 Hyperledger Sawtooth

Hyperledger Sawtooth [10] is a flexible and modular blockchain framework created in Intel Labs and maintained by the Linux Foundation. The architecture seperates the application logic from the network layer. The application logic is implemented as transaction families and run as independent modular processes on every node. Sawtooth supports various consensus algorithms like Practical Byzantine Fault Tolerance and Proof of Elapsed Time. It is a public blockchain, meaning everyone can see the blockchain contents and submit transactions

without a central authority. But is a permisioned network meaning in order to add a peer it must be added by a system administrator.

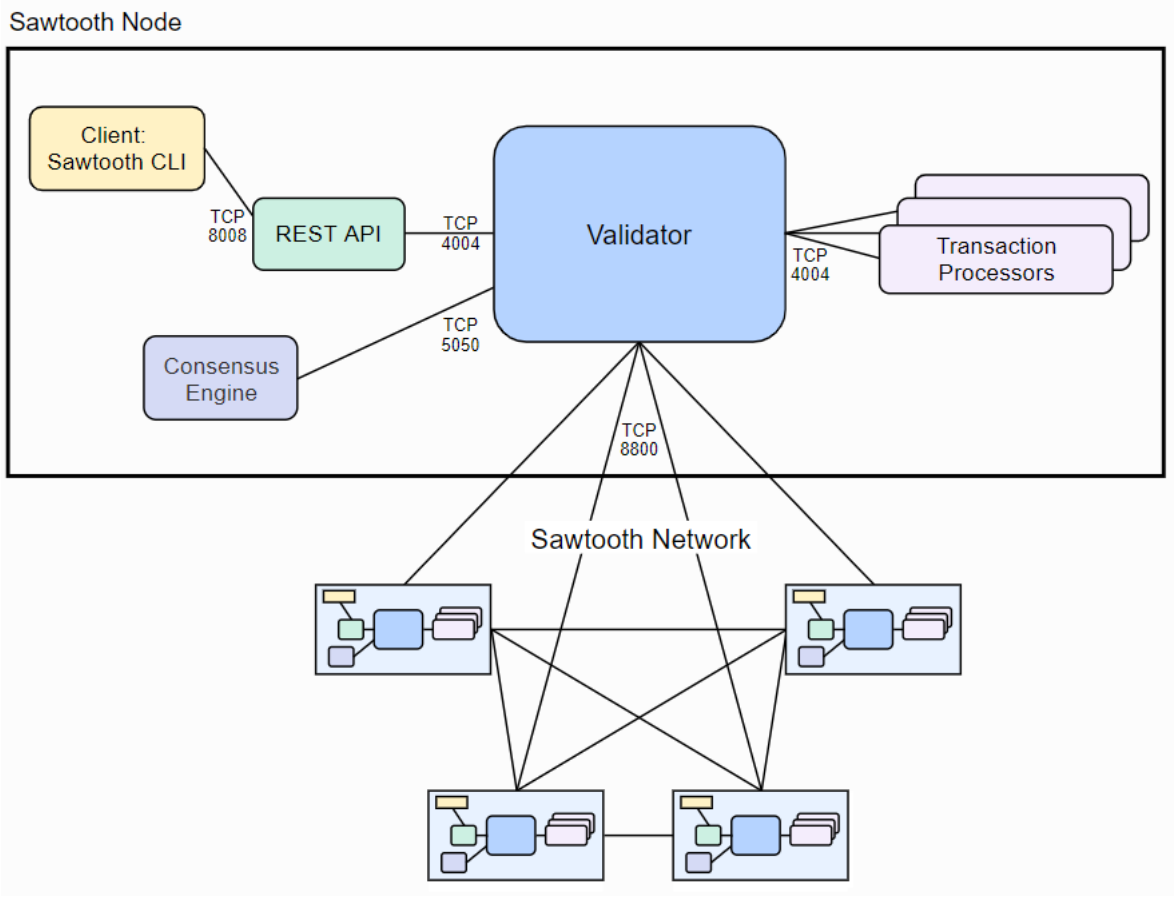

Figure 2.3: Hyperledger Sawtooth Overview [14]

Because of the separation between the core system and the application, the simple transaction family implementation, the Byzantine fault tolerance support. At the same time the ability for permissionless support for clients providing anonymity and transparency. Hyperledger Sawtooth was the choice for the CrowdBED's implementation. In the next chapter an in depth analysis of Sawtooth will be discussed.

2.6 Other Technologies Used

Some other technologies were used for the implementation of CrowdBED and are mentioned in thesis. This section serves as a brief explanation what additional technologies were used in the implementation.

2.6.1 Protocol Buffers

Protocol buffers [12] are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. The developer defines how you want data is to be structured once, then a special generated source code can be used to easily write and read the structured data to and from a variety of data streams and using a variety of languages.

The developer must write a .proto file where he defiones the data structure using the proto language structure. Then he compiles the file and generates a source code for the chosen language. Thus providing a language neutral way to define data and serialization mechanism providing abstraction and interoperability between different technologies.

2.6.2 gRPC

gRPC [6] is a language neutral language RPC framework. In RPC a client can call a method on a server on a different machine as if it were a local object, making it easier for creating distributed applications and services. By default, gRPC uses Protocol Buffers, Google's open source mechanism for serializing structured data. Thus providing an environment for heterogeneous client and server implementations.

2.7 Related Work

There are others Blockchain-based crowdsourcing systems. Notable examples are CrowdBC [30], ZebraLancer [31] and WorkerRep [24]. All of these are implemented on Ethereum [3], an permisionless blockchain network and use some degree of Registration Authority for the users. Meanwhile CrowdBED is implemnted on Hyperledger Sawtooth providing some degree of administration on the network. Additionally providing permisionless registration for the users, thus providing total anonymity for the users.

In ZebraLancer the encrypted submissions are added to the ledger using public key cryptography where provided by the requester. Eventually when the requester will gain enough submissions he will decrypt them and decide how to reward the workers based on the answers. On the other hand in CrowdBED, the transaction family is responsible for the reward of the users without the requester's intervention.

Meanwhile on CrowdBC requester must define how the task will be validated. Whitch re-

quires some effort for creating the task from the requester side. On the other hand CrowdBED uses majority voting for determining the answer. The reason is that CrowdBED aims computational tasks that the data and code is provided by the requester meaning the answer is unique, thus it is easy to cross-check the answer. Additionally the only thing required by the requester to write the code and the data, in the current implementation any simple Python3 script is supported.

On the other hand WorkerRep provides a unique validation scheme where the workers validate each other solutions. In order to get a worker's solution validated he must validate solutions of other workers after his submission. In the CrowdBED implementation the blockchain is responsible for a centralized the validation by the Transaction Family.

Chapter 3

A Deep Dive into Sawtooth

Contents

For the implementation of CrowdBED, Hyperledger Sawtooth was used. Sawtooth is one of the distributed ledger frameworks in the Hyperledger ecosystem. Hyperledger Sawtooth offers a flexible and modular architecture separates the network from the logic that runs on the ledger. Meanong smart contracts can specify the rules for applications without needing to know the underlying design of the core system. Hyperledger Sawtooth supports a variety of consensus algorithms, including Practical Byzantine Fault Tolerance (PBFT) and Proof of Elapsed Time (PoET).

As a distributed ledger, a Sawtooth network is a Peer to Peer network, which consists of multiple nodes. Each peer is called Validator Node and contains various components that run inside containers. The main components are the Validator, the Rest API, the Consensus Engine and one or more Transaction Proccessors. The components communicate with the use of ZeroMQ messaging library. This architecture helps to keep the modularity of the system.

Sawtooth represents state for all transaction families in a single instance of a Merkle-Radix tree on each validator. The process of block validation on each validator ensures that the same transactions result in the same state transitions and that the resulting data is the same for all participants in the network.

In this chapter we will discuss the architecture of Hyperledger Sawtooth in order to gain a better understanding on how CrowdBED operates.

3.1 Global State

The main goal of Sawtooth is to achieve consensus on the global state [16] of data on the ledger. It is achieved efficiently with the use of Radix-Merkle Tree.

3.1.1 Merkle Hashes

Sawtooth uses an addressable Merkle-Radix tree to store data. A Merkle tree is a rooted tree structure where a node's value computed as a hash of it's aggregated children values. The leaf because it doesn't have children and keeps the hash of the data the address stores. It is an efficient way to check the expected value after validating blocks because a single root hash is calculated in the root of the tree as the global state value. If after validating a block the value on the root is a a different hash, the block is not considered valid.

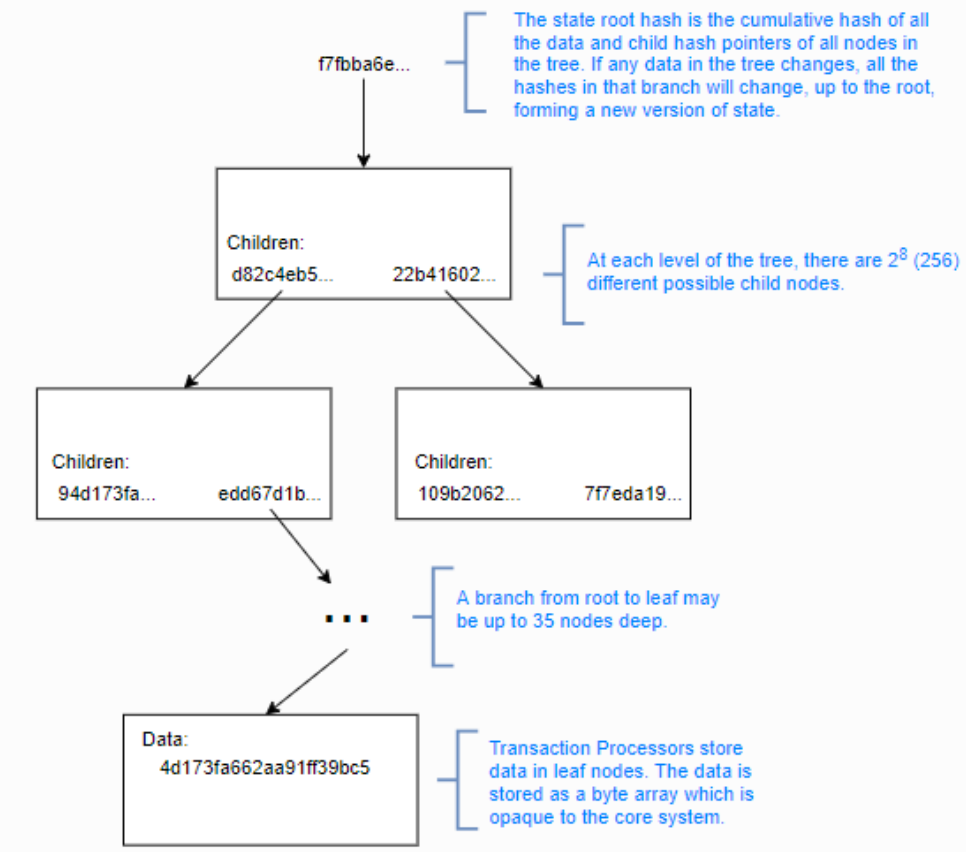

Figure 3.1: Merkle Tree [16]

3.1.2 Radix Addresses

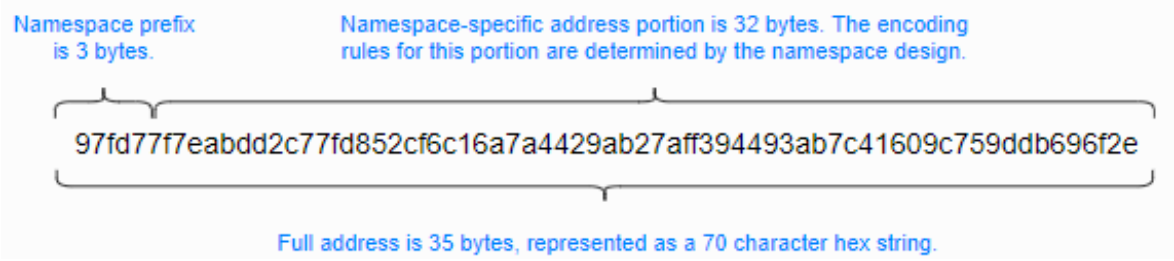

Figure 3.2: Radix Address [16]

The tree is an addressable Radix tree because addresses uniquely identify the paths to leaf nodes in the tree where information is stored. Because every node has 256 children the choice of the next node can be identified by the current bit. An address is a hex-encoded 70 character string representing 35 bytes. This means that the depth of the tree is 35. The address format is divided into two parts: the namespace (3 bytes) and the Namespace-specific address (32 bytes). The namespace prefix is used in dividing types of data on the ledger and adding write

permission to the addresses of same namespace. The remaining bytes are must be defined by the developer to find specific data in the Tree. When a update is occurred, the changes are happening across the path from the leaf to the root, thus calculating the new global state.

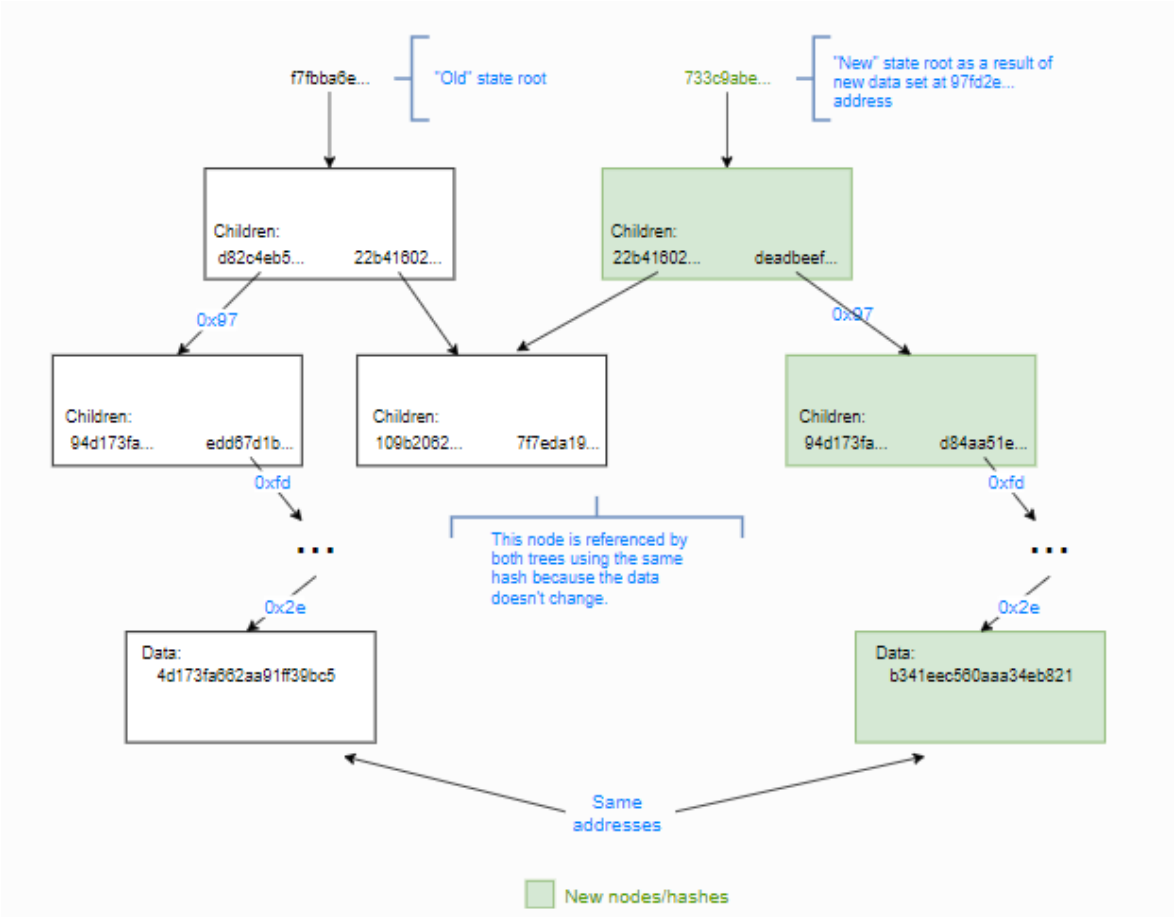

Figure 3.3: Tree Update [16]

3.1.3 Serialization

In addition to namespace design, developers need to define the serialization of data. A simple way to picture the whole process is like a hashmap where the key is the address and value is a byte array of serialized data. The processor provides commands like set(address, data) and get(address). It is important when a data is serialized, the serialization function to provide always the same result across space and time. Data structures which don't enforce ordered serialization (e.g. sets, maps, dicts) should be avoided. Because as explained earlier the leafs compute the hash of data and update the whole path to the root.

3.2 Transactions and Batches

The way to modify the data on the ledger is with transactions. Transactions are proposed directly by the clients and applied by the validator nodes. Thus causing changes to the ledger. Transactions must be wrapped inside Batches, which is a list of Transactions. Batches and Transactions are serialized with the use of Protocol Buffers [21].

3.2.1 Transaction Structure

Transaction is an atomic structure of data that is used to initiate smart contract execution on the ledger.

Transaction
Byte[] **Header**
String **Header Signature**
Byte[] **Payload**

Figure 3.4: Transaction Structure

The fields of a transaction are defined as follows:

- **Header**: Is the serialized version of the transaction header (with the use of Protocol Buffers). Contains metadata about the transaction.
- **Header Signature**: Transaction Header signed by the private key of the client who is constructing the transaction.
- **Payload**: Payload is the serialized data that will determinate the changes to the ledger.

3.2.2 Transaction Header Structure

Transaction Header
String **Batcher Public Key**
String[] **Dependencies**
String **Family Name**
String **Family Version**
String[] **Inputs**
String[] **Outputs**
String **Payload SHA512**
String **Signer Public Key**
String **Nonce**

Figure 3.5: Transaction Header Structure

The fields of a transaction header are defined as follows:

- **Batcher Public Key**: The public ECDSA key using the secp256k1 curve of the entity that will sign the batch that contains the transaction and send it to the ledger. Can be the public key of the client if the batch is created by him.
- **Dependencies**: A list of transaction headers as hex strings. All the specified transactions are needed to be added to the ledger before this transaction is processed. If the transaction headers (that are unique) are not added to the ledger the transaction will be added to a queue and will stay there until these transactions are added before being processed.
- **Family Name**: The name of the transaction Family that will process the transaction.
- **Family Version**: The version of the transaction Family that will process the transaction.
- **Inputs**: A list of radix addresses that the transaction family will need to read in order to process the transaction. If during the processing transaction family tries to read from address that isn't in the list, then the transaction fails.

- **Outputs**: A list of radix addresses that the transaction family will need to modify during the process of the transaction. If during the processing transaction family tries to write to address that isn't in the list, then the transaction fails.
- **Payload SHA512**: The digest of the payload using the SHA512 algorithm.
- **Signer Public Key**: The public ECDSA key using the secp256k1 of the client.
- **Nonce**: A random string.

3.2.3 Batch Structure

Batch is a list of transactions, it is required every transaction to be contained in a batch in order to be processed by the transaction processor.

Batch
Byte[] **Header**
String **Header Signature**
Transaction[] **Transactions**

Figure 3.6: Batch Structure

The fields of a batch are defined as follows:

- **Header**: Is the serialized version of the batch header (with the use of Protocol Buffers). Contains metadata about the batch.
- **Header Signature**: Batch Header signed by the private key of the batcher who is constructing the batch.
- **Transactions**: Is a list of transactions that are included in the batch.

3.2.4 Batch Header Structure

Batch
String **Signer Public Key**
String[] **Transaction IDs**

Figure 3.7: Batch Header Structure

The fields of a batch header are defined as follows:

- **Signer Public Key**: The public ECDSA key using the secp256k1 of the batcher in hex form.
- **Transaction IDs**: The transaction headers serialized in hex form. It is important to be in the same order as in the Transactions field in the batch.

3.3 Sawtooth Network

The network layer is responsible for communication between validators in a Sawtooth network, including performing initial connectivity, peer discovery, and message handling. Upon startup, validator instances begin listening on a specified interface and port for incoming connections. Upon connection and peering, validators exchange messages with each other based on the rules of a gossip or epidemic protocol.

A primary design goal is to keep the network layer as self-contained as possible. For example, the network layer should not need knowledge of the payload of application messages, nor should it need application-layer provided data to connect to peers or to build out the connectivity of the network. Conversely, the application should not need to understand implementation details of the network in order to send and receive messages.

