Diogo Alexandre Valente Mendes

Bachelor in Computer Science and Engineering

Smart-contract Blockchain with Secure Hardware

Dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Nuno Preguiça, Associate Professor, NOVA University of Lisbon

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2020

Smart-contract Blockchain with Secure Hardware

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

First, I would like to express my sincere gratitude to my thesis adviser Nuno Preguiça for allowing me to work on a theme that really interested me. His availability to help me and to discuss ideas for the developed system was crucial, as it was his technical expertise on the areas of distributed system and security. I would also like to include the *Departamento de Informática*, and also FCT-UNL, for the access to its cluster of computers that allowed me to perform an evaluation with multiple machines. This work falls within the context of the project PTDC/CCI-INF/32662/2017, Lisbon - 01 - 0145 - FEDER - 032662 (SAMOA).

I also need to express a thank you to my university colleagues and friends, since they were there to help me when I needed and to discuss ideas pertaining to the thesis.

Finally, I would like to thank my family, more specifically my parents, my brother and my grandmother for all support given during this critical phase in my life.

Abstract

In recent years, blockchains have grown in popularity and the main reason for this growth is the set of properties that they provide, such as user privacy and a public record of transactions. This popularity is verifiable by the number of cryptocurrencies currently available and by the current market value of Bitcoin currency. Since its introduction, blockchain has evolved and another concept closely linked with it is smart-contract, which allows for more complex operations over the blockchain than simple transactions.

Nevertheless, blockchain technologies have significant problems that prevent it to be adopted as a mainstream solution, or at least as an alternative to centralized solutions such as banking systems. The main one is its inefficiency, which is due to the need of a consensus algorithm that provides total order of transactions. Traditional systems easily solve this by having a single central entity that orders transactions, which can't be done in decentralized systems. Thus, blockchain's efficiency and scalability suffer from the need of time-costly consensus algorithms, which means that they can't currently compete with centralized systems that provide a much greater amount of transactional processing power.

However, with the emergence of novel processor architectures, secure hardware and trusted computing technologies (e.g. Intel SGX and ARM TrustZone), it became possible to investigate new ways of improving the inefficiency issues of blockchain systems, by designing better and improved blockchains.

With all this in mind, this dissertation aims to build an efficient blockchain system that leverages trusted technologies, namely the Intel SGX. Also, a previous thesis will serve as a starting point, since it already implements a secure wallet system, that allows authenticated transactions between users, through the Intel SGX. As such, this wallet system will be extended to provide traceability of its transactions through a blockchain. This blockchain will use Intel SGX to provide an efficient causal consistency mechanism for ordering transactions. After this, the following step will be to support the execution of smart-contracts, besides regular transactions.

Keywords: Blockchain, Consensus, Smart-contract, Trusted Computing, Intel SGX

Resumo

Nos últimos anos, as *blockchains* tornaram-se bastante populares e o motivo é o conjunto de propriedades que fornecem, como a privacidade dos utilizadores e um registo público de transações. Essa popularidade é verificável pelo número de criptomoedas existentes e pelo atual valor de mercado da moeda Bitcoin. Desde a sua introdução, o conceito de *blockchain* evoluiu bastante e surgiu o conceito de *smart-contract*, que permite realizar operações mais complexas sobre uma *blockchain*, além de simples transações.

Contudo, existem problemas que impedem *blockchains* de serem adotadas como soluções convencionais ou como uma alternativa a soluções centralizadas, como o caso de sistemas bancários. O seu principal problema é ineficiência, resultante da necessidade de um algoritmo de *consensus* que forneça ordem total das transações. Os sistemas tradicionais resolvem esse problema facilmente, sendo que têm uma única entidade central que ordena transações, o que não pode ser feito em sistemas descentralizados. Assim, a eficiência e a escalabilidade das *blockchains* sofrem com a utilização de algoritmos de *consensus* dispendiosos, o que significa que não conseguem competir atualmente com sistemas centralizados que fornecem uma maior quantidade de poder de processamento transacional.