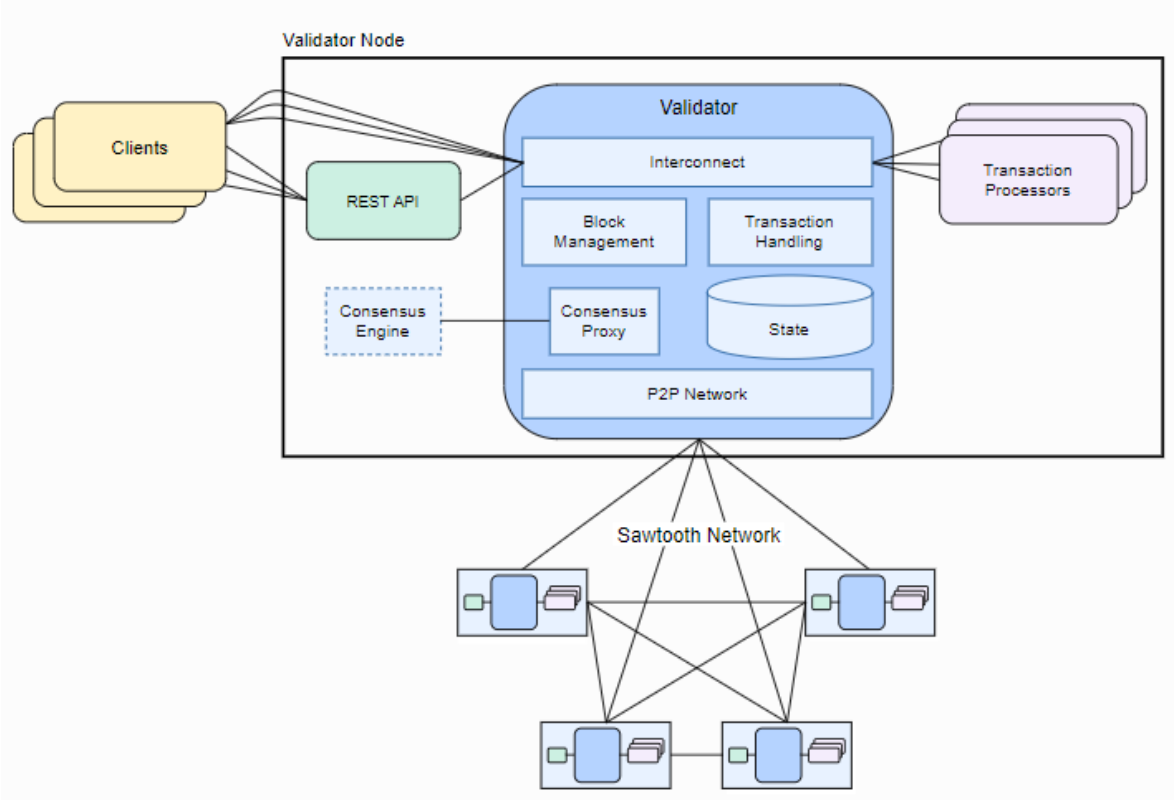

Figure 3.8: High Level Overview [14]

Every node contains various components that help it do it's role as described in the Sawtooth Glossary [17]:

- **Validator**: Component responsible for validating batches of transactions, combining them into blocks, maintaining consensus with the Sawtooth network, and coordinating communication between clients, transaction processors, and other validators on the network. Each validator has the following components:
  - **State**: Database that stores a local (validator-specific) record of transactions for the blockchain. Sawtooth represents state in a single instance of a Merkle-Radix tree on each Sawtooth node.
  - **Consensus Proxy**: Interface that allows a consensus engine to interact with the validator in order to handle consensus functionality in a separate process.
  - **Block Management Component**: A component that is responsible for the block management and validation. [18]

  - **Transaction Handling Component**: A component that is responsible for handling transactions and creation of the blocks. [18]
  - **Interconnect**: Responsible for the communication with the components of the validator.
  - **Network Interface**: A component that is responsible for the communication with the peer to peer network.

- **REST API**: In Sawtooth, a core component that adapts communication with a validator to HTTP/JSON standards. Sawtooth includes a REST API that is used by clients such as the Sawtooth CLI commands. Developers can use this REST API or develop custom APIs for client-validator communication.

- **Consensus Engine**: Sawtooth component that provides consensus-specific functionality for a Sawtooth node. The consensus engine runs as a separate process on the node and communicates with the validator through the consensus API.

- **Transaction Processor**: Validates transactions and updates state based on the rules defined by the associated transaction family.

3.4 REST API

Hyperledger Sawtooth provides a REST API that allows clients to interact with a validator. It provides a simple interface for client use. It simply passes every request to the validator to be authorized with signature verification or another strategy that is defined by a transaction processor. The REST API process runs as a separate process, rather than as part of the validator process. It treats the validator as a black box, simply submitting transactions and fetching the results.

REST API Endpoint Specifications

Some of the requests a client can invoke are the following [20]:

- **POST /batches**: It is the main way that new batches are sent to the ledger to be added.

- **GET /batches**: Gets the list of accepted batches.

- **GET /batch_statuses**: Gets the status of the batch.

- **GET /state**: Gets the addresses and data that exist in the Merkle tree. If it gets as argument a address prefix it will return the list of addresses that have the same prefix with their data.
- **GET /blocks**: Gets the blocks on the blockchain.
- **GET /transactions**: Gets the list of accepted transactions.
- **GET /peers**: Gets the list of peers on the network.

Using the above commands it creates a language neutral interface for communication between client and the blockchain network, additionally it provides the transparency of the ledger.

3.5 Events

Sawtooth events [15] occur when blocks are committed — that is, the validator broadcasts events when a commit operation succeeds — and are not persisted in state. Each transaction family can define the events that are appropriate for its business logic.

An attribute is a key-value pair that contains transparent metadata about the event. The key is the name of the attribute, and the value is the specific contents for that key. The same key can be used for multiple attributes in an event.

A client can subscribe to events and stay in a blocking state while waiting for the network to broadcast them. The events as stated contain a key-value pair where the client can filter in order to get the events it is interested or are defined by the client's logic. The subscription happen through a ZMQ socket. Moreover a browser based applications can subscribe through Web Sockets but the functionality is limited.

3.6 Consensus in Sawtooth

Hyperledger Sawtooth provides four different consensus algorithms to handle the consensus process as written in the Sawtooth Glossary [17]:

- **Devmode consensus**: Simple random-leader consensus algorithm that can be used to test a transaction processor on a single Sawtooth node. (Devmode is short for "devel-

oper mode".) Devmode consensus is not recommended for a multiple-node network; it should not be used for production.

- **Raft**: Leader-based consensus algorithm that is designed for small networks with a restricted membership. Raft is crash fault tolerant, not Byzantine fault tolerant, and has finality (does not fork).

- **Proof of Elapsed Time**: Proof of Elapsed Time, a Nakamoto-style consensus algorithm that is designed to support large networks. PoET does not have finality (can fork). Sawtooth offers two version of PoET consensus:

  - **PoET-SGX** relies on a Trusted Execution Environment (TEE), such as Intel® Software Guard Extensions (SGX), to implement a leader-election lottery system. PoET-SGX is sometimes called PoET/BFT because it is Byzantine fault tolerant.

  - **PoET simulator** provides the same consensus algorithm on a system without a Trusted Execution Environment. PoET simulator is also called PoET/CFT because it is crash fault tolerant, not Byzantine fault tolerant.

- **Practical Byzantine Fault Tolerance**: Practical Byzantine Fault Tolerance, a voting-based consensus algorithm with Byzantine fault tolerance (BFT) that has finality (does not fork). Sawtooth PBFT extends the original PBFT algorithm with features such as dynamic network membership, regular view changes, and a block catch-up procedure.

Practical Byzantine Fault Tolerance

PBFT was proposed in 1999 by Miguel Castro and Barbara Liskov [25]. It is a leader-based and non-forking, unlike other lottery based algorithms. It does not support open-enrollment, but nodes can be added and removed by an administrator and requires full peering.

A PBFT network consists of a series of peers, where n is the number of peers in the network. There is a maximum number of "bad" peers that the PBFT network can tolerate. As long as this number of bad nodes—referred to as the constant f—is not exceeded, the network will work properly. For PBFT, the constant f is equal to one third of the peers in the network or $f = \frac{n-1}{3}$. No more than a third of the network (rounded down) can be "out of order" or dishonest at any given time for the algorithm to work.

As the network progresses, the peers move through a series of "views". A view is a period of time a node is the primary (leader) of the network. In other words peers take turns who will be the primary. In a N peer network, the peer 0 is primary in the view 0, the peer 1 is primary in the view 1 and so on; when the view reaches N then it will start from peer 0 and repeat. An easy to find the primary node is by finding the modulo of the view with the number of nodes. For example during the view 69 with a 7 peer network the node 6 is the primary because $69 \equiv 6 \pmod 7$.

In addition to moving through a series of views, the network moves through a series of sequence numbers. In Sawtooth PBFT, a node's sequence number is the same as the block number of the next block in the chain. For example, a node that is on sequence number 8 has already committed block 7 and is evaluating block 8.

To commit a block and make progress, the nodes in a PBFT network go through three phases Pre-prepare, prepare and commit.

Having the above in mind the following procedure takes place to commit a block:

- **Pre-Prepare Phase**: All peers begin in the PrePreparing phase; the purpose of this phase is for the primary to publish a new block and endorse the block with a PrePrepare message. The block is created and broadcasted through network gossip to the network. Then the primary peer will send a pre-prepare message to all the the peers to endorse the block. After a secondary peer validates the block it enters the preparing phase.

- **Prepare Phase**: In the Preparing phase, all secondary peers (not the primary) broadcast a Prepare message that matches the accepted PrePrepare message. Once a peer has 2f + 1 Prepare messages in its log that match the accepted PrePrepare, it will move on to the Committing phase.

- **Commiting Phase**: The Committing phase is similar to the Preparing phase; peers broadcast a Commit message to all peers in the network, wait until there are $2f + 1$ Commit messages in their logs, then move on to the Finishing phase. The only major difference between the Preparing and Committing phases is that in the Committing phase, the primary is allowed to broadcast a message.

- **Finishing Phase**: Once in the Finishing phase, each peer will tell its validator to commit the block for which they have a matching PrePrepare, $2f + 1$ Prepare messages,

and $2f + 1$ Commit messages. The node will then wait for a BlockCommit notification from its validator to signal that the block has been successfully committed to the chain. After receiving this confirmation, the node will update its state as follows: Increment its sequence number by 1, update its current chain head to the block that was just committed and reset its phase to PrePreparing.

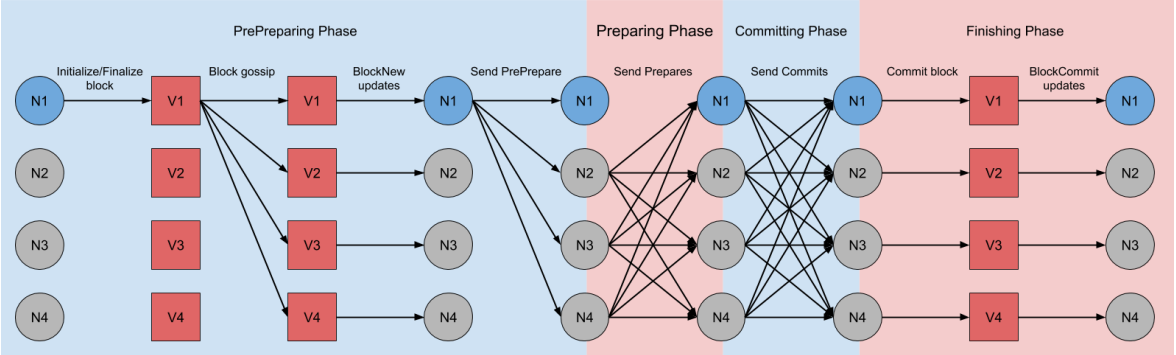

Figure 3.9: Sawtooth PBFT Overview [19]

In case a new peer is added to the network or has fallen behind the Sawtooth implementation has a mechanism to catch up with the rest of the network. If, while waiting to receive $2f + 1$ commit messages from its peers, a peer receives the next block in the chain, the peer has an opportunity to short-circuit consensus. This is done by validating the consensus seal in the new block and confirming that it is a valid proof for the block that the node is currently trying to commit. If it is, then the node copies the commit messages from the consensus seal into its own message log and commits the block.

3.7 A Day in the Life of a Hyperledger Sawtooth Transaction

In this section we will sum up and explain the whole process of how a transaction is added to the blockchain in a short and simple way without getting deep into technical details.

1. **Transaction Creation**: The transaction is created on the client side and added to a batch.

2. **Sending the Batch**: Then the client sends the batch to the validator through the REST API.

3. **Broadcasting the Batch**: The validator checks the signature and makes sure the batch is valid and broadcasts it to the other validators in the networks and store it to the local memory.

4. **Transaction Processing**: The validator takes the transactions from the batch and sends them to the transaction processor to process the transaction and get the new global state.

5. **Block Creation**: After an amount of transactions are complete the validator creates a block and sends it to the other validators through network gossip with minimal information (only the batch headers).

6. **Block Validation**: During the Pre-Preparing phase the seconadry peers validate the block by running the transactions in the batches in the block and making sure they produce the same global state using the radix merkle tree.

7. **Reaching Consensus**: Then the PBFT phases take place and consensus is reached and finally the block is commited to the ledger by every validator.

8. **Reading the Blockchain**: Finally after the block is commited, everyone can query the blockchain through the REST API and read the transaction contents.

3.8 CrowdBED Architecture

In this section we will discuss what were the choices for the CrowdBED architecture. Because Hyperledger is a flexible and modular system there were no many changes in the core architecture.

3.8.1 CrowdBED Network

The CrowdBED Network architecture follows the architecture of the sawtooth network. The reason is that the abstraction the Hyperledger Sawtooth provides, shifts the primary focus from the network managment to the transaction family implementation.

But due the requirement of deadlines and the lack of support for submitting transactions on specific time because of fear of skewed time between the peers. An additional lightweight

component was implemented called batcher that keeps transactions and submits them on a specific time. The requirements of CrowdBED transaction family are not affected by the skewed time.

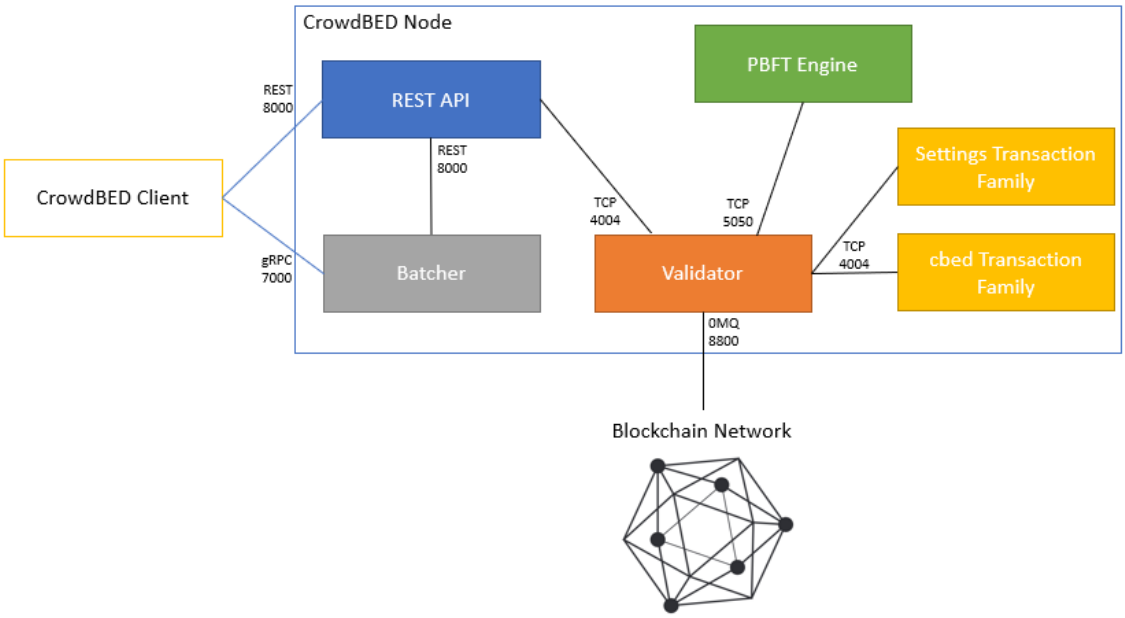

Figure 3.10: CrowdBED Overview

Additionally a single transaction processor was created for the CrowdBED needs. The transaction family was named cbed.

3.8.2 CrowdBED Consensus

For the CrowdBED implementation PBFT consensus was chosen. The main reason it is bevcause it is a Byzantine Fault tolerance, meaning it guarantees liveness and safety in of the network even when some portion of the network is faulty or malicious. Additionally the blockchain wont fork meaning there wont be two different versions of the ledger on different set of nodes. Additionally it does not depend on the hardware like the PoET consensus. But because of the consensus engine it is easy to swap different between different consensus algorithms making it easy to choose the one that fits the system administrator's requirements

Chapter 4

The Data on the Ledger

Contents

For Crowdsourcing to work, basic data types are needed to represent the whole process. The basic are the Task and the Submission. But because the workers work for reward it is important to have a data type User to represent the users that interact with the system. But because the Sawtooth ledger is public and in order to keep the submissions private additional data types were created called Lock and Proof. In order to close a task life a Validation transaction is needed. Finally in order to have a simple network architecture the wrapper data type was created. Each data type is explained in detain in this chapter.

As mentioned earlier, the data is stored in an addressable Merkle Radix tree. The addresses are calculated by the CrowdBED framework and will be explained for each entity. Additionally it's up to the developer to provide a method of serialization. For the serialization protocol buffers from Google were used to ensure correct serialization protocol across various client implementations.

4.1 User

User data type contains user information and metadata. The main component is the User's Public key. The user is created once and then the reputation and tokens are modified according to the user's behaviour.

User
String **Public Key**
Double **Reputation**
Double **Tokens**

Figure 4.1: User Structure

- **Public Key**: The public key of the User. The key is a sepc256k1 public key. The value is the hexadecimal representation of the key bytes.
- **Reputation**: The Current Reputation of the User.
- **Tokens**: The Tokens the user currently posses.

The address that we store an User is calculated the following way:

SHA512("user")[0:6]	SHA512(Public Key)[0:64]

Figure 4.2: User Address

Note the SHA512(x) function calculates the hexadecimal representation of the output of the SHA512 algorithm, and [a:b] notation returns the sub-array with indexes [a,b).
In order to calculate the address we get the namespace of the user data structure that is the first 6 characters of the hexadecimal representation of the SHA512 digest of the string "user", then the first 64 characters of the hexadecimal representation of the SHA512 digest of the field Public Key. With this encoding we can support 2^{512} users.
When constructing the User transaction the input and output field must contain the User address in order to be valid.

4.2 Task

Task data type contains a task information and metadata. The during the task creation the client should include the following values: Name, User, Deadline, Tokens, Code, Data, Description and Lock. The fields will be validated for correctness from the transaction family. The rest of the fields will be filled at the end of the validation.

Task
String **Name**
String **User**
Integer **Deadline**
Double **Tokens**
Boolean **Validated**
Byte[] **Answer**
Byte[] **Code**
Byte[] **Data**
Byte[] **Description**
String **Lock**
Boolean **Locked**

Figure 4.3: Task Structure

- **Name**: The Name of the task. Added on the task creation.
- **User**: The address on the ledger of the Requester. Added on the task creation.
- **Deadline**: The deadline of the task. Using the Unix time format. Added on the task creation.
- **Tokens**: The amount of tokens the requester will pay for the task completion. The amount is gathered from the requester at the task creation (will be subtracted from the requester's account) , and distributed to the workers after the task validation. Added on the task creation.

- **Validated**: A Boolean value to indicate if the task was validated. True if validated and false if not.
- **Answer**: The accepted answer for the task. Added after the task validation.
- **Code**: The script that will process the data. Currently there is only Python3 support. Added on the task creation.
- **Data**: The Data that will be processed. Added on the task creation.
- **Description**: The description of the task. Added on the task creation.
- **Lock**: A signed transaction header. After this transaction will be processed the submissions will be added to the ledger. This field is an important component to ensure the submissions will stay hidden until the end of the deadline.
- **Locked**: A Boolean variable that shows if the lock has applied to the task.

The address that we store a Task is calculated the following way:

SHA512(”task”)[0:6]	SHA512(User)[0:32]	SHA512(Name)[0:32]

Figure 4.4: Task Address

Note the SHA512(x) function calculates the hexadecimal representation of the output of the SHA512 algorithm, and [a:b] notation returns the sub-array with indexes [a,b).

In order to calculate the address we gate the namespace of the task data structure that is the first 6 characters of the hexadecimal representation of the SHA512 digest of the string ”task”, then the first 32 characters of the hexadecimal representation of the SHA512 digest of the field User, and finally the first 32 characters of the hexadecimal representation of the SHA512 digest of the field Name. With this encoding we can support 2^{256} users and each user can have 2^{256} different task names. Additionally using this encoding we can easily use the REST API to get the tasks by a specific user using the first 40 characters of the address (the namespace and the user field).

When constructing the Task transaction the input field must contain the User address and the Task address meanwhile the output filed should contain the Task address because if not the transaction will be invalid.

4.3 Submission

Submission data type contains a submission information and metadata. The during the submission creation the client should fill every field. The fields will be validated for correctness from the transaction family.

Submission
String **Task**
Byte[] **Data**
String **User**
String **Lock**
String **Proof**
String **Signature**

Figure 4.5: Submission Structure

- **Task**: The address of the task that the submission was created for.
- **Data**: The worker's answer to the task.
- **User**: The address of the user.
- **Lock**: The header of the lock transaction. The need for this field is to verify that the submission can be added to the ledger without leaking the worker's answer. It can be found in the Task's Locks field.
- **Proof**: The submissions proof address. In order to verify that the submission was computed before the deadline.
- **Signature**: The signature of the data. Signed by the worker. A crucial component to ensure the submission was created before the deadline.

The address that we store a Submission is calculated the following way:

SHA512("submission")[0:6]	SHA512(Task)[0:32]	SHA512(User)[0:32]

Figure 4.6: Submission Address

Note the SHA512(x) function calculates the hexadecimal representation of the output of the SHA512 algorithm, and [a:b] notation returns the sub-array with indexes [a,b).

In order to calculate the address we gate the namespace of the task data structure that is the first 6 characters of the hexadecimal representation of the SHA512 digest of the string ”submission”, then the first 32 characters of the hexadecimal representation of the SHA512 digest of the field Task, and finally the first 32 characters of the hexadecimal representation of the SHA512 digest of the field User. With this encoding we can support 2^{256} tasks and each task can have 2^{256} different submissions (from different users). Additionally using this encoding we can easily use the REST API to get the submissions for a specific task using the first 40 characters of the address (the namespace and the task field).

When constructing the Submission transaction the dependency field should contain the lock field of the submission, the input field must contain the user and proof addresses and the output field must contain the submission address and the user address otherwise the transaction will be invalid.

4.4 Lock

Lock data type contains a lock information and metadata. A lock is submitted to the network after the deadline of the task. The reason is the submissions are dependant to the lock transactions. After the lock is submitted after and only after the submission will be processed. The lock is created by the requester and sent to the batcher service where the batcher will keep the lock until the deadline. The transaction header is saved in the Locks field of the task. The fields will be validated for correctness from the transaction family.

Lock
String **Task**

Figure 4.7: Lock Structure

- **Task**: The address of the task that the lock was created for.

The locks are not saved in an address. It is only enough to include the transaction in a block on the ledger to fulfill it’s purpose. When constructing the lock transaction the input

field must contain the task address otherwise the transaction will be invalid.

4.5 Proof

Proof data type contains a proof information and metadata. A proof is created by the worker and submitted with the submission. Because the lock will be submitted after the deadline this means the submission will be processed after the deadline, thus we implemented the mechanism Proof of Submission that ensures that the submission was created and submitted by the specific worker before the deadline. The proof is created by the worker and submitted at the same time with the submission. During the proof creation every field should be filled. The fields will be validated by the transaction family.

Proof
String **Submission**
String **Task**
String **User**
String **Hignature**

Figure 4.8: Proof Structure

- **Submission**: The submission address that will need the proof in order to be added.
- **Task**: The task address that the submission was created.
- **User**: The address of the worker.
- **Hignature**: A hexadecimal representation of the SHA512 digest of the signature of the submission data.

The address that we store a Submission is calculated the following way:

SHA512(”proof”)[0:6]	SHA512(Task)[0:32]	SHA512(Submission)[0:32]

Figure 4.9: Proof Address

Note the SHA512(x) function calculates the hexadecimal representation of the output of the SHA512 algorithm, and [a:b] notation returns the subarray with indexes [a,b).

In order to calculate the adress we get the namespace of the task data structure that is the first 6 characters of the hexadecimal representation of the SHA512 digest of the string ”proof”, then the first 32 characters of the hexadecimal representation of the SHA512 digest of the field Task, and finally the first 32 characters of the hexadecimal representation of the SHA512 digest of the field Submission. With this encoding we can support 2^{256} tasks and each task can have 2^{256} different proofs (from different users). Additionally using this encoding we can easily use the REST API to get the submissions for a specific task using the first 40 characters of the address (the namespace and the task field).

When constructing the Proof transaction the input field must contain the user and task addresses and the output field must contain the proof and the user otherwise the transaction will be invalid.

4.6 Validation

Validation is the data type that is responsible to start the validation for a task. The validation can be submitted any moment after the deadline.

Validation
String **Task**

Figure 4.10: Validation Structure

- **Task**: Is the task address of the task that will do the validation.

When creating the transaction input and output fields must contain the task address, the submissions addresses, the proofs addresses and the user addresses for every submission for the task, otherwise the transaction will be invalid.

4.7 Wrapper

Because the protocol buffers can't identify the data type of the serialised data in the payload filed of the transaction. One way top process data was with the use of multiple transaction processors in order to handle each data type. But that will increase the amount of containers needed in order to maintain the network. A better approach is to create a Wrapper data type that will be used to execute commands in the transaction family.

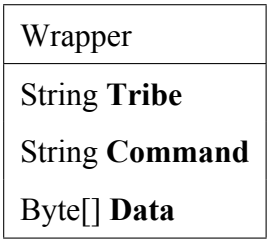

Figure 4.11: Wrapper Structure

- **Tribe**: The aim of this field is to determinate what type of data is serialized in the field Data. Can get only values from the following set of strings {"Task", "Submission", "Lock", "Proof", "User", "Validation"} according what type of data was serialized in the Data field.
- **Command**: This field is responsible on how will the data will be processed in the transaction family.
- **Data**: The serialized data of the transaction.

When constructing the transaction, the fields of the transaction are the same as explained previously for every data type.

Chapter 5

CrowdBED Transaction Family

Contents

Transaction families are the logic of the Sawtooth Ledger. They are responsible for processing every transaction and update the global state. When a transaction is applied the transaction family gets as arguments the transaction and the context (the state of the ledger). A transaction to be valid it must be executed from every peer in the network and be accepted after consensus is reached with the PBFT algorithm. In this chapter will be discussed how the transaction family of CrowdBED operates.

Before implementing the CrowdBED some technical details must be defined like the FamilyName, FamilyVersion and the Namespaces of the transaction family. The FamilyName is ”cbed”, the FamilyVersion is ”1.0”. Because cbed transaction family handles multiple data types and each is saved in a different namespace the following namespaces are used:

- Namespace **366785**: Is the namespace for Tasks and is calculated using the calculation SHA512(”task”)[0:6].
- Namespace **5e34c8**: Is the namespace for Submissions and is calculated using the calculation SHA512(”submission”)[0:6].
- Namespace **d570ab**: Is the namespace for Locks and is calculated using the calculation SHA512(”lock”)[0:6].
- Namespace **2178da**: Is the namespace for Proofs and is calculated using the calculation SHA512(”proof”)[0:6].
- Namespace **b14361**: Is the namespace for Users and is calculated using the calculation SHA512(”user”)[0:6].

5.1 When a Transaction Arrives

As stated each transaction sent to cbed Transaction Family is serialized using the Wrapper Data type. The first thing cbed does is to get the transaction payload and unmarshal the byte array using the proto.Unmarshal() method. After the wrapper is deserialised, depending what value is set in the Tribe field the Transaction Family will act. If non of the predetermined values are set, the transaction is invalid.

5.2 New User Appears

When a User Registration is issued the following procedure takes place:

Algorithm 1: User Registration

Data: TransactionHeader, User, Context

Result: A new User is Registered

if *!UserExists(TransactionHeader.PublicKey)* **then**

    InitializeReputation(User);

    Context.SetAddress(UserAddress(User), User);

else

    error(”InvalidTransaction”);

end

In order to keep anonymity on the network the only requirement is to not have a registered account. The reputation is initialized and the user is registered. Note that in the current implementation Users define how many tokens they have. After the process is complete the user is registered in the ledger and the data about him is stored in the merkle radix tree.

5.3 Task Creation

Tasks contains the task information by the requester. When a Task Creation is initiated the following procedure takes place in the transaction family:

Algorithm 2: Task Creation

```
Data: TransactionHeader, Task, Context
Result: A new Task is Added
if UserExists(TransactionHeader.PublicKey) and
  Task.User==UserAddress(TransactionHeader.PublicKey) and !TaskExists(Task)
 then
    User=Context.getAddress(Task.User);
    if User.Tokens >Task.Tokens then
        User.Tokens=User.Tokens-Task.Tokens;
        Context.SetAddress(Task.User, User);
        Task.Validated=False;
        Task.Locked=False;
        Context.SetAddress(TaskAddress(Task), Task);
        Context.CreateEvent(”cbed/NewTask”, TaskAddess);
    else
        error(”InvalidTransaction”);
    end
else
    error(”InvalidTransaction”);
end
```

When a Task creation is proposed some checks must take place before the task is created in order to be valid. The transaction family will check if the user exists based on the public key that signed the transaction, if the user field in the task matches the expected user address and if the task doesn't already exists (a task is defined by the name and the user posting it), in case it exists the transaction is invalid. After that the Transaction Family checks if the requester has enough tokens to propose the task and adds it to the ledger. After the transaction will be added to a block and then to the blockchain, an event will broadcasted that a new task was created.

5.4 Locking the Task

As mentioned locks are responsible for keeping the Submission hidden until the deadline passes. They are submitted after the deadline and allow the submissions to be added to the ledger. When a lock is added the following procedure takes place:

Algorithm 3: Adding the Lock

```
Data: TransactionHeader, Lock, Context
Result: A Lock transaction is added
if TaskExists(Task) then
    Task=Context.getAddress(Lock.Task);
    if Task.User==UserAddress(TransactionHeader.PublicKey) and
     isAfterDeadline(Task.Deadline) and ¬ Task.Locked then
        Task.Locked=true;
        Context.setAddress(TaskAddress(Task), Task);
    else
        error(”InvalidTransaction”);
    end
else
    error(”InvalidTransaction”);
end
```

The transaction family ensures that task exists and that the lock was created by the requester who posted the task. Finally it is important that the lock is submitted after the deadline because if a lock is added before the deadline then the submissions that depend on it will become public and can then be stolen.

Batcher Service

The existence of locks means that the requester must manually submit the lock in order to get the submissions. In order to solve this tedious responsibility batcher service was implemented. It is an additional process on every node in the CrowdBED network. The main responsibility is to collect locks and when the deadline arrives, it will submit them to the ledger.

It is implemented in Golang and every lock is just a goroutine to keep it light as possible.

The lock submission happens a through gRPC. The protocol is that the requester requests the public key of the batcher in order to create the transaction header and then sends the lock transaction to the batcher to keep the lock until the deadline comes.

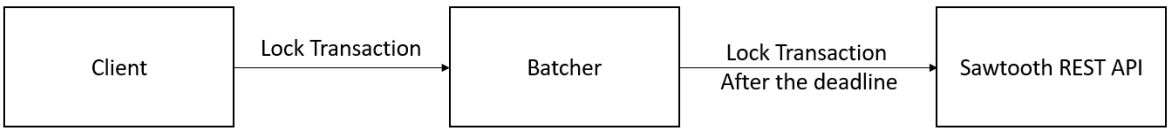

Figure 5.1: Batcher

But because if we have one batcher in case of a fault the submissions wont appear at the ledger. To solve this issue multiple batchers are created in order to keep the service robust. In the current implementation every Node in the CrowdBED network has a Batcher proccess running on port 7000. It is a lightweight solution because the amount of locks that will be held are as the number of unfinished tasks, each lock is held in an goroutine and it is in a blocking state until the deadline arrives. So a requester just needs to send the Lock transaction to all nodes in the network to keep it until the deadline. The IP of the nodes can be easily found with the use of REST API provided by the core Sawtooth implementation. The only lock that will be added will be the first one because it will change the state of the Task to locked, thus reducing the amount of transactions on the ledger.

Finally after submitting the lock transaction, after some time the batcher service generates the validation transaction and submits it to the ledger.

5.5 Adding the Proof

The Proof transaction is a way to ensure the specific worker calculated the data of his submission before the deadline. When a proof is added the following procedure takes place:

Algorithm 4: Adding the Proof

```
Data: TransactionHeader, Proof, Context
Result: A Proof is added
if UserExists(TransactionHeader.PublicKey) and TaskExists(Proof.Task) and
  SubmissionAdress(Proof.Task,
  TransactionHeader.PublicKey)==Proof.Submission then
    Task=Context.getAddress(Proof.Task);
    if ¬ Task.Locked then
      Context.setAdress(ProofAddress(Proof), Proof);
    else
      error("InvalidTransaction");
    end
else
    error("InvalidTransaction");
end
```

When a Proof is added the transaction family checks if the user is registered, the task exists and if the submission is for that task is the specific for that user. After that checks if it is submitted before the deadline in order to ensure that the submission was generated before the deadline. When everything is alright the proof is saved on the ledger.

5.6 Submitting Answers

The submission is the answer of the worker for a task. When a submission is submitted the following procedure takes place:

Algorithm 5: Adding a Submission

```
Data: TransactionHeader, Submission, Context
Result: A Submission is added
if UserExists(TransactionHeader.PublicKey) and TaskExists(Submission.Task) and
  !SubmissionExists(Submission) and ProofExists(Submission.Proof) then
    Task=Context.getAddress(Submission.Task);
    Proof=Context.getAddress(Submission.Proof);
    if Task.Locked and ProofOfSubmission(Proof, Submission,
      TransactionHeader.PublicKey) then
        Context.setAdress(SubmissionAddress(Submission), Submission);
    else
        error(”InvalidTransaction”);
    end
else
    error(”InvalidTransaction”);
end
```

When receiving a submission the transaction family checks if the Task, User and Proof exist. Aditionally checks that the submission doesn't exist in order to ensure one submission is added per task for every user. After that we check if the submission is after the deadline and if passes the proof of submission. If everything is alright then the submission is placed.

Proof of Submission

Algorithm 6: Proof of Submission

```
Data: Proof, Submission, PublicKey
Result: True if the proof holds, False otherwise
if VerifySignature(Submission.Signature, Submission.Data, PublicKey) and
  SHA512(Submission.Signature)==Proof.Hignature then
    return True;
else
    return False;
end
```

It is a simple way to ensure that a user created the specific submission before the deadline. There is a small window between the deadline and the validation we must ensure that when

submissions go public someone wont copy and submit them as their own submission. So there is a need to prove that a worker submitted before the deadline.

One approach is for proof to use the digest of the submission data and submit it as proof. This can be easily exploited because a malicious user can get this digest generate a proof with this digest and submit the proof. When this specific submission becomes public the malicious will steal the submission and submit it before the deadline. Other approach is to use the signature but this doesn't make much difference with the previous case because the public key is displayed in the transaction header and the malicious user can just decode the digest generate a proof and sign the digest with his key and then steal the submission in the deadline-validation window.

Given that every signature is distinct, that cryptographic hash functions (in this case SHA512) are one way functions. When publishing the digest of the signature it is impossible for someone to get the digest of the submission data. Because every proof for the same result will be unique for every worker.

Suppose some the answer for a task is D and two different workers computed the same result. In order to get the signature first we need to hash the data and then encrypt it with the private key. We know that for every user the private/public key pair is unique. That means:

$$Encrypt(SHA512(D), privKey_x)! = Encrypt(SHA512(D), privKey_y)$$

Finally thanks to the collision and pre-image resistance resistance it is almost impossible that:

$$SHA512(Encrypt(SHA512(D), privKey_x)) = SHA512(Encrypt(SHA512(D), privKey_y))$$

Meaning it is almost impossible to break Proof of Submission and can easily be used to prove that a specific worker submitted a specific submission and it is impossible for a malicious worker to steal other worker's work. Meaning it can provide the possibility for validation to happen even long time after the deadline period.

5.7 Validation

The validation is the final stage of a task's life. It is when submissions are validated, users rewarded or punished and the answer is chosen. It is the most Important operation in the Transaction Family because it is responsible for the reputation changes, token distribution

and a task's answer. Having a correct validation procedure is crucial for the CrowdBED's goal. Assuming that the tasks are deterministic the following procedure takes place:

Algorithm 7: Validation of a Task

```
Data: TaskAddress, Context
Result: Validate the Submissions of a Task
if AddressExists(TaskAddress) then
    Task=Context.getAddress(TaskAddress);
    if isAfterDeadline(Task.Deadline+SubmissionWindow) then
        Proofs[]=Context.getAddresses(ProofPrefix(Task));
        Users, Submissions=GetUsersAndSubsFromProofs(Proofs);
        Answer, HowMany=FindAnswer(Submissions, Users);
        if HowMany=0 then
            Requester=Context.getAddress(Task.User);
            Requester.Tokens+=Task.Tokens;
            Context.setAddress(UserAdress(Requester), Requester);
        else
            RewardAndPunishment(Users, Submissions);
        end
        Task.Answer=Answer;
        Task.Validated=True;
        context.setAddress(TaskAddress(Task), Task);
    else
        error("InvalidTransaction");
    end
else
    error("InvalidTransaction");
end
```

Firstly the transaction family checks if the task exists. Then checks happens after the deadline plus some small submission window, this window is the time when submissions are added to the ledger and in order to proceed correctly with validation. Thanks to Proof of Submission it is impossible to add more submissions than the ones that have proof added before the deadline, and proofs after the deadline are not accepted. Meaning the window can

be indefinitely long.

Other approach would be to add submission dependencies to the validation transaction header, meaning the validation will start the moment all submissions are added. But no one can guarantee that a worker will add the submission although he added the proof resulting in a deadlock.

After that all the proofs for the task are collected using the common prefix of the addresses (40 first characters) and after using the proof data we acquire the workers who submitted and the submissions. It is important to use proofs and not submissions because as stated previously there is no guarantee that if a proof exists then the submission exists meaning there may be some workers that didn't follow the submission protocol.

After gathering the workers and the submissions everything is ready for the validation. Currently there is simple procedure where the answer with the most occurrences is chosen. The data inside the submissions are passed through SHA512 hash function and then compare their digests. If there are no submissions the tokens of the task return to the requester. Otherwise iterating through every worker and checking if his submission data is the same as the answer. If it is the user gets reward and reputation increase, if not or the submission is missing then the worker gets punished. In the current implementation the tokens are divided evenly among the workers that gave the correct answer and increase/decrease the reputation by 1 if the answer is correct or missing or wrong.

The way the final answer is chosen and how reward and punishment can be easily updated according to system administrator's needs for a better reputation/reward scheme.

Chapter 6

CrowdBED Client

Contents

Here will be explained how a CrowdBED client should be implemented in order to interact with the Network. A client is expected to do the following operations:

- Generate and register a new user.
- Create a Task.
- Submit a Submission.
- Read from the Ledger.

The current implementation is implemented as a Command Line Interface and serves as an example how a client can be implemented. Because data is serialized using Protocol Buffers gives the opportunity for developers to implement a Client using their preferred language that supports protocol buffers and gRPC.

6.1 Generating a User

Before any user can interact with the ledger he need to create an account on the ledger in order to keep the reputation and token amount of the user.

Algorithm 8: Creation of a User

Result: User Creation on the Client Side

```
privateKey=GenerateSecp256k1Key();
SaveLocally(privateKey);
User=getUser(privateKey);
Transaction=GetUserTransaction(User);
Batch=CreateBatch(Transaction);
SendBatchToLedger(Batch);
```

The user creation is pretty simple. Anyone can generate a Secp256k1 key pair and use the public key as Identification for the ledger. After that the key is saved locally, because without you can't do anything on the ledger with this user. Then the transaction is created and afterwards the batch is generated and sent.

User Transaction Header
Batcher Public Key=User Public Key
Dependencies=[]
Family Name="cbed"
Family Version="1.0"
Inputs={UserAddress}
Outputs={UserAddress}
Payload SHA512=SerializedUser
Signer Public Key=User Public Key

Figure 6.1: User Transaction Header Structure

In order for the transaction to be processed correctly the user address should be added in the inputs and outputs section, if it is not added the transaction will be invalid the moment when it will try to access this address.

In the current implementation the user defines how much tokens he has in possession.

6.2 Requester

A requester is the user that creates and adds the task to the ledger.

Algorithm 9: Creation of a Task

Result: Task Creation on the Client Side

```
name=getTaskName();
privateKey=readPrivateKey(keyLocation);
lock=getTaskLock(name, privatekey.getPublicKey());
description=getTaskDescription();
tokens=getTaskTokens();
code=getTaskCode();
data=getTaskData();
task=getTask(pricateKey.getPublicKey(), name, data, code, task, tokens, lock);
Transaction=GetTaskTransaction(task);
Batch=CreateBatch(Transaction);
SendBatchToLedger(Batch);
Transaction=GetTaskTransaction(lock);
Batch=CreateBatch(Transaction);
IPs=GetPeerIPs();
foreach batcherIP ∈ IPs do
    SendBatchToBatcher(Batch, batcherIP);
end
```

The requester must undergo the following procedure. He must first get the metadata of the task: name, description, code, input data, tokens and requester's public key. After getting the task name and the public key he can generate the lock transaction for the task. Then the task transaction is generated and submitted directly to the ledger.

The Lock transaction replicated and sent to every peer in the network. Specifically to the batcher process that runs on the peer. The reason is that if the lock transaction is lost then the submissions will be lost. Thus fault tolerance is crucial in the lock mechanism, and because the lock processing in the transaction family is designed in a way that at least one

lock is submitted we get a high degree of fault tolerance. Even if the lock transaction is lost the requester can recreate the lock using the public key and the task name and send it to the ledger after the deadline in order to get the submissions unlocked.

Task Transaction Header
Batcher Public Key=User Public Key
Dependencies=[]
Family Name=”cbed”
Family Version=”1.0”
Inputs={UserAddress, TaskAdress}
Outputs={UserAddress, TaskAdress}
Payload SHA512=SerializedTask
Signer Public Key=User Public Key

Figure 6.2: Task Transaction Header Structure

When creating the task transaction header the client must add the user and task addresses in the input and the output fields. If they are not specified the task transaction will fail.

Lock Transaction Header
Batcher Public Key=User Public Key
Dependencies=[]
Family Name=”cbed”
Family Version=”1.0”
Inputs={TaskAdress}
Outputs={TaskAdress}
Payload SHA512=SerializedLock
Signer Public Key=User Public Key

Figure 6.3: Lock Transaction Header Structure

When creating the task transaction header the client must add the task address in the input and the output field. If they are not specified the task transaction will fail.

6.3 Worker

The worker is the user that submits the submission for a task.

Algorithm 10: Creation of a Submission

Result: Submission Creation on the Client Side

```
taskAdress=getTaskAdress();
privateKey=readPrivateKey(keyLocation);
task=getTaskData(TaskAdress);
answer=execute(task);
signature=sign(answer, privateKey);
hignature=SHA512(signature);
proof=getProof(hignature, taskAdress, privateKey.getPublicKey());
submission=getSubmission(answer, privateKey.getPublicKey(), task,
  proofAddress);
Transaction=GetProofTransaction(proof);
Batch=CreateBatch(Transaction);
SendBatchToLedger(Batch);
Transaction=GetSubmissionTransaction(submission);
Batch=CreateBatch(Transaction);
SendBatchToLedger(Batch);
```

The worker does the following procedure in order to submit his answer to the ledger. It is only required to have the task address for the worker to access the task. Before doing anything else the worker reads the private key from the key location, because without it he can't submit. After that he requests the task from the ledger and downloads the task. After that executes the task code with input the data provided by the requester and takes the answer. Then the client must create the proof for the submission. First he signs the data and then gets the digest after applying the SHA512 algorithm. After that he generates the submission and proof transactions and submits them directly to the ledger. The whole procedure must take place before the deadline because the proof wont be accepted.