No entanto, com o aparecimento de novas arquiteturas de processadores, *hardware* seguro e tecnologias de computação confiável (por exemplo, Intel SGX e ARM TrustZone), tornou-se possível investigar novas formas de melhorar os problemas de ineficiência dos sistemas de *blockchain* e a construção de sistemas melhores e mais eficientes.

Assim sendo, esta dissertação visa construir uma *blockchain* eficiente com recurso ao Intel SGX. O ponto de partida será um sistema de *wallet*, que permite transações autenticadas entre usuários através do Intel SGX, desnvolvido numa dissertação anterior. Como tal, esse sistema será estendido para fornecer rastreabilidade das transações através de uma *blockchain*. Esta *blockchain* utilizará o Intel SGX para fornecer um mecanismo de consistência causal eficiente para a ordenação das transações. Depois disto, o passo seguinte será suportar a execução de *smart-contract*, além de simples transações.

Palavras-chave: Blockchain, Computação Confiável, Intel SGX, Consensus, Smart-contract

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

The first cryptocurrency solutions can be traced back all the way to the 1980's [25], where they used public key cryptography as it's core. In general, these solutions allowed users to sign their owned currency tokens and to send them to some vendor, as a form of payment. This way, the vendor could verify the signature's authenticity. But one problem remained, namely the double spending problem. This problem consists of sending the same digital token to multiple vendors being that they have no way to verify that such situation is indeed happening. While this is easily solved by centralized solutions, since a third party (e.g. a bank) can mediate the conflict, the problem becomes significantly harder if decentralization is a system requirement.

In 2009, Bitcoin proposed a distributed system that solved the double spending problem [24]. Bitcoin was introduced as being an "electronic payment system based on cryptographic proof instead of trust" [24], leveraging novel mechanisms to achieve it. It used a global data structure that recorded all transfers in blocks and these blocks would be chained together, in order to build an ordered chain. This structure would be known as a blockchain. Since all transfers are recorded in the blockchain, it's possible to verify if a given user has already spent a token and thus solve the double spending problem.

In recent years, Bitcoin's popularity skyrocketed and, currently, a single Bitcoin token is worth thousands of euros. As such, other cryptocurrencies have also gained popularity (e.g. Ethereum [38]).

Since its introduction, blockchain systems have evolved and, according to Swan [34], they can be categorized into three stages: 1.0, 2.0 and 3.0. Blockchain 1.0 refers to the earliest systems, such as Bitcoin, which were only used to deploy cryptocurrencies in financial applications. Blockchain 2.0 extends the previous by adding support to decentralized code execution, or in other words, smart-contracts. An example of this is Ethereum. Lastly, Blockchain 3.0 is the application of the blockchain systems to areas other than currency and finance, such as government and health.

1.2 Motivation

If a comparison is to be made between blockchains and traditional banking systems, the former holds many advantages over the latter: their integrity is completely verifiable by anyone due to its public nature and, being a decentralized system, it does not have a single point of failure and thus, it's resilient to crashes and other types of failures. This solves many of the problems that current banking systems have. Unlike blockchain, financial solutions associate the contents of an account to a person, which means that in case of the system being compromised, the clients identities can be compromised. Another problem is their centralized nature, due to the possibility of a single failure crashing the whole system.

Even though all this, blockchains also have some significant problems. The main one is its scalability [11]. More specifically, the time necessary to confirm transactions and, in this aspect, a blockchain is far behind centralized systems. While centralized systems can have a single entity deciding on its state, a blockchain does not have that privilege. It needs to have some sort of consensus mechanism to ensure that all users agree on the same view of the system and that is very time-consuming. Therefore, this has been an area of interest in the recent years and there are many proposals of systems that try to find some way to provide scalable blockchains (e.g. Hedera's Hashgraph [6] and Hyperledger's Fabric [2]), while retaining its central properties.