Submission Transaction Header
Batcher Public Key=User Public Key
Dependencies={Task.Lock}
Family Name=”cbed”
Family Version=”1.0”
Inputs={TaskAdress, SubmissionAdress, ProofAdress}
Outputs={TaskAdress, SubmissionAdress, ProofAdress}
Payload SHA512=SerializedSubmission
Signer Public Key=User Public Key

Figure 6.4: Submission Transaction Header Structure

In order for the submission to be valid the input and output field on the transaction header must include task, proof and submission addresses. Additionally it is important that the dependencies field ontain the task lock header (that can be found in the task lock field). The reason is that without the dependency the submission will be shown before the deadline. The client can submit anytime the transaction before the validation.

Proof Transaction Header
Batcher Public Key=User Public Key
Dependencies={}
Family Name=”cbed”
Family Version=”1.0”
Inputs={TaskAddress, ProofAddress, UserAddress}
Outputs={TaskAddress, ProofAddress, UserAddress}
Payload SHA512=SerializedProof
Signer Public Key=User Public Key

Figure 6.5: Proof Transaction Header Structure

For a proof to be accepted the transaction header of a transaction must contain the proof, task and user addresses. Otherwise the transaction will be invalid. Note that it is important

to submit the proof before the deadline.

Automating the Submissions

In the current implementation a worker has the option to start the client in background mode. When a task is added to the ledger an event is created and broadcasted to the subscribers. Meaning that a client can subscribe for task creations and be be in a blocked state until a task is broadcast ed. After a task is added to the ledger the worker will unblock and automatically get the task, calculate the answer and submit it to the ledger in a SETI@Home fashion.

6.4 Validation

The validation is the transaction that initiates the transaction. Currently the validation transaction is generated in the batcher service.

Algorithm 11: Creation of a Validation

Result: Validation Creation on the Batcher Side

```
taskAdress=getTaskAdress();
proofprefix=getTaskProofs(taskAddress);
proofs=getProofs(proofPrefix);
useraddresses, submissionaddresses, proofaddresses=getAddresses(proofs);
validation=getValidation(getTaskAddress);
Transaction=getValidationTransaction(validation, useraddresses,
  submissionaddresses, proofaddresses);
Batch=CreateBatch(Transaction);
SendBatchToLedger(Batch);
```

The procedure for a validation transaction to be created is the following. Knowing the task address after the deadline the client requests from the ledger the proof list and gets the proof, user and submission addresses. These addresses are required for the transaction family to validate the submissions. After that the validation transaction is generated and sent to the ledger. The validation must be submitted after the deadline, the reason is that the submissions must be added before the validation.

Validation Transaction Header
Batcher Public Key=User Public Key
Dependencies={}
Family Name="cbed"
Family Version="1.0"
Inputs={TaskAddress, RequesterAddress, $ProofAddress_1, \cdots, ProofAddress_n$, $WorkerAddress_1, \cdots, WorkerAddress_n, SubmissionAddress_1, \cdots, SubmissionAddress_n$}
Outputs={TaskAddress, RequesterAddress, $ProofAddress_1, \cdots, ProofAddress_n$, $WorkerAddress_1, \cdots, WorkerAddress_n, SubmissionAddress_1, \cdots, SubmissionAddress_n$}
Payload SHA512=SerializedValidation
Signer Public Key=User Public Key

Figure 6.6: Validation Transaction Header Structure

The transaction header must include the addresses of the requester, the task , the workers who submitted, the submissions and the proofs. Without the above addresses the validation won't be able to complete.

Chapter 7

Implementation

Contents

In this chapter we will discuss how we implemented CrowdBED. We will talk about the project structure (the full code implementation will be in the Appendix), how the current client operates and how to start a network simulation and how to start a real network using different machines fore the implementation.

7.1 CrowdBED Project Structure

A brief explanation of the CrowdBED project structure:

CrowdBED: The root directory

- **Batcher**: This folder contains the batcher implementation
- **Client**: This folder contains the client implementation and test scripts
- **CrowdBED API**: This folder is the CrowdBED library
  - **batcherrpc**: Batcher gRPC configuration
  - **blockchain**: Functions that interact with the blockchain
  - crowdsourcing
    - **Submission**: Functions for Submission and Proof Data Types
    - **Task**: Functions for Task and Lock Data Types
  - **hashtools**: Hashing Tools
  - reputation
    - **User**: Functions for User Data Type
    - **Validation**: Functions for Validation Data Type
  - **Tribes**: Proto file that defines the CrowdBED data types
  - **unlock**:Tools for key managment
- **Transaction Family**: This folder contains the cbed TF implementation
- **DockerNetworks**: Docker Network simulations
  - **SingleNode.yaml**: Single Node simulation with Devmode consensus
  - **PBFT-Network.yaml**: Network simulation with PBFT consensus
- **Node**: PBFT CrowdBED node containers
  - **FirstNode.yaml**: The first node of the network
  - **OtherNodes.yaml**: The other nodes in the network
- **Install.sh**: Installation script
- **makefile**: make commands for compilation and deployment automation

7.2 CrowdBED Client

The CrowdBED client is implemented as a Command Line Interface using Golang. The CLI works with arguments and supports the following operations:

- Register User
- Add Task
- Add Submission
- Work
- List
  - Submissions of a Task
  - Tasks
  - Users

The commands follow the following pattern:

```
./Client -key=<key location> -IP=<IP of one of the peers in the
   ↪ CrowdBED network> -tribe=<The command the client will follow>
```

After the Tribe there will be a tail of data depending on the Tribe argument the client will execute. The details based how it is structured will be explained in the subsections.

7.2.1 Register User

To register a user the client must run the client process with the following way

```
./Client -key=<Key to be saved or use location> -IP=<IP of one of the
   ↪ validators> -tribe=User Register
```

the key argument will go to the location mentioned if it contains a key then it will use it for user registration (meaning that a user can use a key that was used in the past) otherwise a new key will be generated. An example will be:

```
./Client -key=adamos.pem -IP=127.0.0.1 -tribe=User Register
```

7.2.2 Add Task

To add a task the client must run the client process with the following way:

```
./Client -key=<key location> -IP=<IP of one of the validators> -tribe=
   ↪ Task Add <Name of the task> <Tokens> <The python3 script> <input
   ↪ file> <task description>
```

The key argument finds the key of the requester, it is important that the user is registered on the CrowdBED network. The IP could be any IP of a peer in the CrowdBED network. Then after tribe argument ”Task Add” must written and after that the name of the task, the amount of tokens the requester wishes to pay, the location of the python3 script, the location of the input file and the location of the task description. An example:

```
./Client -key=adamos.pem -IP=127.0.0.1 -tribe=Task Add Sum 42.0 Sum/sum
   ↪ .py Sum/input.txt Sum/description.txt
```

In order to show the simplicity of the task creation the following code is the code of the task:

```
a=input()
a=a.split()
sum=0
for x in a:
    sum+=int(x)
print(sum)
```

and the input data:

```
45 67 98 103
```

It is important that the code is deterministic or has a very high probability that it will produce the same result every time, because of the freedom of answer format the answers are validated based on the digest.

7.2.3 Add Submission

To add a submission the command should follow the following the following pattern:

```
./Client -malicious=<To be or not to be> -key=<Key location> -IP=<IP of
   ↪  one of the validators> -tribe=Submssion Add <Task Address>
```

The key argument finds the key of the worker, it is important that the user is registered on the CrowdBED network. The IP could be any IP of a peer in the CrowdBED network. Then after tribe argument ”Submission Add” the task address must be added. Then the Client will download the task script/data and run it locally and send the answer to the ledger. The malicious argument is just a boolean argument that if it set true then it will send ”Hello World” as an answer to the ledger (it was used to test implementation correctness). An example run is the following:

```
./Client -malicious=False -key=adamos.pem -IP=127.0.0.1 -tribe=
    ↪ Submission Add 3667854231
    ↪ eaaf9704c81d700bb4b7a4f1b78a6189278b21a4ddf3ca77f10ac8182939
```

The address can be found from the List command. Additionally the client can't check if the code that is running, if it is malicious or because of the halting problem if it will stop. But possible solutions would be to add a time limit and run the client inside a container.

7.2.4 Work

If a worker doesn't want do manually every task, an automated solution was implemented. In a SETI@home fashion the client has the ability to run in the background and do automatically every task is added to the ledger. It is possible with the Sawtooth events that the client subscribes to. In order to do this the following pattern should be written:

```
./Client -key=<Key location> -IP=<IP of one of the validators> -tribe=
    ↪ Work
```

An example run will be:

```
./Client -key=adamos.pem -IP=127.0.0.1 -tribe=Work
```

With this call the worker can leave the machine and have the CrowdBED client do all the job automatically. It waits for task creation events in a blocking state, after a task is created the event is broadcasted and the client creates a submission for that task (by following the submission procedure) and then waits for the next task.

7.2.5 List

It is the way anyone can read from the ledger and see at any moment. The pattern for listing tasks and users is:

```
./Client -key=<Key location> -IP=<IP of one of the validators> -tribe=
    ↪ List <Tasks or Users>
```

meanwhile for task submissions is:

```
./Client -key=<Key location> -IP=<IP of one of the validators> -tribe=
    ↪ List Submissions <Task Address>
```

with the following examples:

```
./Client -key=adamos.pem -IP=127.0.0.1 -tribe=List Tasks
./Client -key=adamos.pem -IP=127.0.0.1 -tribe=List Users
./Client -key=adamos.pem -IP=127.0.0.1 -tribe=List Submission
    ↪ 3667854231
    ↪ eaaf9704c81d700bb4b7a4f1b78a6189278b21a4ddf3ca77f10ac8182939
```

It is a user friendly way to present the ledger data. In the future a Graphical User Interface could be written for a better presentation.

```
There are 2 Tasks:

Task ID: 3667854920609148bf3828bd1b9fccc414298d6189278b21a4ddf3ca77f10ac8182939
Name: Sum
Description: Given a list of numbers compute the sum.
By: b14361c3f196f269a97778642ea8346f70901f97f473cd5fbb08837c923ac84b4d4040

Task ID: 366785f10f8cd4c199f25e60189467d4e5ed03c7f6559544cfd47b132532a2fe42483f
Name: Product
Description: Given a list of numbers compute the product.
By: b14361592c30683967ad7f315b5470c4f99601df385238a04f20df89dc380c7aa77016
```

Figure 7.1: Task List

```
There are 5 Submissions:

By b14361c3f196f269a97778642ea8346f70901f97f473cd5fbb08837c923ac84b4d4040:
313


By b143619f023952c22c79fc4d8d4e6ff9d454f152aef3c41af5de4f3e0b27bb293d6447:
313


By b143610c141146dab58ebd3bea5f632043fce2f02c366b72dd72f04ed7de5f53b3fc6b:
Hello World

By b143618c417b6e8be753ef7b4cbfe2a7cf944364c6c6aca028c26f4042f4eb95333673:
Hello World

By b14361592c30683967ad7f315b5470c4f99601df385238a04f20df89dc380c7aa77016:
313
```

Figure 7.2: Submissions List for Sum Task

```
There are 5 users:

User b14361c3f196f269a97778642ea8346f70901f97f473cd5fbb08837c923ac84b4d4040:
Reputation: 102.000000
Tokens: 55.000000

User b143610c141146dab58ebd3bea5f632043fce2f02c366b72dd72f04ed7de5f53b3fc6b:
Reputation: 98.000000
Tokens: 69.000000

User b14361592c30683967ad7f315b5470c4f99601df385238a04f20df89dc380c7aa77016:
Reputation: 102.000000
Tokens: 55.000000

User b143618c417b6e8be753ef7b4cbfe2a7cf944364c6c6aca028c26f4042f4eb95333673:
Reputation: 98.000000
Tokens: 69.000000

User b143619f023952c22c79fc4d8d4e6ff9d454f152aef3c41af5de4f3e0b27bb293d6447:
Reputation: 102.000000
Tokens: 97.000000
```

Figure 7.3: User List

7.3 Running Docker Simulations

There are two docker compose files that run CrowdBED simulations, the one is for one node for development and the other is for a PBFT network simulation with five nodes.

The single node deployment was used mainly for transaction family testing, that the logic is correct.

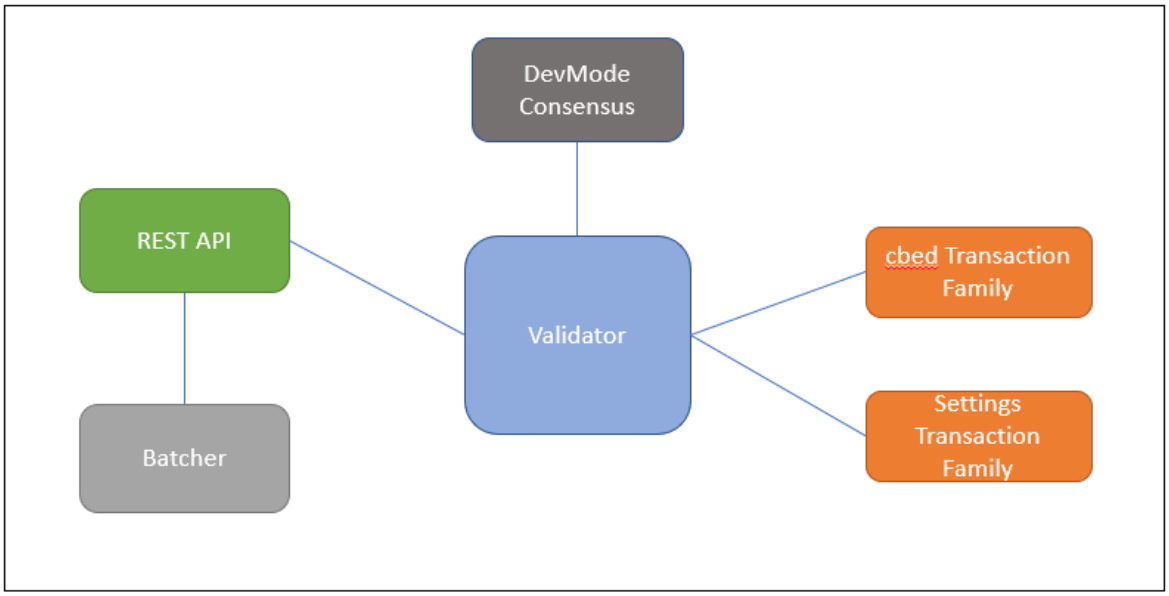

Figure 7.4: Single Node Deployment

In order to start a single node deployment for testing run the following command:

```
make OneNode
```

The above command builds the batcher and cbed transaction family images and starts the containers for a single node deployment.

The PBFT simulation network was used to run a simulation to check network correctness. In order to run a PBFT network simulation with five nodes run the following command:

```
make PBFT-Net
```

The above command builds the cbed Transaction Family and batcher and starts a PBFT network. During the execution PBFT logs will appear like view change and block proposals.

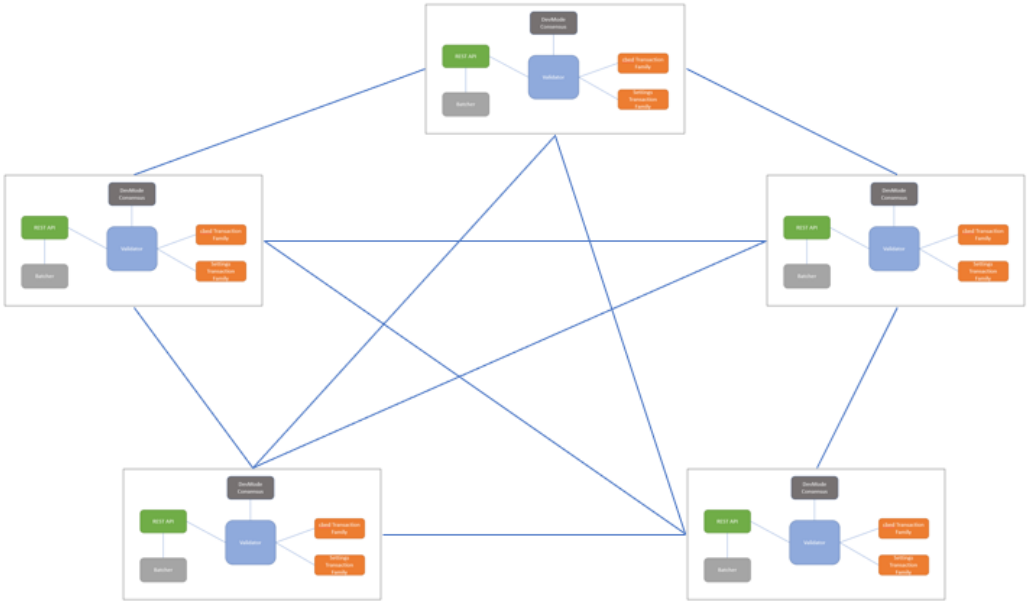

Figure 7.5: PBFT Network Deployment

7.4 Deployment on a Real Network

After testing on network simulations using docker containers a proof of concept execution is needed. For starting a real deployment it takes some additional steps. For a PBFT network to start there is a requirement of at least four nodes to create the initial network in order to ensure Byzantine Fault Tolerance.

For the network start, a node must create the genesis block that contains at least four public keys. The keys are generated on every node and are sent to the first node to start the network. Then every node starts and connects to all the nodes in the network. For the current network five nodes were deployed, the one was provided by Algolysis and the others are deployed in AWS.

Before deployment on the validator container configuration at the –endpoint field add the IP of the machine:

```
...
sawtooth-validator -vv \
    \bfseries --endpoint tcp://<insert IP here>:8800 \
    --bind component:tcp://eth0:4004 \
    --bind consensus:tcp://eth0:5050 \
    --bind network:tcp://eth0:8800 \
    --scheduler parallel \
    --peering static \
    --maximum-peer-connectivity 10000
...
```

The first node is responsible for the genesis node and must configure the PBFT with the kkeys of the other nodes so in the sawtooth.consensus.pbft.members field when a proposal would be made should contain the keys (note that it's not the location of the keys) in the FirstNode.yaml file:

```
...
sawset proposal create \
  -k /etc/sawtooth/keys/validator.priv \
  sawtooth.consensus.algorithm.name=pbft \
  sawtooth.consensus.algorithm.version=1.0 \
  sawtooth.consensus.pbft.members=\\['\"'$$(cat <Public key file 1>)
     ↪ '\"', ... , '\"'$$(cat <Public key file N>)'\"'\\]\
  sawtooth.publisher.max_batches_per_block=1200 \
  -o config.batch
...
```

then just by calling:

```
make GenesisNode
```

tjhe first node will be deployed. After that for every node to start in the OtherNodes.yaml file in the validator container configuration every other node that started in the network must be added to the –peers argument:

```
sawtooth-validator -vv \
    --endpoint tcp://validator:8800 \
    --bind component:tcp://eth0:4004 \
    --bind consensus:tcp://eth0:5050 \
    --bind network:tcp://eth0:8800 \
    --scheduler parallel \
    --peering static \
    --maximum-peer-connectivity 10000 \
    --peers tcp://<IP of Node 1>:8800 \
    ...
    --peers tcp://<IP of Node j-1>:8800
```

and deploy the node with:

```
make FatherNode
```

For more details we refer to the official Sawtooth documentation [13].

Chapter 8

Conclusion

Contents

8.1 Summary

In this thesis a prototype for crowdsourcing computational tasks on top of blockchain network was implemented. The aim was to provide reliability and decentralization to achieve full potential of crowdsourcing. The end result was the implementation of a blockchain network. A secure transaction family that implements the Master-Worker paradigm supporting a new user registration, task creation, answer submission and submission evaluation. On top of that

a Client example was implemented as a Command Line Interface for interacting with the blockchain. Additionally Sawtooth events were used to create enable the client to run on the background without the user to interfere with the whole process in a SETI@home fashion but at the same time providing easy to use client API for arbitrary submissions for a closer to Amazon Mechanical Turk usage. Finally, we tested the implementation's correctness on a real network on the Amazon Web Services cloud infrastructure using 5 EC2 t2.micro Virtual Machines, hence providing a proof of concept for a decentralized crowdsourcing platform.

8.2 Challenges

The whole experience was generally smooth. But some issues occurred.

Initially I had no experience with blockchains so I had to devote time to understand how blockchains work. After that I had to read countless of pages of Hyperledger documentation and communicate with the Linux Community to solve some questions either through StackOverflow or the Hyperledger chat. Mastering a Hyperledger framework takes a lot of time and dedication to know all the technology details. Sometimes I had to get my hands dirty and read the Sawtooth code in order to gain better understanding because there was no documentation or it was inconsistent. But through this process I developed some ideas how to contribute to the Hyperledger Sawtooth in the future.

The biggest challenge was how deal with deadlines. How can a worker show publicly that he computed the answer without leaking it to the other workers? The proof of submission had many failed versions and was exciting to find ways to break it until it becomes unbreakable.

Generally CrowdBED meets the requirments but has some weaknesses that need to be addressed and solved in the future to be more functional.

8.3 Future Work

8.3.1 Finite Submissions

Some master-worker implementations give the ability to the master to choose the workers that will solve the task or have a finite number of submissions. The task structure should be updated in order to specify the users that will be allowed to solve the task. In case the master

doesn't care who solves the task a pseudo-random procedure could take place for choosing randomly N online users that only they will be able to submit an answer.

But because Sawtooth requires the transactions to be deterministic the following approach could take place: Add new namespace that displays the online users, every user when wants to participate in crowdsouring could update it's status to online and then update it to offline when wants to stop. Thus giving a responcibility to users to maintain their online status. When a task is added the transaction family could gather the online users sort them based on the address, append them and at the end the task header, hash the long sequence with the SHA512 algorithm and use the digest as a seed to choose randomly N clients to solve the task. Because the determinism is defined by the state of blockchain and the transaction it doesn't break the Sawtooth requirments for transaction family.

8.3.2 Modular Validation

Because Sawtooth permits the sharing of namespaces, a good design practice is to create a separate Transaction Family for the validation giving the opportunity for trying different reputation schemes and validation algorithms by creating a container.

8.3.3 New Clients

Sawtooth supports stable SDKs for the following languages: Python, Go, Javascript and Rust. This means clients can be written in these languages and a web client for Amazon Mechanical Turk style tasks.

8.3.4 Auditors

Adding the auditor role to the model. Auditors are trusted entities that will validate the answers of the submissions. The auditors can be automated with the use of Hyperledger Avalon a trusted off chain computation environment. It could randomly solve tasks to determinate and the submission will be chosen by default and punish the users who did not submit the same answer.

8.3.5 Malicious Tasks

A requester can submit a Task that doesn't stop or that harms the clients machine. The never ending task can be detected due to the halting problem and the harm can be avoided by running the task in a container for isolating the task execution from the rest of the system. A time limit can be introduced to the task execution and can be adjusted by the task creator. The bigger the task time limit for execution the higher the price of the task. Thus higher reward for the workers.

8.3.6 Sybil Attack and Collusion

The biggest weakness of CrowdBED is a case of a sybil attack. Consider a case where a worker registers N accounts and then genuinely solves a task and submits the same correct answer from every account without computing it N times. Meaning that he will get N times more rewards. The current implementation does not handle this kind of attack since it treats every submission as a different user.

A solution was proposed in the "Algorithmic Mechanisms for Reliable Crowdsourcing Computation under Collusion" paper by Antonio Fernández Anta et al [27]. This solution requires the selection of specific set of workers to solve the task, and it was discussed in the previous section how it can be implemented.

Other approach would be to add a Central Authority for registering users. The Central Authority can be a decentralized identity like Hyperledger Aries or Indy in order to not allow a user to create multiple accounts. This will require communication between the Sawtooth network and the central authority when registering new user.

8.3.7 Storage

The current implementation stores the metadata of a submission and a task on the ledger. Meaning that each peer needs to keep a copy of the blockchain. Eventually the blockchain will not be scalable with data files like images, the requirments for huge data storages will be an issue for CrowdBED. A common approach is the use of a Distributed Hash Table like InterPlanetary File System (IPFS) [23]. Thus removing the data from the ledger and keeping only the hashses of the data. Eventually the transactions will have constant size and can scale freely.

8.3.8 Leaky Validators

A secret stops being a secret when you share it with at least one person. Meaning when a worker sends a submission to the network, even if the default implementation of a validator can't display the uncommitted transactions, a custom implementation of a malicious validator could take submission data and submit it as its own submission to the system.

For solving this issue there are two possible solution. By defying a window for submitting the submission after the deadline and before the validation, thanks to proof of submission it can't be leaked. However by preselcting the group of workers who will submit, thus reducing the probability the validator will be on the list and gain reward from an other workers work.

Bibliography

[1] Amazon mechanical turk. https://www.mturk.com/. Accessed: 2020-05-26.

[2] Coinlore. https://www.coinlore.com/all_coins. Accessed: 2020-05-26.

[3] Ethereum. https://ethereum.org/. Accessed: 2020-05-26.

[4] Fabric peers. https://hyperledger-fabric.readthedocs.io/en/release-2.0/peers/peers.html. Accessed: 2020-05-26.

[5] Folding@Home. https://foldingathome.org/. Accessed: 2020-05-26.

[6] grpc. https://grpc.io/. Accessed: 2020-05-26.

[7] Hyperledger. https://www.hyperledger.org/. Accessed: 2020-05-26.

[8] Hyperledger iroha. https://www.hyperledger.org/use/iroha. Accessed: 2020-05-26.

[9] Hyperledger iroha architecture. https://iroha.readthedocs.io/en/master/concepts_architecture/architecture.html. Accessed: 2020-05-26.

[10] Hyperledger sawtooth. https://www.hyperledger.org/use/sawtooth. Accessed: 2020-05-26.

[11] Libra. https://libra.org. Accessed: 2020-05-26.

[12] Protocol buffers. https://developers.google.com/protocol-buffers. Accessed: 2020-05-26.

[13] Sawtooth Administrators Guide. https://sawtooth.hyperledger.org/docs/core/releases/latest/sysadmin_guide.html. Accessed: 2020-05-26.

[14] Sawtooth architecture. https://sawtooth.hyperledger.org/docs/core/releases/latest/architecture.html. Accessed: 2020-05-26.

[15] Sawtooth events. https://sawtooth.hyperledger.org/docs/core/releases/latest /architecture/events_and_transactions_receipts.html. Accessed: 2020-05-26.

[16] Sawtooth global state. https://sawtooth.hyperledger.org/docs/core/releases/latest /architecture/global_state.html. Accessed: 2020-05-26.

[17] Sawtooth glossary. https://sawtooth.hyperledger.org/docs/core/releases/latest/glossary.html. Accessed: 2020-05-26.

[18] Sawtooth journal. https://sawtooth.hyperledger.org/docs/core/releases/latest/architecture/journal.html. Accessed: 2020-05-26.

[19] Sawtooth PBFT. https://sawtooth.hyperledger.org/docs/pbft/nightly/master/introduction-to-sawtooth-pbft.html. Accessed: 2020-05-26.

[20] Sawtooth rest api endpoint specification. https://sawtooth.hyperledger.org/docs/-core/releases/latest/rest_api/endpoint_specs.html. Accessed: 2020-05-26.

[21] Sawtooth transactions and batches. https://sawtooth.hyperledger.org/docs/core/releases/latest
/architecture/transactions_and_batches.html. Accessed: 2020-05-26.

[22] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, and et al. Hyperledger fabric. *Proceedings of the Thirteenth EuroSys Conference*, Apr 2018.

[23] Juan Benet. Ipfs - content addressed, versioned, p2p file system. 07 2014.

[24] Gurpriya Bhatia, Shubham Gupta, Alpana Dubey, and Ponnurangam Kumaraguru. Workerrep: Immutable reputation system for crowdsourcing platform based on blockchain. 05 2020.

[25] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[26] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[27] Antonio Fernández Anta, Chryssis Georgiou, Miguel A. Mosteiro, and Daniel Pareja. Algorithmic mechanisms for reliable crowdsourcing computation under collusion. *PLOS ONE*, 10(3):1–22, 03 2015.

[28] Eric Korpela, D. Werthimer, David Anderson, Jeff Cobb, and M. Leboisky. Seti@home-massively distributed computing for seti. *Computing in Science Engineering*, 3:78 – 83, 02 2001.

[29] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[30] M. Li, J. Weng, A. Yang, W. Lu, Y. Zhang, L. Hou, J. Liu, Y. Xiang, and R. H. Deng. Crowdbc: A blockchain-based decentralized framework for crowdsourcing. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1251–1266, 2019.

[31] Yuan Lu, Qiang Tang, and Guiling Wang. Zebralancer: Decentralized crowdsourcing of human knowledge atop open blockchain, 2018.

[32] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[33] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system.

[34] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, USA, 2016.

[35] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. page 305–320, 2014.

[36] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[37] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[38] Jeffrey Shneidman and David C. Parkes. Rationality and self-interest in peer to peer networks. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, pages 139–148, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[39] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems*. Pearson Education, 2013.

Appendices

A.1 Batcher

```
package main

import (
	"Tribes"
	pb "batcherrpc"
	"blockchain"
	"context"
	"log"
	"net"
	"os"
	"reputation/Validation"
	"time"
	"unlock"

	"github.com/hyperledger/sawtooth-sdk-go/protobuf/batch_pb2"

	"github.com/golang/protobuf/proto"
	"github.com/hyperledger/sawtooth-sdk-go/protobuf/transaction_pb2
	   ↪ "
	"github.com/hyperledger/sawtooth-sdk-go/signing"
	"google.golang.org/grpc"
)

type HolderServer struct {
	pb.UnimplementedHolderServer
	signer *signing.Signer
	IP string
}

func (srv *HolderServer) GetKey(ctx context.Context, nothing *pb.Empty)
```

```
    ↪  (*pb.Key, error) {
        return &pb.Key{Hex: srv.signer.GetPublicKey().AsHex()}, nil
}

func (srv *HolderServer) SendTransaction(ctx context.Context, trans *pb
    ↪ .Trans) (*pb.Empty, error) {
        go handleLock(trans.Serial, srv.signer, srv.IP)
        return &pb.Empty{}, nil
}

func handleLock(transBytes []byte, signer *signing.Signer, validatorIP
    ↪ string) {
        var batch batch_pb2.Batch
        err := proto.Unmarshal(transBytes, &batch)
        var transaction transaction_pb2.Transaction
        if len(batch.Transactions) != 1 {
               return
        }
        err = proto.Unmarshal(transBytes, batch.Transactions[0])
        if err != nil {
               log.Println("Error in Transaction Bytes", err)
               return
        }
        var wrap Tribes.Wrapper
        proto.Unmarshal(transaction.Payload, &wrap)
        var lock Tribes.Lock
        proto.Unmarshal(wrap.Data, &lock)
        time.Sleep(10 * time.Second)
        states := blockchain.GetStates(lock.Task, validatorIP)
        if len(states) != 1 {
               log.Println("No task found")
```

```
                return
        }
        var task Tribes.Task
        proto.Unmarshal(states[lock.Task], &task)
        time.Sleep(time.Until(time.Unix(task.Deadline, 0)))
        //time.Sleep(1 * time.Minute) //DeadLine
        blockchain.SendTransaction(transaction, signer, validatorIP)
        time.Sleep(2 * time.Second)
        validation := Tribes.Validation{
                Task: lock.Task,
        }
        blockchain.SendTransaction(Validation.Transaction(validation,
           ↪ signer, validatorIP), signer, validatorIP)
}
func main() {
        user := unlock.GetNewUser("key.pem")
        batcherServer := HolderServer{
                signer: user,
                IP: os.Args[1],
        }
        lis, err := net.Listen("tcp", ":7000")
        if err != nil {
                log.Panicln("failed to listen: %v", err)
        }
        grpcServer := grpc.NewServer()
        pb.RegisterHolderServer(grpcServer, &batcherServer)
        grpcServer.Serve(lis)
}
```

A.2 Client

```
package main


import (
	"Tribes"
	"blockchain"
	"crowdsourcing/Submission"
	"crowdsourcing/Task"
	"flag"
	"fmt"
	"hashtools"
	"io/ioutil"
	"log"
	"os"
	"os/exec"
	"reputation/User"
	"strconv"
	"time"
	"unlock"

	"github.com/hyperledger/sawtooth-sdk-go/protobuf/events_pb2"
)


func RunCode(code []byte, data []byte) ([]byte, error) {
	ioutil.WriteFile("task.py", code, 0666)
	ioutil.WriteFile("input.txt", data, 0666)
	cmd := exec.Command("python3", "task.py")
	cmd.Stdin, _ = os.Open("input.txt")
	cmd.Stdout, _ = os.Create("output.txt")
	err := cmd.Run()
	if err != nil {
```

```
                return nil, err
        }
        var ans []byte
        ans, err = ioutil.ReadFile("output.txt")
        if err != nil {
                return nil, err
        }
        os.Remove("task.py")
        os.Remove("input.txt")
        os.Remove("output.txt")
        return ans, nil
}

//Myth Hunter
func Worker(validatorIP string, keypath string, todos <-chan string) {
        signer := unlock.GetUser(keypath)
        for taskID := range todos {
                log.Printf("Started working for task %s\n", taskID)
                task := Task.GetTaskList(validatorIP)[taskID]
                lock := task.Lock
                ans, err := RunCode(task.Code, task.Data)
                if err != nil {
                        log.Println(err)
                        continue
                }
                submission := Tribes.Submission{
                        Task: taskID,
                        User: User.KeyAddr(signer.GetPublicKey().AsHex()),
                        Data: ans,
                        Lock: lock,
                }
```

```
                Submission.SendSubmission(submission, signer, validatorIP
                    ↪ )
                log.Printf("Submission sent for task %s\n", taskID)
        }
}

func WhatToDo(IP string, keypath string, tribe string) {
        validatorIP := "http://" + IP + ":8000"
        //batcherIP := []string{IP + ":7000"}
        batcherIP := []string{"95.216.219.236:7000", "34.236.159.44:7000
            ↪ ", "3.87.161.101:7000", "100.27.23.26:7000", "
            ↪ 3.83.51.184:7000"}
        //batcherIP := []string{IP + ":7000", IP + ":7001", IP +
            ↪ ":7002", IP + ":7003", IP + ":7004"}
        args := flag.Args()
        switch tribe {
        case "User":
                switch args[0] {
                case "Register":
                        signer := unlock.GetNewUser(keypath)
                        user := User.NewUser(signer.GetPublicKey().AsHex()
                            ↪ )
                        User.RegisterUser(user, signer, validatorIP)
                }
        case "Task":
                /*
                        Arg_0 what to do with the task.
                        Arg_1 name of the task
                        Arg_2 cost of task
                        Arg_3 codefile
                        Arg_4 datafile
```

```
        Arg_5 descriptionfile
*/
switch args[0] {
case "Add":
        signer := unlock.GetUser(keypath)
        task := Tribes.Task{
                Name: args[1],
                User: User.KeyAddr(signer.GetPublicKey().
                    ↪ AsHex()),
                Deadline: time.Now().Add(time.Minute).Unix
                    ↪ (),
        }
        var err error
        task.Tokens, err = strconv.ParseFloat(args[2], 64)
        if err != nil {
                log.Panicln(err)
        }
        task.Code, err = ioutil.ReadFile(args[3])
        if err != nil {
                log.Panicln(err)
        }
        task.Data, err = ioutil.ReadFile(args[4])
        if err != nil {
                log.Panicln(err)
        }
        task.Description, err = ioutil.ReadFile(args[5])
        if err != nil {
                log.Panicln(err)
        }
        Task.SendTask(task, signer, validatorIP, batcherIP
            ↪ )
```

```
                fmt.Print(Task.TaskAddr(task))
        }
    case "Submission":
        switch args[0] {
        case "Add":
                signer := unlock.GetUser(keypath)
                taskID := args[1]
                task := Task.GetTaskList(validatorIP)[taskID]
                lock := task.Lock
                ans, err := RunCode(task.Code, task.Data)
                if err != nil {
                        log.Panicln(err)
                }
                submission := Tribes.Submission{
                        Task: args[1],
                        User: User.KeyAddr(signer.GetPublicKey().
                            ↪ AsHex()),
                        Data: ans,
                        Lock: lock,
                }
                Submission.SendSubmission(submission, signer,
                    ↪ validatorIP)
        }
    case "Work":
        Filter := events_pb2.EventFilter{
                Key: "ID",
                MatchString: hashtools.Hexdigest("task")[:6] + ".*
                    ↪ ",
                FilterType: events_pb2.EventFilter_REGEX_ANY,
        }
        todos := make(chan string, 2)
```

```
            go blockchain.Listen(IP, Filter, "cbed/NewTask", todos)
            Worker(validatorIP, keypath, todos)
        case "List":
            switch args[0] {
            case "Submissions":
                subs := Submission.GetSubmissionList(args[1],
                    ↪ validatorIP)
                fmt.Printf("There are %d Submissions: \n\n", len(
                    ↪ subs))
                for _, sub := range subs {
                    fmt.Printf("By %s:\n", sub.User)
                    fmt.Printf("%s\n\n", sub.Data)
                }
            case "Tasks":
                tasks := Task.GetTaskList(validatorIP)
                fmt.Printf("There are %d Tasks: \n\n", len(tasks))
                for addr, task := range tasks {
                    fmt.Printf("Task ID: %s\n", addr)
                    fmt.Printf("Name: %s\n", task.Name)
                    fmt.Printf("Description: %s\n", task.
                        ↪ Description)
                    fmt.Printf("By: %s\n\n", task.User)
                }
            case "Users":
                users := User.GetUsers(validatorIP)
                fmt.Printf("There are %d users: \n\n", len(users))
                for addr, user := range users {
                    fmt.Printf("User %s:\n", addr)
                    fmt.Printf("Reputation: %f\n", user.
                        ↪ Reputation)
                    fmt.Printf("Tokens: %f\n\n", user.Tokens)
```

```
                        }
                }
        default:
                panic("What is this?\n")
        }
}
func main() {
        //signer := unlock.GetUser("key.pem")
        //validatorIP := "http://localhost:8008"
        var gui bool
        flag.BoolVar(&gui, "gui", false, "Use graphical user interface")
        var keypath string
        flag.StringVar(&keypath, "key", "key.pem", "User key")
        var IP string
        flag.StringVar(&IP, "IP", "localhost", "Validator IP to connect"
           ↪ )
        var tribe string
        flag.StringVar(&tribe, "tribe", "", "Type of command to iniate")
        flag.Parse()
        WhatToDo(IP, keypath, tribe)
}
```

A.3 Blockchain API

```
package blockchain


import (
        "batcherrpc"
        "bytes"
        "context"
        "encoding/base64"
        "encoding/hex"
        "encoding/json"
        "errors"
        "fmt"
        "io/ioutil"
        "log"
        "net/http"

        "github.com/pebbe/zmq4"
        "google.golang.org/grpc"

        "github.com/golang/protobuf/proto"
        "github.com/hyperledger/sawtooth-sdk-go/messaging"
        "github.com/hyperledger/sawtooth-sdk-go/protobuf/batch_pb2"
        "github.com/hyperledger/sawtooth-sdk-go/protobuf/
           ↪ client_event_pb2"
        "github.com/hyperledger/sawtooth-sdk-go/protobuf/events_pb2"
        "github.com/hyperledger/sawtooth-sdk-go/protobuf/transaction_pb2
           ↪ "
        "github.com/hyperledger/sawtooth-sdk-go/signing"
)

func CreateBatch(transaction transaction_pb2.Transaction, signer *
```

```
    ↪ signing.Signer) batch_pb2.Batch {
        TransactionHeaders := []string{transaction.GetHeaderSignature()}

        rawBatchHeader := batch_pb2.BatchHeader{
                SignerPublicKey: signer.GetPublicKey().AsHex(),
                TransactionIds: TransactionHeaders,
        }
        batchHeaderBytes, _ := proto.Marshal(&rawBatchHeader)

        signature := hex.EncodeToString(signer.Sign(batchHeaderBytes))
        batch := batch_pb2.Batch{
                Header: batchHeaderBytes,
                Transactions: []*transaction_pb2.Transaction{&transaction
                    ↪ },
                HeaderSignature: signature,
        }
        return batch
}
func SendTransaction(transaction transaction_pb2.Transaction, signer *
    ↪ signing.Signer, validatorIP string) {
        batch := CreateBatch(transaction, signer)
        sendBatch(batch, validatorIP)
}
func SendToBatcher(transaction transaction_pb2.Transaction, signer *
    ↪ signing.Signer, batcherIP string) {
        conn, err := grpc.Dial(batcherIP, grpc.WithInsecure())
        if err != nil {
                log.Panicln(err)
        }
        defer conn.Close()
        batch := CreateBatch(transaction, signer)
```

```
        batchBytes, _ := proto.Marshal(&batch)
        trans := batcherrpc.Trans{
                Serial: batchBytes,
        }
        client := batcherrpc.NewHolderClient(conn)
        _, err = client.SendTransaction(context.Background(), &trans)
        if err != nil {
                log.Panicln(err)
        }
}
func sendBatch(batch batch_pb2.Batch, validatorIP string) {
        rawBatchList := batch_pb2.BatchList{
                Batches: []*batch_pb2.Batch{&batch},
        }
        batchListBytes, _ := proto.Marshal(&rawBatchList)

        response, err := http.Post(
                validatorIP+"/batches",
                "application/octet-stream",
                bytes.NewBuffer(batchListBytes),
        )
        if err != nil {
                log.Panicf("Failed to send Transaction:\n%s\n%s\n",
                   ↪ response, err)
        }

}

func GetStates(prefix string, validatorIP string) map[string][]byte {
        response, err := http.Get(validatorIP + "/state?address=" +
           ↪ prefix)
```

```
        if err != nil {
                fmt.Println("Error in http, ", err)
                return nil
        }
        defer response.Body.Close()
        reponseBody, _ := ioutil.ReadAll(response.Body)
        var states map[string][]map[string]string
        json.Unmarshal(reponseBody, &states)
        retVal := make(map[string][]byte)
        for _, singTask := range states["data"] {
                decode, _ := base64.StdEncoding.DecodeString(singTask["
                    ↪ data"])
                retVal[singTask["address"]] = decode
        }
        return retVal
}

func Listen(IP string, Filter events_pb2.EventFilter, eventType string,
    ↪ todos chan<- string) error {
        zmq_context, err := zmq4.NewContext()
        // Error creating a ZMQ context
        if err != nil {
                return err
        }
        zmq_connection, err := messaging.NewConnection(
                zmq_context,
                zmq4.DEALER,
                "tcp://"+IP+":4004",
                false,
        )
        if err != nil {
```

```
        return err
    }
    filters := []*events_pb2.EventFilter{&Filter}
    my_identifier_subscription := events_pb2.EventSubscription{
        EventType: "cbed/NewTask",
        Filters: filters,
    }
    request := client_event_pb2.ClientEventsSubscribeRequest{
        Subscriptions: []*events_pb2.EventSubscription{
            &my_identifier_subscription,
        },
    }

    serialized_subscribe_request, err := proto.Marshal(&request)
    if err != nil {
        return err
    }

    /*
        Instead of 500 there should be validator_pb2.
           ↪ Message_CLIENT_EVENTS_SUBSCRIBE_REQUEST but
        I am getting compile error so I changed it to 500 (
           ↪ because it's a constant value)
        No I dea why the error
        by Adamos
    */
    corrId, err := zmq_connection.SendNewMsg(500,
       ↪ serialized_subscribe_request)

    if err != nil {
        return err
```

```
}
_, response, err := zmq_connection.RecvMsgWithId(corrId)
if err != nil {
        return err
}

events_subscribe_response := client_event_pb2.
   ↪ ClientEventsSubscribeResponse{}
err = proto.Unmarshal(response.Content, &
   ↪ events_subscribe_response)
if err != nil {
        return err
}
if events_subscribe_response.Status != client_event_pb2.
   ↪ ClientEventsSubscribeResponse_OK {
        return errors.New("Client subscription failed")
}

// Listen for events in an infinite loop
fmt.Println("Listening to events.")
for {
        _, message, err := zmq_connection.RecvMsg()
        if err != nil {
                return err
        }
        /*
                Instead of 504 there should be validator_pb2.
                   ↪ Message_CLIENT_EVENTS_SUBSCRIBE_REQUEST but
                I am getting compile error so I changed it to 504
                   ↪  (because it's a constant value)
                No I dea why the error
```

```
                by Adamos
        */
        if message.MessageType != 504 {
                continue
        }
        event_list := events_pb2.EventList{}
        err = proto.Unmarshal(message.Content, &event_list)
        if err != nil {
                return err
        }
        // Received following events from validator
        for _, event := range event_list.Events {
                // handle event here
                fmt.Printf("Task received: %s\n", (*event).
                    ↪ Attributes[0].Value)
                todos <- (*event).Attributes[0].Value
        }
}

// Unsubscribe from events
events_unsubscribe_request := client_event_pb2.
    ↪ ClientEventsUnsubscribeRequest{}
var serialized_unsubscribe_request []byte
serialized_unsubscribe_request, err = proto.Marshal(&
    ↪ events_unsubscribe_request)
if err != nil {
        return err
}
/*
        Instead of 502 there should be validator_pb2.
            ↪ Message_CLIENT_EVENTS_UNSUBSCRIBE_REQUEST but
```

```
            I am getting compile error so I changed it to 502 (
               ↪ because it's a constant value)
            No I dea why the error
            by Adamos
        */
        corrId, err = zmq_connection.SendNewMsg(502,
           ↪ serialized_unsubscribe_request)
        if err != nil {
            return err
        }
        _, unsubscribe_response, err := zmq_connection.RecvMsgWithId(
           ↪ corrId)
        if err != nil {
            return err
        }
        events_unsubscribe_response :=
            client_event_pb2.ClientEventsUnsubscribeResponse{}
        err = proto.Unmarshal(unsubscribe_response.Content, &
           ↪ events_unsubscribe_response)
        if err != nil {
            return err
        }
        if events_unsubscribe_response.Status !=
            client_event_pb2.ClientEventsUnsubscribeResponse_OK {
            return errors.New("Client couldn't unsubscribe
               ↪ successfully")
        }
        return nil
}
```

A.4 Task API

```
package Task

import (
	"Tribes"
	"blockchain"
	"encoding/hex"
	"hashtools"

	"github.com/golang/protobuf/proto"
	"github.com/hyperledger/sawtooth-sdk-go/protobuf/transaction_pb2
	   ↪ "
	"github.com/hyperledger/sawtooth-sdk-go/signing"
)

func GetTaskList(validatorIP string) map[string]Tribes.Task {
	states := blockchain.GetStates(hashtools.Hexdigest("task")[:6],
	   ↪ validatorIP)
	retVal := map[string]Tribes.Task{}
	for addr, trans := range states {
		var temp Tribes.Task
		proto.Unmarshal(trans, &temp)
		retVal[addr] = temp
	}
	return retVal
}

//
   ↪ ------------------------------------------------------------------------
   ↪
```

```
func WrapTask(task Tribes.Task) Tribes.Wrapper {
        data, _ := proto.Marshal(&task)
        wrap := Tribes.Wrapper{
                Tribe: "Task",
                Data: data,
        }
        return wrap
}

func TaskTransaction(task Tribes.Task, signer *signing.Signer)
   ↪ transaction_pb2.Transaction {
        wrap := WrapTask(task)
        payloadBytes, _ := proto.Marshal(&wrap)
        rawTransactionHeader := transaction_pb2.TransactionHeader{
                SignerPublicKey: signer.GetPublicKey().AsHex(),
                FamilyName: "cbed",
                FamilyVersion: "1.0",
                Dependencies: []string{},
                BatcherPublicKey: signer.GetPublicKey().AsHex(),
                Inputs: []string{task.User},
                Outputs: []string{TaskAddr(task), task.User},
                PayloadSha512: hashtools.HashData(payloadBytes),
        }
        transactionHeaderBytes, _ := proto.Marshal(&rawTransactionHeader
           ↪ )
        signature := hex.EncodeToString(signer.Sign(
           ↪ transactionHeaderBytes))
        transaction := transaction_pb2.Transaction{
                Header: transactionHeaderBytes,
                HeaderSignature: signature,
                Payload: payloadBytes,
```

```
        }
        return transaction
}

func TaskAddr(task Tribes.Task) string {
        ans := hashtools.Hexdigest("task")[:6] + hashtools.Hexdigest(
            ↪ task.User)[:32] + hashtools.Hexdigest(task.Name)[:32]
        return ans
}

/*
func DeadlinePassed(task Tribes.Task) bool {
        return !task.Deadline.Before(time.Now())
}*/

func getLock(task Tribes.Task, signer *signing.Signer) Tribes.Lock {
        lock := Tribes.Lock{
                Task: TaskAddr(task),
        }
        return lock
}

func SendTask(task Tribes.Task, signer *signing.Signer, validatorIP
    ↪ string, batcherIP []string) {
        lock := getLock(task, signer)
        for _, batcher := range batcherIP {
                lockTransaction := LockTransaction(lock, signer)
                task.Lock = lockTransaction.HeaderSignature
                blockchain.SendToBatcher(lockTransaction, signer, batcher
                    ↪ )
        }
```

```
        taskTransaction := TaskTransaction(task, signer)
        blockchain.SendTransaction(taskTransaction, signer, validatorIP)
}

//
   ↪ ------------------------------------------------------------------------
   ↪

func LockTransaction(lock Tribes.Lock, signer *signing.Signer)
   ↪ transaction_pb2.Transaction {
        wrap := WrapLock(lock)
        payloadBytes, _ := proto.Marshal(&wrap)
        rawTransactionHeader := transaction_pb2.TransactionHeader{
                SignerPublicKey: signer.GetPublicKey().AsHex(),
                FamilyName: "cbed",
                FamilyVersion: "1.0",
                Dependencies: []string{},
                BatcherPublicKey: signer.GetPublicKey().AsHex(),
                Inputs: []string{lock.GetTask()},
                Outputs: []string{LockAddr(lock), lock.GetTask()},
                PayloadSha512: hashtools.HashData(payloadBytes),
        }
        transactionHeaderBytes, _ := proto.Marshal(&rawTransactionHeader
           ↪ )
        signature := hex.EncodeToString(signer.Sign(
           ↪ transactionHeaderBytes))
        transaction := transaction_pb2.Transaction{
                Header: transactionHeaderBytes,
                HeaderSignature: signature,
                Payload: payloadBytes,
        }
```

```
        return transaction
}

func LockAddr(lock Tribes.Lock) string {
        ans := hashtools.Hexdigest("lock")[:6] + hashtools.Hexdigest(
           ↪ lock.Task)[:64]
        return ans
}

func WrapLock(lock Tribes.Lock) Tribes.Wrapper {
        data, _ := proto.Marshal(&lock)
        wrap := Tribes.Wrapper{
                Tribe: "Lock",
                Data: data,
        }
        return wrap
}
```

A.5 Submission API

```
package Submission


import (
        "Tribes"
        "blockchain"
        "encoding/hex"
        "fmt"
        "hashtools"


        "github.com/golang/protobuf/proto"
        "github.com/hyperledger/sawtooth-sdk-go/protobuf/transaction_pb2
           ↪ "
        "github.com/hyperledger/sawtooth-sdk-go/signing"
)


func ProofTransaction(proof Tribes.Proof, signer *signing.Signer)
   ↪ transaction_pb2.Transaction {
        wrap := WrapProof(proof)
        payloadBytes, _ := proto.Marshal(&wrap)
        rawTransactionHeader := transaction_pb2.TransactionHeader{
                SignerPublicKey: signer.GetPublicKey().AsHex(),
                FamilyName: "cbed",
                FamilyVersion: "1.0",
                Dependencies: []string{},
                BatcherPublicKey: signer.GetPublicKey().AsHex(),
                Inputs: []string{proof.GetUser(), proof.GetTask()},
                Outputs: []string{ProofAddr(proof), proof.GetUser()},
                PayloadSha512: hashtools.HashData(payloadBytes),
        }
        transactionHeaderBytes, _ := proto.Marshal(&rawTransactionHeader
```

```
            ↪ )
        signature := hex.EncodeToString(signer.Sign(
            ↪ transactionHeaderBytes))
        transaction := transaction_pb2.Transaction{
                Header: transactionHeaderBytes,
                HeaderSignature: signature,
                Payload: payloadBytes,
        }
        return transaction
}

func ProofAddr(proof Tribes.Proof) string {
        ans := hashtools.Hexdigest("proof")[:6] + hashtools.Hexdigest(
            ↪ proof.Task)[:32] + hashtools.Hexdigest(proof.Submission)
            ↪ [:32]
        return ans
}

func WrapProof(proof Tribes.Proof) Tribes.Wrapper {
        data, _ := proto.Marshal(&proof)
        wrap := Tribes.Wrapper{
                Tribe: "Proof",
                Data: data,
        }
        return wrap
}

//
    ↪ ------------------------------------------------------------------------
    ↪
```

```
func SubmissionTransaction(submission Tribes.Submission, signer *
    ↪ signing.Signer) transaction_pb2.Transaction {
        wrap := WrapSubmission(submission)
        payloadBytes, _ := proto.Marshal(&wrap)
        rawTransactionHeader := transaction_pb2.TransactionHeader{
                SignerPublicKey: signer.GetPublicKey().AsHex(),
                FamilyName: "cbed",
                FamilyVersion: "1.0",
                Dependencies: []string{submission.Lock},
                BatcherPublicKey: signer.GetPublicKey().AsHex(),
                Inputs: []string{submission.User, submission.Proof},
                Outputs: []string{SubmissionAddr(submission), submission.
                    ↪ GetUser()},
                PayloadSha512: hashtools.HashData(payloadBytes),
        }
        transactionHeaderBytes, _ := proto.Marshal(&rawTransactionHeader
            ↪ )
        signature := hex.EncodeToString(signer.Sign(
            ↪ transactionHeaderBytes))
        transaction := transaction_pb2.Transaction{
                Header: transactionHeaderBytes,
                HeaderSignature: signature,
                Payload: payloadBytes,
        }
        return transaction
}

func SubmissionAddr(submission Tribes.Submission) string {
        ans := hashtools.Hexdigest("submission")[:6] + hashtools.
            ↪ Hexdigest(submission.Task)[:32] + hashtools.Hexdigest(
            ↪ submission.User)[:32]
```

```
        return ans
}

func getProof(submission *Tribes.Submission, signer *signing.Signer)
    ↪ Tribes.Proof {
        proof := Tribes.Proof{
                Submission: SubmissionAddr((*submission)),
                Task: (*submission).Task,
                User: (*submission).User,
                Hignature: hashtools.Hexdigest(hex.EncodeToString(signer.
                    ↪ Sign([]byte((*submission).Data)))),
        }
        (*submission).Proof = ProofAddr(proof)
        return proof
}

func WrapSubmission(submission Tribes.Submission) Tribes.Wrapper {
        data, _ := proto.Marshal(&submission)
        wrap := Tribes.Wrapper{
                Tribe: "Submission",
                Data: data,
        }
        return wrap
}

func SendSubmission(submission Tribes.Submission, signer *signing.
    ↪ Signer, validatorIP string) {
        submission.Signature = hex.EncodeToString(signer.Sign([]byte(
            ↪ submission.Data)))
        proof := getProof(&submission, signer)
        proofTransaction := ProofTransaction(proof, signer)
```

```
        submissionTransaction := SubmissionTransaction(submission,
            ↪ signer)

        //fmt.Println("Depends on:", submission.Lock)
        blockchain.SendTransaction(proofTransaction, signer, validatorIP
            ↪ )
        blockchain.SendTransaction(submissionTransaction, signer,
            ↪ validatorIP)
}

func GetSubmissionList(task string, validatorIP string) map[string]
    ↪ Tribes.Submission {
        prefix := hashtools.Hexdigest("submission")[:6] + hashtools.
            ↪ Hexdigest(task)[:32]
        states := blockchain.GetStates(prefix, validatorIP)
        retVal := map[string]Tribes.Submission{}
        for addr, sub := range states {
                var temp Tribes.Submission
                proto.Unmarshal(sub, &temp)
                retVal[addr] = temp
        }
        return retVal
}
func GetProofList(task string, validatorIP string) map[string]Tribes.
    ↪ Proof {
        prefix := hashtools.Hexdigest("proof")[:6] + hashtools.Hexdigest
            ↪ (task)[:32]
        fmt.Println(prefix)
        states := blockchain.GetStates(prefix, validatorIP)
        retVal := map[string]Tribes.Proof{}
        for addr, proof := range states {
```

```
                var temp Tribes.Proof
                proto.Unmarshal(proof, &temp)
                retVal[addr] = temp
        }
        return retVal
}
```

A.6 User API

```
package User

import (
	"Tribes"
	"blockchain"
	"encoding/hex"
	"hashtools"

	"github.com/golang/protobuf/proto"
	"github.com/hyperledger/sawtooth-sdk-go/protobuf/transaction_pb2
	    ↪ "
	"github.com/hyperledger/sawtooth-sdk-go/signing"
)

func NewUser(PublicKey string) Tribes.User {
	user := Tribes.User{
		PubKey: PublicKey,
		Reputation: 100.0,
		Tokens: 69.0,
	}
	return user
}
func UserAddr(user Tribes.User) string {
	ans := hashtools.Hexdigest("user")[:6] + hashtools.Hexdigest(
	    ↪ user.PubKey)[:64]
	return ans
}
func KeyAddr(key string) string {
	ans := hashtools.Hexdigest("user")[:6] + hashtools.Hexdigest(key
	    ↪ )[:64]
```

```
        return ans
}
func WrapUser(user Tribes.User, command string) Tribes.Wrapper {
        data, _ := proto.Marshal(&user)
        wrap := Tribes.Wrapper{
                Tribe: "User",
                Command: command,
                Data: data,
        }
        return wrap
}

func Punish(user *Tribes.User) {
        (*user).Reputation -= 1.0
}
func Reward(user *Tribes.User) {
        (*user).Reputation += 1.0
}
func UserTransaction(user Tribes.User, command string, signer *signing.
   ↪ Signer) transaction_pb2.Transaction {
        wrap := WrapUser(user, command)
        payloadBytes, _ := proto.Marshal(&wrap)
        rawTransactionHeader := transaction_pb2.TransactionHeader{
                SignerPublicKey: signer.GetPublicKey().AsHex(),
                FamilyName: "cbed",
                FamilyVersion: "1.0",
                Dependencies: []string{},
                BatcherPublicKey: signer.GetPublicKey().AsHex(),
                Inputs: []string{UserAddr(user)},
                Outputs: []string{UserAddr(user)},
                PayloadSha512: hashtools.HashData(payloadBytes),
```

```
        }
        transactionHeaderBytes, _ := proto.Marshal(&rawTransactionHeader
            ↪ )
        signature := hex.EncodeToString(signer.Sign(
            ↪ transactionHeaderBytes))
        transaction := transaction_pb2.Transaction{
                Header: transactionHeaderBytes,
                HeaderSignature: signature,
                Payload: payloadBytes,
        }
        return transaction
}

func RegisterUser(user Tribes.User, signer *signing.Signer, validatorIP
    ↪  string) {
        userTransaction := UserTransaction(user, "Register", signer)
        blockchain.SendTransaction(userTransaction, signer, validatorIP)
}

func GetUsers(validatorIP string) map[string]Tribes.User {
        states := blockchain.GetStates(hashtools.Hexdigest("user")[:6],
            ↪ validatorIP)
        users := map[string]Tribes.User{}
        for addr, userBytes := range states {
                var temp Tribes.User
                proto.Unmarshal(userBytes, &temp)
                users[addr] = temp
        }
        return users
}
```

A.7 Validation API

```
package Validation

import (
	"Tribes"
	"crowdsourcing/Submission"
	"encoding/hex"
	"hashtools"
	"reputation/User"

	"github.com/golang/protobuf/proto"
	"github.com/hyperledger/sawtooth-sdk-go/protobuf/transaction_pb2
	   ↪ "
	"github.com/hyperledger/sawtooth-sdk-go/signing"
)

func ValidationWrap(validation Tribes.Validation) Tribes.Wrapper {
	data, _ := proto.Marshal(&validation)
	wrap := Tribes.Wrapper{
		Tribe: "Validation",
		Data: data,
	}
	return wrap
}
func SplitProof(proofs map[string]Tribes.Proof) ([]string, []string,
   ↪ []string) {
	submission := []string{}
	proof := []string{}
	user := []string{}
	for addr, data := range proofs {
		submission = append(submission, data.Submission)
```

```
            proof = append(proof, addr)
            user = append(user, data.User)
        }
        return submission, proof, user
}

func GetAnswer(submissions map[string]Tribes.Submission) (int, []byte)
    ↪ {
        counter := map[string]int{}
        var maxVal int
        var maxAns []byte
        for _, submission := range submissions {
            counter[hashtools.HashData(submission.Data)]++
            if maxVal < counter[hashtools.HashData(submission.Data)]
                ↪ {
                maxVal = counter[hashtools.HashData(submission.
                    ↪ Data)]
                maxAns = submission.Data
            }
        }
        return maxVal, maxAns
}

func ReputationUpdate(users map[string]Tribes.User, submissions map[
    ↪ string]Tribes.Submission, answer []byte, reward float64) map[
    ↪ string]Tribes.User {
        tempUsers := users
        hanswer := hashtools.HashData(answer)
        for _, submission := range submissions {
            temp := tempUsers[submission.User]
            if hashtools.HashData(submission.Data) == hanswer {
```

```
                User.Reward(&temp)
                temp.Tokens += reward
            } else {
                User.Punish(&temp)
            }
            tempUsers[submission.User] = temp
        }
        return tempUsers
}

func Transaction(validation Tribes.Validation, signer *signing.Signer,
    ↪ validatorIP string) transaction_pb2.Transaction {
        submittedProofs := Submission.GetProofList(validation.Task,
            ↪ validatorIP)
        submission, proof, user := SplitProof(submittedProofs)
        var access []string
        access = append(access, submission...) //r,w on submissions
        access = append(access, proof...) //r,w on proofs
        access = append(access, user...) //r,w on users
        access = append(access, validation.Task) //r,w on task
        wrap := ValidationWrap(validation)
        payloadBytes, _ := proto.Marshal(&wrap)
        rawTransactionHeader := transaction_pb2.TransactionHeader{
            SignerPublicKey: signer.GetPublicKey().AsHex(),
            FamilyName: "cbed",
            FamilyVersion: "1.0",
            Dependencies: []string{},
            BatcherPublicKey: signer.GetPublicKey().AsHex(),
            Inputs: access,
            Outputs: access,
            PayloadSha512: hashtools.HashData(payloadBytes),
```

```
        }
        transactionHeaderBytes, _ := proto.Marshal(&rawTransactionHeader
            ↪ )
        signature := hex.EncodeToString(signer.Sign(
            ↪ transactionHeaderBytes))
        transaction := transaction_pb2.Transaction{
                Header: transactionHeaderBytes,
                HeaderSignature: signature,
                Payload: payloadBytes,
        }
        return transaction
}
```

A.8 Hashtools

```
package hashtools

import (
        "crypto/sha512"
        "encoding/hex"
        "strings"
)

func Hexdigest(str string) string {
        hash := sha512.New()
        hash.Write([]byte(str))
        hashBytes := hash.Sum(nil)
        return strings.ToLower(hex.EncodeToString(hashBytes))
}

func HashData(payloadBytes []byte) string {
        hashHandler := sha512.New()
```

```
        hashHandler.Write(payloadBytes)
        payloadSha512 := strings.ToLower(hex.EncodeToString(hashHandler.
            ↪ Sum(nil)))
        return payloadSha512
}
```

A.9 Unlock

```
package unlock

import (
	"crypto/ecdsa"
	"crypto/elliptic"
	"crypto/rand"
	"crypto/x509"
	"encoding/pem"
	"fmt"
	"io/ioutil"
	"os"

	"github.com/hyperledger/sawtooth-sdk-go/signing"
)

func GetSybilUser() *signing.Signer {
	context := signing.NewSecp256k1Context()
	privateKey := context.NewRandomPrivateKey()
	signer := signing.NewCryptoFactory(context).NewSigner(privateKey
	    ↪ )
	return signer
}

func GetNewUser(path string) *signing.Signer {
	priv, _ := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
	keyOut, _ := os.OpenFile(path, os.O_WRONLY|os.O_CREATE|os.
	    ↪ O_TRUNC, 0600)
	privBytes, _ := x509.MarshalPKCS8PrivateKey(priv)
	pem.Encode(keyOut, &pem.Block{Type: "PRIVATE KEY", Bytes:
	    ↪ privBytes})
```

```
        keyOut.Close()

        return GetUser(path)
}

func GetUser(path string) *signing.Signer {
        buf, _ := ioutil.ReadFile(path)
        key := string(buf)
        context := signing.NewSecp256k1Context()
        privateKey, err := signing.PemToSecp256k1PrivateKey(key, "")
        if err != nil {
                fmt.Print("Epie tou kolou i prospathia")
        }
        signer := signing.NewCryptoFactory(context).NewSigner(privateKey
            ↪ )
        return signer
}
```

A.10 Transaction Family

```
package handler

import (
	"Tribes"
	"crowdsourcing/Submission"
	"crowdsourcing/Task"
	"encoding/hex"
	"fmt"
	"hashtools"
	"reputation/User"
	"reputation/Validation"
	"time"

	"github.com/golang/protobuf/proto"

	"protobuf/processor_pb2"

	"github.com/hyperledger/sawtooth-sdk-go/logging"
	"github.com/hyperledger/sawtooth-sdk-go/processor"
	"github.com/hyperledger/sawtooth-sdk-go/signing"
)

type CbedHandler struct {
	Restapi string
}

func (self *CbedHandler) FamilyName() string {
	return "cbed"
}
```

```
func (self *CbedHandler) FamilyVersions() []string {
	return []string{"1.0"}
}

func (self *CbedHandler) Namespaces() []string {
	return []string{hashtools.Hexdigest("task")[:6], hashtools.
	    ↪ Hexdigest("submission")[:6], hashtools.Hexdigest("lock")
	    ↪ [:6], hashtools.Hexdigest("proof")[:6], hashtools.
	    ↪ Hexdigest("user")[:6]}
}

var logger *logging.Logger = logging.Get()

func logIn(useraddr string, pubkey string, context *processor.Context)
    ↪ (Tribes.User, error) {
	if useraddr != User.KeyAddr(pubkey) {
		return Tribes.User{}, &processor.InvalidTransactionError{
			Msg: fmt.Sprint("Invalid UserID"),
		}
	}
	state, err := context.GetState([]string{useraddr})
	if err != nil {
		return Tribes.User{}, &processor.InvalidTransactionError{
			Msg: fmt.Sprint("Failed to Retrieve User: ", err),
		}
	}
	if _, ok := state[useraddr]; !ok {
		return Tribes.User{}, &processor.InvalidTransactionError{
			Msg: fmt.Sprint("User doesn't exist!!!"),
		}
	}
```

```
	var user Tribes.User
	err = proto.Unmarshal(state[useraddr], &user)
	if err != nil {
		return Tribes.User{}, &processor.InvalidTransactionError{
			Msg: fmt.Sprint("Failed to Decode User: ", err),
		}
	}
	return user, nil
}
func (self *CbedHandler) Apply(request *processor_pb2.TpProcessRequest,
   ↪ context *processor.Context) error {
	/*defer func() {
		if err := recover(); err != nil {
			logger.Info("Crashed but stays strong")
		}
	}()*/
	var wrap Tribes.Wrapper
	err := proto.Unmarshal(request.Payload, &wrap)
	if err != nil {
		return &processor.InvalidTransactionError{
			Msg: fmt.Sprint("Failed to decode payload: ", err)
			   ↪ ,
		}
	}
	switch wrap.Tribe {
	case "Task":
		var task Tribes.Task
		var user Tribes.User
		var state map[string][]byte
		err := proto.Unmarshal(wrap.Data, &task)
		if err != nil {
```

```
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("Failed to decode Task: ",
                    ↪ err),
        }
}
user, err = logIn(task.User, request.Header.
    ↪ SignerPublicKey, context)
if err != nil {
        return err
}
state, err = context.GetState([]string{Task.TaskAddr(task
    ↪ )})
if _, ok := state[Task.TaskAddr(task)]; ok {
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("Task already exists", task
                    ↪ .Name),
        }
}


if user.Tokens < task.Tokens {
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("User doesn't have enough
                    ↪ founds!!!"),
        }
}
user.Tokens -= task.Tokens
UserBytes, _ := proto.Marshal(&user)
_, err = context.SetState(map[string][]byte{task.User:
    ↪ UserBytes})
if err != nil {
        return &processor.InvalidTransactionError{
```

```
                        Msg: fmt.Sprint("Failed to Update User
                            ↪ Funds: ", err),
                }
        }
        _, err = context.SetState(map[string][]byte{Task.TaskAddr
            ↪ (task): wrap.Data})
        if err != nil {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Failed to Add Task: ", err
                            ↪ ),
                }
        }

        newTask := processor.Attribute{
                Key: "ID",
                Value: Task.TaskAddr(task),
        }
        attributes := []processor.Attribute{newTask}
        var empty []byte
        context.AddEvent("cbed/NewTask", attributes, empty)

case "Submission":
        var submission Tribes.Submission
        var state map[string][]byte
        var proof Tribes.Proof

        //TODO: Check if it is before the Deadline
        err := proto.Unmarshal(wrap.Data, &submission)
        if err != nil {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Failed to decode
```

```
                    ↪ Submission: ", err),
        }
}
_, err = logIn(submission.User, request.Header.
    ↪ SignerPublicKey, context)
if err != nil {
        return err
}

state, err = context.GetState([]string{submission.Proof})
if err != nil {
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("Failed to Retrieve Proof:
                    ↪ ", err),
        }
}
if _, ok := state[submission.Proof]; !ok {
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("Proof doesn't exist!!!"),
        }
}
err = proto.Unmarshal(state[submission.Proof], &proof)
if err != nil {
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("Failed to decode Proof: ",
                    ↪  err),
        }
}

//Proof of Submission
bytekey, _ := hex.DecodeString(request.GetHeader().
```

```
            ↪ SignerPublicKey)
        PubKey := signing.NewSecp256k1PublicKey(bytekey)
        keycontext := signing.NewSecp256k1Context()
        signature, _ := hex.DecodeString(submission.Signature)

        if !keycontext.Verify([]byte(signature), []byte(
            ↪ submission.Data), PubKey) || proof.Hignature !=
            ↪ hashtools.Hexdigest(submission.Signature) {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Submission Doesn't match
                            ↪ with proof!!!"),
                }
        }

        state, err = context.GetState([]string{Submission.
            ↪ SubmissionAddr(submission)})
        if _, ok := state[Submission.SubmissionAddr(submission)];
            ↪  ok {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Already Submitted!!!"),
                }
        }
        _, err = context.SetState(map[string][]byte{Submission.
            ↪ SubmissionAddr(submission): wrap.Data})
        if err != nil {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Failed to Add Submission:
                            ↪ ", err),
                }
        }
case "Lock":
```

```
//TODO: Check if after Deadline (UTC Time)
var lock Tribes.Lock
var task Tribes.Task

var states map[string][]byte
err := proto.Unmarshal(wrap.Data, &lock)
if err != nil {
	return &processor.InvalidTransactionError{
		Msg: fmt.Sprint("Failed to decode lock: ",
		    ↪ lock),
	}
}
states, err = context.GetState([]string{lock.Task})
if err != nil {
	return &processor.InvalidTransactionError{
		Msg: fmt.Sprint("Failed to Retrieve task: "
		    ↪ , err),
	}
}
if _, ok := states[lock.Task]; !ok {
	return &processor.InvalidTransactionError{
		Msg: fmt.Sprint("Task doesn't exists: ",
		    ↪ lock.Task),
	}
}
err = proto.Unmarshal(states[lock.Task], &task)
if err != nil {
	return &processor.InvalidTransactionError{
		Msg: fmt.Sprint("Failed to decode task: ",
		    ↪ err),
	}
```

```
            }
            if time.Now().Before(time.Unix(task.Deadline, 0)) || task
               ↪ .Locked {
                return &processor.InternalError{
                        Msg: fmt.Sprint("Too soon for the lock"),
                }
            }
            task.Locked = true
            state, _ := proto.Marshal(&task)
            _, err = context.SetState(map[string][]byte{lock.Task:
               ↪ state})
            if err != nil {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Failed to update task: ",
                           ↪ err),
                }
            }
            if User.KeyAddr(request.Header.SignerPublicKey) != task.
               ↪ User {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Invalid user adds lock"),
                }
            }

        case "Proof":
            //TODO: Check if before Deadline (UTC Time)
            var proof Tribes.Proof
            var states map[string][]byte
            err := proto.Unmarshal(wrap.Data, &proof)
            if err != nil {
                return &processor.InvalidTransactionError{
```

```
                Msg: fmt.Sprint("Failed to decode Proof: ",
                    ↪  err),
        }
}
_, err = logIn(proof.User, request.Header.SignerPublicKey
    ↪ , context)
if err != nil {
        return err
}
states, err = context.GetState([]string{proof.Task})
if err != nil {
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("Failed to Retrieve task: "
                    ↪ , err),
        }
}
if _, ok := states[proof.Task]; !ok {
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("Task doesn't exists: ",
                    ↪ proof.Task),
        }
}
var task Tribes.Task
proto.Unmarshal(states[proof.Task], &task)
if !time.Now().Before(time.Unix(task.Deadline, 0)) {
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("Too late for the proof"),
        }
}
_, err = context.SetState(map[string][]byte{Submission.
    ↪ ProofAddr(proof): wrap.Data})
```

```
            if err != nil {
                  return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Failed to Add Proof: ",
                           ↪ err),
                  }
            }

      case "User":
            switch wrap.Command {
            case "Register":
                  var user Tribes.User
                  err := proto.Unmarshal(wrap.Data, &user)
                  if err != nil {
                        return &processor.InvalidTransactionError{
                              Msg: fmt.Sprint("Failed to decode
                                 ↪ User: ", err),
                        }
                  }
                  users, _ := context.GetState([]string{User.KeyAddr
                     ↪ (request.Header.SignerPublicKey)})
                  if len(users) != 0 {
                        return &processor.InvalidTransactionError{
                              Msg: fmt.Sprint("User already exists
                                 ↪ : ", err),
                        }
                  }
                  _, err = context.SetState(map[string][]byte{User.
                     ↪ KeyAddr(request.Header.SignerPublicKey):
                     ↪ wrap.Data})
                  if err != nil {
                        return &processor.InvalidTransactionError{
```

```
                                Msg: fmt.Sprint("Failed to Register
                                    ↪ User: ", err),
                        }
                }
        }
case "Validation":
        var validation Tribes.Validation
        err := proto.Unmarshal(wrap.Data, &validation)
        if err != nil {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Failed to decode
                            ↪ Validation: ", err),
                }
        }
        states, err := context.GetState([]string{validation.Task
            ↪ })
        if err != nil {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Failed to get Task: ", err
                            ↪ ),
                }
        }
        if len(states) == 0 {
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Task doesn't exist!!!"),
                }
        }
        var task Tribes.Task
        proto.Unmarshal(states[validation.Task], &task)
        if task.Validated {
                //return nil
```

```
        return &processor.InvalidTransactionError{
                Msg: fmt.Sprint("Task already Validated!!!"
                    ↪ ),
        }
}
if time.Now().Before(time.Unix(task.Deadline, 0).Add(10 *
    ↪  time.Second)) {
        return &processor.InternalError{
                Msg: fmt.Sprint("Too soon for the
                    ↪ Validation"),
        }
}
proofs := Submission.GetProofList(validation.Task, self.
    ↪ Restapi)
submissionAddr, _, userAddr := Validation.SplitProof(
    ↪ proofs)
states, _ = context.GetState(submissionAddr)

submissions := map[string]Tribes.Submission{}
for addr, submissionBytes := range states {
        var temp Tribes.Submission
        proto.Unmarshal(submissionBytes, &temp)
        submissions[addr] = temp
}

states, _ = context.GetState(userAddr)
users := map[string]Tribes.User{}
for addr, UserBytes := range states {
        var temp Tribes.User
        proto.Unmarshal(UserBytes, &temp)
        users[addr] = temp
```

```
}

howmany, answer := Validation.GetAnswer(submissions)
if howmany == 0 {
        requesterbytes, _ := context.GetState([]string{
            ↪ task.User})
        var requester Tribes.User
        proto.Unmarshal(requesterbytes[task.User], &
            ↪ requester)
        requester.Tokens += task.Tokens
        task.Tokens = 0
        requesterbytes[task.User], _ = proto.Marshal(&
            ↪ requester)
        context.SetState(requesterbytes)
} else {
        reward := task.Tokens / float64(howmany)
        users = Validation.ReputationUpdate(users,
            ↪ submissions, answer, reward)

        for addr, user := range users {
                states[addr], _ = proto.Marshal(&user)
        }

        context.SetState(states)
        task.Validated = true
        task.Answer = answer
        task.Tokens = 0
        taskBytes, _ := proto.Marshal(&task)
        context.SetState(map[string][]byte{Task.TaskAddr(
            ↪ task): taskBytes})
}
```

```
        default:
                return &processor.InvalidTransactionError{
                        Msg: fmt.Sprint("Non existent Tribe"),
                }
        }
        return nil
}

package main

import (
        cbed "handler"
        "os"
        "syscall"

        "github.com/hyperledger/sawtooth-sdk-go/processor"
)

func main() {
        //"tcp://validator:4004"
        endpoint := os.Args[1]
        handler := cbed.CbedHandler{
                Restapi: os.Args[2],
        }
        processor := processor.NewTransactionProcessor(endpoint)
        processor.AddHandler(&handler)
        processor.ShutdownOnSignal(syscall.SIGINT, syscall.SIGTERM)
        processor.Start()
}
```

A.11 Data Types

```
syntax="proto3";


package Tribes;
option go_package = "CrowdBED_API/src/Tribes";


message Task{
    string Name=1;
    string User=2;
    int64 Deadline=3;
    double Tokens=4;
    bool Validated=5;
    bytes Answer=6;
    bytes Code=7;
    bytes Data=8;
    bytes Description=9;
    string Lock=10;
    bool Locked=11;
}
message Submission{
    string Task=1;
    bytes Data=2;
    string User=3;
    string Lock=4;
    string Proof=5;
    string Signature=6;
}
message Wrapper{
    string Tribe=2;
    string Command=3;
    bytes Data=4;
```

```
}
message Lock{
    string Task=1;
}
message User{
    string PubKey=1;
    double Reputation=2;
    double Tokens=3;
}
message Validation{
    string Task=1;
}
message Proof{
    string Submission=1;
    string Task=2;
    string User=3;
    string Hignature=4;
}
```

A.12 Batcher gRPC

```
syntax="proto3";

package batcherrpc;
option go_package = ".;batcherrpc";
message Empty{

}
message Key{
    string hex=1;
}
message Trans{
```

```
    bytes serial=1;
}
service Holder{
    rpc GetKey(Empty)returns(Key){}
    rpc SendTransaction(Trans)returns(Empty){}
}
```

A.13 Single Node (Dev Mode)

```
# Copyright 2017 Intel Corporation
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   ↪ implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
   ↪ ------------------------------------------------------------------------------
   ↪

version: "3.3"

services:

  settings-tp:
    image: hyperledger/sawtooth-settings-tp:latest
    container_name: sawtooth-settings-tp-default
    depends_on:
      - validator
    entrypoint: settings-tp -vv -C tcp://validator:4004

  cbed-tp-go:
```

```
  image: sawtooth-cbed-tp-go:latest
  container_name: cbed-tp-go-default
  depends_on:
    - validator
  entrypoint: "./bin/cbed tcp://validator:4004 http://rest-api:8000"

batcher-go:
  image: sawtooth-batcher-go:latest
  container_name: sawtooth-batcher-go
  ports:
    - "7000:7000"
  depends_on:
    - rest-api
  entrypoint: "./bin/batcher http://rest-api:8000"

validator:
  image: hyperledger/sawtooth-validator:latest
  container_name: sawtooth-validator-default
  expose:
    - 4004
  ports:
    - "4004:4004"
  # start the validator with an empty genesis batch
  entrypoint: "bash -c \"\
      sawadm keygen && \
      sawtooth keygen my_key && \
      sawset genesis -k /root/.sawtooth/keys/my_key.priv && \
      sawset proposal create \
        -k /root/.sawtooth/keys/my_key.priv \
        sawtooth.consensus.algorithm.name=Devmode \
        sawtooth.consensus.algorithm.version=0.1 \
```

```
        -o config.batch && \
      sawadm genesis config-genesis.batch config.batch && \
      sawtooth-validator -vv \
        --endpoint tcp://validator:8800 \
        --bind component:tcp://eth0:4004 \
        --bind network:tcp://eth0:8800 \
        --bind consensus:tcp://eth0:5050 \
      \""

  devmode-engine:
    image: hyperledger/sawtooth-devmode-engine-rust:latest
    container_name: sawtooth-devmode-engine-rust-default
    depends_on:
      - validator
    entrypoint: devmode-engine-rust -C tcp://validator:5050

  rest-api:
    image: hyperledger/sawtooth-rest-api:latest
    container_name: sawtooth-rest-api-default
    ports:
      - "8000:8000"
    depends_on:
      - validator
    entrypoint: sawtooth-rest-api -C tcp://validator:4004 --bind rest-
      ↪ api:8000

  shell:
    image: hyperledger/sawtooth-shell:latest
    container_name: sawtooth-shell-default
    depends_on:
      - rest-api
```

```
entrypoint: "bash -c \"\
    sawtooth keygen && \
    tail -f /dev/null \
    \""
```

A.14 PBFT Network

```
# Copyright 2019 Cargill Incorporated
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   ↪ implied.
# See the License for the specific language governing permissions and
# limitations under the License.

version: '3.3'

volumes:
  pbft-shared:

services:

# -------------=== rest api ===-------------

  rest-api-0:
    image: hyperledger/sawtooth-rest-api:latest
    container_name: sawtooth-rest-api-default-0
    expose:
      - 8008
    ports:
```

```
    - "8000:8008"
  command: |
    bash -c "
      sawtooth-rest-api \
        --connect tcp://validator-0:4004 \
        --bind rest-api-0:8008
    "
  stop_signal: SIGKILL

rest-api-1:
  image: hyperledger/sawtooth-rest-api:latest
  container_name: sawtooth-rest-api-default-1
  expose:
    - 8008
  ports:
    - "8001:8008"
  command: |
    bash -c "
      sawtooth-rest-api \
        --connect tcp://validator-1:4004 \
        --bind rest-api-1:8008
    "
  stop_signal: SIGKILL

rest-api-2:
  image: hyperledger/sawtooth-rest-api:latest
  container_name: sawtooth-rest-api-default-2
  expose:
    - 8008
  ports:
    - "8002:8008"
```

```
  command: |
    bash -c "
      sawtooth-rest-api \
        --connect tcp://validator-2:4004 \
        --bind rest-api-2:8008
    "
  stop_signal: SIGKILL

rest-api-3:
  image: hyperledger/sawtooth-rest-api:latest
  container_name: sawtooth-rest-api-default-3
  expose:
    - 8008
  ports:
    - "8003:8008"
  command: |
    bash -c "
      sawtooth-rest-api \
        --connect tcp://validator-3:4004 \
        --bind rest-api-3:8008
    "
  stop_signal: SIGKILL

rest-api-4:
  image: hyperledger/sawtooth-rest-api:latest
  container_name: sawtooth-rest-api-default-4
  expose:
    - 8008
  ports:
    - "8004:8008"
  command: |
```

```
      bash -c "
        sawtooth-rest-api \
          --connect tcp://validator-4:4004 \
          --bind rest-api-4:8008
      "
    stop_signal: SIGKILL

# -------------=== settings tp ===--------------

  settings-tp-0:
    image: hyperledger/sawtooth-settings-tp:latest
    container_name: sawtooth-settings-tp-default-0
    expose:
      - 4004
    command: settings-tp -C tcp://validator-0:4004
    stop_signal: SIGKILL

  settings-tp-1:
    image: hyperledger/sawtooth-settings-tp:latest
    container_name: sawtooth-settings-tp-default-1
    expose:
      - 4004
    command: settings-tp -C tcp://validator-1:4004
    stop_signal: SIGKILL

  settings-tp-2:
    image: hyperledger/sawtooth-settings-tp:latest
    container_name: sawtooth-settings-tp-default-2
    expose:
      - 4004
    command: settings-tp -C tcp://validator-2:4004
```

```
    stop_signal: SIGKILL

  settings-tp-3:
    image: hyperledger/sawtooth-settings-tp:latest
    container_name: sawtooth-settings-tp-default-3
    expose:
      - 4004
    command: settings-tp -C tcp://validator-3:4004
    stop_signal: SIGKILL

  settings-tp-4:
    image: hyperledger/sawtooth-settings-tp:latest
    container_name: sawtooth-settings-tp-default-4
    expose:
      - 4004
    command: settings-tp -C tcp://validator-4:4004
    stop_signal: SIGKILL

# --------------=== shell ===---------------

  shell:
    image: hyperledger/sawtooth-shell:latest
    container_name: sawtooth-shell-default
    volumes:
      - pbft-shared:/pbft-shared
    command: |
      bash -c "
        sawtooth keygen
        tail -f /dev/null
      "
    stop_signal: SIGKILL
```

```
# --------------=== validators ===---------------

  validator-0:
    image: hyperledger/sawtooth-validator:latest
    container_name: sawtooth-validator-default-0
    expose:
      - 4004
      - 5050
      - 8800
    volumes:
      - pbft-shared:/pbft-shared
    command: |
      bash -c "
        if [ -e /pbft-shared/validators/validator-0.priv ]; then
          cp /pbft-shared/validators/validator-0.pub /etc/sawtooth/keys/
             ↪ validator.pub
          cp /pbft-shared/validators/validator-0.priv /etc/sawtooth/keys
             ↪ /validator.priv
        fi &&
        if [ ! -e /etc/sawtooth/keys/validator.priv ]; then
          sawadm keygen
          mkdir -p /pbft-shared/validators || true
          cp /etc/sawtooth/keys/validator.pub /pbft-shared/validators/
             ↪ validator-0.pub
          cp /etc/sawtooth/keys/validator.priv /pbft-shared/validators/
             ↪ validator-0.priv
        fi &&
        if [ ! -e config-genesis.batch ]; then
          sawset genesis -k /etc/sawtooth/keys/validator.priv -o config-
             ↪ genesis.batch
```

```
fi &&
while [[ ! -f /pbft-shared/validators/validator-1.pub || \
        ! -f /pbft-shared/validators/validator-2.pub || \
        ! -f /pbft-shared/validators/validator-3.pub || \
        ! -f /pbft-shared/validators/validator-4.pub ]];
do sleep 1; done
echo sawtooth.consensus.pbft.members=\\['\"'$$(cat /pbft-shared/
   ↪ validators/validator-0.pub)'\"','\"'$$(cat /pbft-shared/
   ↪ validators/validator-1.pub)'\"','\"'$$(cat /pbft-shared/
   ↪ validators/validator-2.pub)'\"','\"'$$(cat /pbft-shared/
   ↪ validators/validator-3.pub)'\"','\"'$$(cat /pbft-shared/
   ↪ validators/validator-4.pub)'\"'\\] &&
if [ ! -e config.batch ]; then
 sawset proposal create \
   -k /etc/sawtooth/keys/validator.priv \
   sawtooth.consensus.algorithm.name=pbft \
   sawtooth.consensus.algorithm.version=1.0 \
   sawtooth.consensus.pbft.members=\\['\"'$$(cat /pbft-shared/
      ↪ validators/validator-0.pub)'\"','\"'$$(cat /pbft-
      ↪ shared/validators/validator-1.pub)'\"','\"'$$(cat /
      ↪ pbft-shared/validators/validator-2.pub)'\"','\"'$$(cat
      ↪  /pbft-shared/validators/validator-3.pub)'\"','\"'$$(
      ↪ cat /pbft-shared/validators/validator-4.pub)'\"'\\] \
   sawtooth.publisher.max_batches_per_block=1200 \
   -o config.batch
fi &&
if [ ! -e /var/lib/sawtooth/genesis.batch ]; then
  sawadm genesis config-genesis.batch config.batch
fi &&
if [ ! -e /root/.sawtooth/keys/my_key.priv ]; then
  sawtooth keygen my_key
```

```
      fi &&
      sawtooth-validator -vv \
        --endpoint tcp://validator-0:8800 \
        --bind component:tcp://eth0:4004 \
        --bind consensus:tcp://eth0:5050 \
        --bind network:tcp://eth0:8800 \
        --scheduler parallel \
        --peering static \
        --maximum-peer-connectivity 10000
    "

validator-1:
  image: hyperledger/sawtooth-validator:latest
  container_name: sawtooth-validator-default-1
  expose:
    - 4004
    - 5050
    - 8800
  volumes:
    - pbft-shared:/pbft-shared
  command: |
    bash -c "
      if [ -e /pbft-shared/validators/validator-1.priv ]; then
        cp /pbft-shared/validators/validator-1.pub /etc/sawtooth/keys/
           ↪ validator.pub
        cp /pbft-shared/validators/validator-1.priv /etc/sawtooth/keys
           ↪ /validator.priv
      fi &&
      if [ ! -e /etc/sawtooth/keys/validator.priv ]; then
        sawadm keygen
        mkdir -p /pbft-shared/validators || true
```

```
        cp /etc/sawtooth/keys/validator.pub /pbft-shared/validators/
           ↪ validator-1.pub
        cp /etc/sawtooth/keys/validator.priv /pbft-shared/validators/
           ↪ validator-1.priv
      fi &&
      sawtooth keygen my_key &&
      sawtooth-validator -vv \
        --endpoint tcp://validator-1:8800 \
        --bind component:tcp://eth0:4004 \
        --bind consensus:tcp://eth0:5050 \
        --bind network:tcp://eth0:8800 \
        --scheduler parallel \
        --peering static \
        --maximum-peer-connectivity 10000 \
        --peers tcp://validator-0:8800
    "

validator-2:
  image: hyperledger/sawtooth-validator:latest
  container_name: sawtooth-validator-default-2
  expose:
    - 4004
    - 5050
    - 8800
  volumes:
    - pbft-shared:/pbft-shared
  command: |
    bash -c "
      if [ -e /pbft-shared/validators/validator-2.priv ]; then
        cp /pbft-shared/validators/validator-2.pub /etc/sawtooth/keys/
           ↪ validator.pub
```

```
        cp /pbft-shared/validators/validator-2.priv /etc/sawtooth/keys
           ↪ /validator.priv
      fi &&
      if [ ! -e /etc/sawtooth/keys/validator.priv ]; then
        sawadm keygen
        mkdir -p /pbft-shared/validators || true
        cp /etc/sawtooth/keys/validator.pub /pbft-shared/validators/
           ↪ validator-2.pub
        cp /etc/sawtooth/keys/validator.priv /pbft-shared/validators/
           ↪ validator-2.priv
      fi &&
      sawtooth keygen my_key &&
      sawtooth-validator -vv \
        --endpoint tcp://validator-2:8800 \
        --bind component:tcp://eth0:4004 \
        --bind consensus:tcp://eth0:5050 \
        --bind network:tcp://eth0:8800 \
        --scheduler parallel \
        --peering static \
        --maximum-peer-connectivity 10000 \
        --peers tcp://validator-0:8800 \
        --peers tcp://validator-1:8800
    "

  validator-3:
    image: hyperledger/sawtooth-validator:latest
    container_name: sawtooth-validator-default-3
    expose:
      - 4004
      - 5050
      - 8800
```

```
volumes:
  - pbft-shared:/pbft-shared
command: |
  bash -c "
    if [ -e /pbft-shared/validators/validator-3.priv ]; then
     cp /pbft-shared/validators/validator-3.pub /etc/sawtooth/keys/
        ↪ validator.pub
     cp /pbft-shared/validators/validator-3.priv /etc/sawtooth/keys/
        ↪ validator.priv
    fi &&
    if [ ! -e /etc/sawtooth/keys/validator.priv ]; then
     sawadm keygen
     mkdir -p /pbft-shared/validators || true
     cp /etc/sawtooth/keys/validator.pub /pbft-shared/validators/
        ↪ validator-3.pub
     cp /etc/sawtooth/keys/validator.priv /pbft-shared/validators/
        ↪ validator-3.priv
    fi &&
    sawtooth keygen my_key &&
    sawtooth-validator -vv \
      --endpoint tcp://validator-3:8800 \
      --bind component:tcp://eth0:4004 \
      --bind consensus:tcp://eth0:5050 \
      --bind network:tcp://eth0:8800 \
      --scheduler parallel \
      --peering static \
      --maximum-peer-connectivity 10000 \
      --peers tcp://validator-0:8800 \
      --peers tcp://validator-1:8800 \
      --peers tcp://validator-2:8800
  "
```

```
validator-4:
  image: hyperledger/sawtooth-validator:latest
  container_name: sawtooth-validator-default-4
  expose:
    - 4004
    - 5050
    - 8800
  volumes:
    - pbft-shared:/pbft-shared
  command: |
    bash -c "
      if [ -e /pbft-shared/validators/validator-4.priv ]; then
        cp /pbft-shared/validators/validator-4.pub /etc/sawtooth/keys/
           ↪ validator.pub
        cp /pbft-shared/validators/validator-4.priv /etc/sawtooth/keys
           ↪ /validator.priv
      fi &&
      if [ ! -e /etc/sawtooth/keys/validator.priv ]; then
        sawadm keygen
        mkdir -p /pbft-shared/validators || true
        cp /etc/sawtooth/keys/validator.pub /pbft-shared/validators/
           ↪ validator-4.pub
        cp /etc/sawtooth/keys/validator.priv /pbft-shared/validators/
           ↪ validator-4.priv
      fi &&
      sawtooth keygen my_key &&
      sawtooth-validator -vv \
        --endpoint tcp://validator-4:8800 \
        --bind component:tcp://eth0:4004 \
        --bind consensus:tcp://eth0:5050 \
```

```
        --bind network:tcp://eth0:8800 \
        --scheduler parallel \
        --peering static \
        --maximum-peer-connectivity 10000 \
        --peers tcp://validator-0:8800 \
        --peers tcp://validator-1:8800 \
        --peers tcp://validator-2:8800 \
        --peers tcp://validator-3:8800
    "

# --------------=== pbft engines ===--------------

  pbft-0:
    image: hyperledger/sawtooth-pbft-engine:latest
    container_name: sawtooth-pbft-engine-default-0
    command: pbft-engine -vv --connect tcp://validator-0:5050
    stop_signal: SIGKILL

  pbft-1:
    image: hyperledger/sawtooth-pbft-engine:latest
    container_name: sawtooth-pbft-engine-default-1
    command: pbft-engine -vv --connect tcp://validator-1:5050
    stop_signal: SIGKILL

  pbft-2:
    image: hyperledger/sawtooth-pbft-engine:latest
    container_name: sawtooth-pbft-engine-default-2
    command: pbft-engine -vv --connect tcp://validator-2:5050
    stop_signal: SIGKILL

  pbft-3:
```

```
    image: hyperledger/sawtooth-pbft-engine:latest
    container_name: sawtooth-pbft-engine-default-3
    command: pbft-engine -vv --connect tcp://validator-3:5050
    stop_signal: SIGKILL

  pbft-4:
    image: hyperledger/sawtooth-pbft-engine:latest
    container_name: sawtooth-pbft-engine-default-4
    command: pbft-engine -vv --connect tcp://validator-4:5050
    stop_signal: SIGKILL

# --------------=== cbed tps ===--------------

  cbed-tp-go-0:
    image: sawtooth-cbed-tp-go:latest
    container_name: cbed-tp-go-default-0
    expose:
      - 4004
    depends_on:
      - validator-0
    entrypoint: "./bin/cbed tcp://validator-0:4004 http://rest-api
       ↪ -0:8008"

  cbed-tp-go-1:
    image: sawtooth-cbed-tp-go:latest
    container_name: cbed-tp-go-default-1
    expose:
      - 4004
    depends_on:
      - validator-1
    entrypoint: "./bin/cbed tcp://validator-1:4004 http://rest-api
```

```
      ↪ -1:8008"

  cbed-tp-go-2:
    image: sawtooth-cbed-tp-go:latest
    container_name: cbed-tp-go-default-2
    expose:
      - 4004
    depends_on:
      - validator-2
    entrypoint: "./bin/cbed tcp://validator-2:4004 http://rest-api
      ↪ -2:8008"

  cbed-tp-go-3:
    image: sawtooth-cbed-tp-go:latest
    container_name: cbed-tp-go-default-3
    expose:
      - 4004
    depends_on:
      - validator-3
    entrypoint: "./bin/cbed tcp://validator-3:4004 http://rest-api
      ↪ -3:8008"

  cbed-tp-go-4:
    image: sawtooth-cbed-tp-go:latest
    container_name: cbed-tp-go-default-4
    expose:
      - 4004
    depends_on:
      - validator-4
    entrypoint: "./bin/cbed tcp://validator-4:4004 http://rest-api
      ↪ -4:8008"
```

```
# --------------=== Batchers ===-------------
  batcher-go-0:
    image: sawtooth-batcher-go:latest
    container_name: sawtooth-batcher-go-0
    expose:
      - 7000
    ports:
      - "7000:7000"
    depends_on:
      - rest-api-0
    entrypoint: "./bin/batcher http://rest-api-0:8008"

  batcher-go-1:
    image: sawtooth-batcher-go:latest
    container_name: sawtooth-batcher-go-1
    expose:
      - 7000
    ports:
      - "7001:7000"
    depends_on:
      - rest-api-1
    entrypoint: "./bin/batcher http://rest-api-1:8008"

  batcher-go-2:
    image: sawtooth-batcher-go:latest
    container_name: sawtooth-batcher-go-2
    expose:
      - 7000
    ports:
      - "7002:7000"
```

```
    depends_on:
      - rest-api-2
    entrypoint: "./bin/batcher http://rest-api-2:8008"

  batcher-go-3:
    image: sawtooth-batcher-go:latest
    container_name: sawtooth-batcher-go-3
    expose:
      - 7000
    ports:
      - "7003:7000"
    depends_on:
      - rest-api-3
    entrypoint: "./bin/batcher http://rest-api-3:8008"

  batcher-go-4:
    image: sawtooth-batcher-go:latest
    container_name: sawtooth-batcher-go-4
    expose:
      - 7000
    ports:
      - "7004:7000"
    depends_on:
      - rest-api-4
    entrypoint: "./bin/batcher http://rest-api-4:8008"
```

A.15 First Node

```
version: '3.3'

volumes:
  pbft-shared:

services:

# -------------=== rest api ===-------------

  rest-api:
    image: hyperledger/sawtooth-rest-api:latest
    container_name: sawtooth-rest-api-default
    expose:
      - 8008
    ports:
      - "8000:8008"
    command: |
      bash -c "
        sawtooth-rest-api \
          --connect tcp://validator:4004 \
          --bind rest-api:8008
      "
    stop_signal: SIGKILL

# --------------=== settings tp ===============

  settings-tp:
    image: hyperledger/sawtooth-settings-tp:latest
    container_name: sawtooth-settings-tp-default
    expose:
```

```
      - 4004
    command: settings-tp -C tcp://validator:4004
    stop_signal: SIGKILL

# -------------=== shell ===---------------

  shell:
    image: hyperledger/sawtooth-shell:latest
    container_name: sawtooth-shell-default
    volumes:
      - ./pbft-shared:/pbft-shared
    command: |
      bash -c "
        sawtooth keygen
        tail -f /dev/null
      "
    stop_signal: SIGKILL

# -------------=== validators ===---------------

  validator:
    image: hyperledger/sawtooth-validator:latest
    container_name: sawtooth-validator-default
    expose:
      - 4004
      - 5050
      - 8800
    volumes:
      - ./pbft-shared:/pbft-shared
    command: |
      bash -c "
```

```
if [ -e /pbft-shared/validators/validator-0.priv ]; then
  cp /pbft-shared/validators/validator-0.pub /etc/sawtooth/keys/
     ↪ validator.pub
  cp /pbft-shared/validators/validator-0.priv /etc/sawtooth/keys
     ↪ /validator.priv
fi &&
if [ ! -e config-genesis.batch ]; then
  sawset genesis -k /etc/sawtooth/keys/validator.priv -o config-
     ↪ genesis.batch
fi &&
echo sawtooth.consensus.pbft.members=\\['\"'$$(cat /pbft-shared/
   ↪ validators/validator-0.pub)'\"','\"'$$(cat /pbft-shared/
   ↪ validators/validator-1.pub)'\"','\"'$$(cat /pbft-shared/
   ↪ validators/validator-2.pub)'\"','\"'$$(cat /pbft-shared/
   ↪ validators/validator-3.pub)'\"','\"'$$(cat /pbft-shared/
   ↪ validators/validator-4.pub)'\"'\\] &&
if [ ! -e config.batch ]; then
 sawset proposal create \
    -k /etc/sawtooth/keys/validator.priv \
    sawtooth.consensus.algorithm.name=pbft \
    sawtooth.consensus.algorithm.version=1.0 \
    sawtooth.consensus.pbft.members=\\['\"'$$(cat /pbft-shared/
       ↪ validators/validator-0.pub)'\"','\"'$$(cat /pbft-
       ↪ shared/validators/validator-1.pub)'\"','\"'$$(cat /
       ↪ pbft-shared/validators/validator-2.pub)'\"','\"'$$(cat
       ↪  /pbft-shared/validators/validator-3.pub)'\"','\"'$$(
       ↪ cat /pbft-shared/validators/validator-4.pub)'\"'\\] \
    sawtooth.publisher.max_batches_per_block=1200 \
    -o config.batch
fi &&
if [ ! -e /var/lib/sawtooth/genesis.batch ]; then
```

```
        sawadm genesis config-genesis.batch config.batch
      fi &&
      if [ ! -e /root/.sawtooth/keys/my_key.priv ]; then
        sawtooth keygen my_key
      fi &&
      sawtooth-validator -vv \
        --endpoint tcp://validator:8800 \
        --bind component:tcp://eth0:4004 \
        --bind consensus:tcp://eth0:5050 \
        --bind network:tcp://eth0:8800 \
        --scheduler parallel \
        --peering static \
        --maximum-peer-connectivity 10000
    "
# -------------=== pbft engines ===-------------

  pbft:
    image: hyperledger/sawtooth-pbft-engine:latest
    container_name: sawtooth-pbft-engine-default
    command: pbft-engine -vv --connect tcp://validator:5050
    stop_signal: SIGKILL


# -------------=== cbed tps ===-------------

  cbed-tp-go:
    image: sawtooth-cbed-tp-go:latest
    container_name: cbed-tp-go-default
    expose:
      - 4004
    depends_on:
```

```
      - validator
    entrypoint: "./bin/cbed tcp://validator:4004 http://rest-api:8008"



# --------------=== Batchers ===--------------
  batcher-go:
    image: sawtooth-batcher-go:latest
    container_name: sawtooth-batcher-go
    expose:
      - 7000
    ports:
      - "7000:7000"
    depends_on:
      - rest-api
    entrypoint: "./bin/batcher http://rest-api:8008"
```

A.16 Other Nodes

```
version: '3.3'

volumes:
  pbft-shared:

services:

# -------------=== rest api ===-------------

  rest-api:
    image: hyperledger/sawtooth-rest-api:latest
    container_name: sawtooth-rest-api-default
    expose:
      - 8008
    ports:
      - "8000:8008"
    command: |
      bash -c "
        sawtooth-rest-api \
          --connect tcp://validator:4004 \
          --bind rest-api:8008
      "
    stop_signal: SIGKILL

# --------------=== settings tp ===============

  settings-tp:
    image: hyperledger/sawtooth-settings-tp:latest
    container_name: sawtooth-settings-tp-default
    expose:
```

```
      - 4004
    command: settings-tp -C tcp://validator:4004
    stop_signal: SIGKILL
# --------------=== shell ===---------------

  shell:
    image: hyperledger/sawtooth-shell:latest
    container_name: sawtooth-shell-default
    volumes:
      - ./pbft-shared:/pbft-shared
    command: |
      bash -c "
        sawtooth keygen
        tail -f /dev/null
      "
    stop_signal: SIGKILL

# --------------=== validators ===---------------

  validator:
    image: hyperledger/sawtooth-validator:latest
    container_name: sawtooth-validator-default
    expose:
      - 4004
      - 5050
      - 8800
    volumes:
      - ./pbft-shared:/pbft-shared
    ports:
      - "4004:4004"
      - "5050:5050"
```

```
      - "8800:8800"
    command: |
      bash -c "
        if [ -e /pbft-shared/validators/validator-.priv ]; then
          cp /pbft-shared/validators/validator-.pub /etc/sawtooth/keys/
             ↪ validator.pub
          cp /pbft-shared/validators/validator-.priv /etc/sawtooth/keys/
             ↪ validator.priv
        fi &&
        sawtooth keygen my_key &&
        sawtooth-validator -vv \
          --endpoint tcp://validator:8800 \
          --bind component:tcp://eth0:4004 \
          --bind consensus:tcp://eth0:5050 \
          --bind network:tcp://eth0:8800 \
          --scheduler parallel \
          --peering static \
          --maximum-peer-connectivity 10000 \
          --peers tcp://95.216.219.236:8800
      "
# --------------=== pbft engines ===--------------

  pbft:
    image: hyperledger/sawtooth-pbft-engine:latest
    container_name: sawtooth-pbft-engine-default
    command: pbft-engine -vv --connect tcp://validator:5050
    stop_signal: SIGKILL

# --------------=== cbed tps ===--------------

  cbed-tp-go:
```

```
    image: sawtooth-cbed-tp-go:latest
    container_name: cbed-tp-go-default
    expose:
      - 4004
    depends_on:
      - validator
    entrypoint: "./bin/cbed tcp://validator:4004 http://rest-api:8008"



# --------------=== Batchers ===-------------
  batcher-go:
    image: sawtooth-batcher-go:latest
    container_name: sawtooth-batcher-go
    expose:
      - 7000
    ports:
      - "7000:7000"
    depends_on:
      - rest-api
    entrypoint: "./bin/batcher http://rest-api:8008"
```

A.17 Makefile

```
BuildTF:
	-@rm -rf TransactionFamily/relative
	mkdir TransactionFamily/relative
	cp -r CrowdBED_API/src/* TransactionFamily/relative
	docker build -f TransactionFamily/TransactionFamily.dockerfile -
	   ↪ t sawtooth-cbed-tp-go TransactionFamily/
	rm -rf TransactionFamily/relative
BuildBatcher:
	-@rm -rf Batcher/relative
	mkdir Batcher/relative
	cp -r CrowdBED_API/src/* Batcher/relative
	docker build -f Batcher/Batcher.dockerfile -t sawtooth-batcher-
	   ↪ go Batcher/
	rm -rf Batcher/relative
RunClient:
	go run Client/src/main.go
OneNode:
	make BuildTF
	make BuildBatcher
	sleep 2
	docker-compose -f DockerNetworks/SingleNode.yaml up
PBFT-Net:
	make BuildTF
	make BuildBatcher
	sleep 2
	docker-compose -f DockerNetworks/PBFT_Network.yaml up
Nuke:
	-@docker stop `docker ps -aq`
	docker system prune -f
	docker volume prune -f
```

```
Tribes:
	protoc --go_out=. CrowdBED_API/src/Tribes/*.proto
BatcherRPC:
	protoc -I=./CrowdBED_API/src/batcherrpc --go_out=plugins=grpc:./
	   ↪ CrowdBED_API/src/batcherrpc ./CrowdBED_API/src/batcherrpc/
	   ↪ batcher.proto
InstallSawtooth:
	sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --
	   ↪ recv-keys 8AA7AF1F1091A5FD
	sudo add-apt-repository 'deb [arch=amd64] http://repo.sawtooth.
	   ↪ me/ubuntu/chime/stable bionic universe'
	sudo apt-get update
	sudo apt-get install -y sawtooth
	sudo apt-get install -y sawtooth sawtooth-pbft-engine
GenerateKeys:
	-@mkdir ./Node/pbft-shared
	-@mkdir ./Node/pbft-shared/validators
	-@sudo rm /etc/sawtooth/keys/validator*
	sudo sawadm keygen
	sudo chmod 644 /etc/sawtooth/keys/validator.priv
	sudo chmod 644 /etc/sawtooth/keys/validator.pub
	sudo cp /etc/sawtooth/keys/validator.pub ./Node/pbft-shared/
	   ↪ validators/validator-$(WHO).pub
	sudo cp /etc/sawtooth/keys/validator.priv ./Node/pbft-shared/
	   ↪ validators/validator-$(WHO).priv
GenesisNode:
	make GenerateKeys WHO="0"
	docker-compose -f Node/FirstNode.yaml up
FatherNode:
	docker-compose -f Node/OtherNodes.yaml up
```