In the few last years, there has been a rise in trusted computing technology (e.g. TPM [35] and Intel SGX [9]) and with it, a new way of providing consensus primitives to blockchains. An example of this is Hyperledger Sawtooth, which proposes a novel consensus algorithm named Proof of Elapsed Time (PoET) [28]. Succinctly, it consists of electing a leader based on a function executed by all users and the SGX provides a secure area to do it. This way, a user can attest the authenticity and integrity of the result to others, without using an expensive protocol, such as the Proof of Work (PoW). Therefore, it's possible to leverage trusted computing technologies (e.g. Intel SGX) to develop consensus primitives that are both time and energy efficient. However, these solutions still rely on using a distributed ledger for registering operations. This leads to some latency before an operation can be confirmed and makes the system unavailable if it is not possible to contact the ledger. An interesting question is whether it is possible to leverage the trusted computing technology to allow clients to exchange information about transactions directly, making the registry of these transactions asynchronously.

1.3 Objectives

The main objective of this dissertation project is to extend the work started in a previous one, where it was implemented a secure wallet system using Intel SGX. This system allows users to perform money transactions between each other, where the SGX is used to prevent users to perform illegal operations, e.g. perform a transaction without the necessary funds. This is achieved by the creation and usage of a secure container in the wallet's platform, which executes the transaction code and afterwards generates a blob that serves as a proof to other users that it was indeed a legal operation.

The proposed way to extend such a system is to support it with a blockchain. Currently, each wallet only stores its current state and disregards information about past transactions. As a consequence, it provides no traceability of the transactions and the only source of confidence in the validity of operations is the SGX. This would be solved by supporting the system with a blockchain, which would provide a public ledger containing all transactions between the users.

A central component of any blockchain is its consensus mechanism. The consensus mechanism is used to totally order transactions. However, if the validity of operations is confirmed through the SGX, then there is no longer the need to totally order operations, thus the distributed ledger does not need to be totally ordered. Briefly, the idea is: when a user performs a transaction, he sends the transaction's information to the receiver, alongside with a signed proof of its validity. This proof consists of a set of the necessary previous transactions that allowed the user to possess enough funds to perform the intended transaction, signed by the SGX. The blockchain system will serve as a registry of transactions. The goal is that this registry to be causally consistent, so that a user's transaction, t_2, that uses funds received in some prior transactions t_1 always appears in the registry after t_1. This simplifies the problem of developing the blockchains, as it now only needs to provide application-defined causal consistency [4] (and reliable broadcast of all transactions).

The user that performs a transaction will asynchronously register it in the blockchain. It will also directly send the transaction to other parties involved in the transaction. When the user receiving the funds later executes a transaction, it will register not only its new transaction but also the prior transaction, to enforce the causal dependencies. These address the problems that could occur if the prior transactions were not registered due to some failure, but also guarantees that a set of transactions submitted to the distributed ledger can be locally executed without having to wait for dependencies.

Figure 1.1 shows a simple example of the intended system, in which the numbers represent distinct phases. The first one consists of transactions, t_1 and t_2, generated by the system's genesis, to pump currency into clients. Afterwards, Alice performs a transaction t_3 to Bob, while also registering it in the servers, which contain the complete ledger. Both these operations correspond to the second phase. An important detail is that Alice points t_3 to t_1, since this last transaction was what enabled her to perform the other one. The third phase is very similar to the previous one, but this time it's Carol that performs a transaction t_4 to Bob and links it to t_2. At the same time, she registers the operation in the servers. In the end, the servers will have knowledge about the 4 transactions. Since Alice linked t_1 and t_3 and Carol linked t_2 and t_4, the servers will also know that there exists an order between the linked transactions, but not between the pairs. In other words, it's possible to state that t_1 was performed before t_3, but it's not possible to state that it occurred before than the other two transactions. It's important to note that, in case of Bob performing a transaction, he would have to point it to t_3 and t_4, so they serve as proof of how he got the money necessary to transfer it.

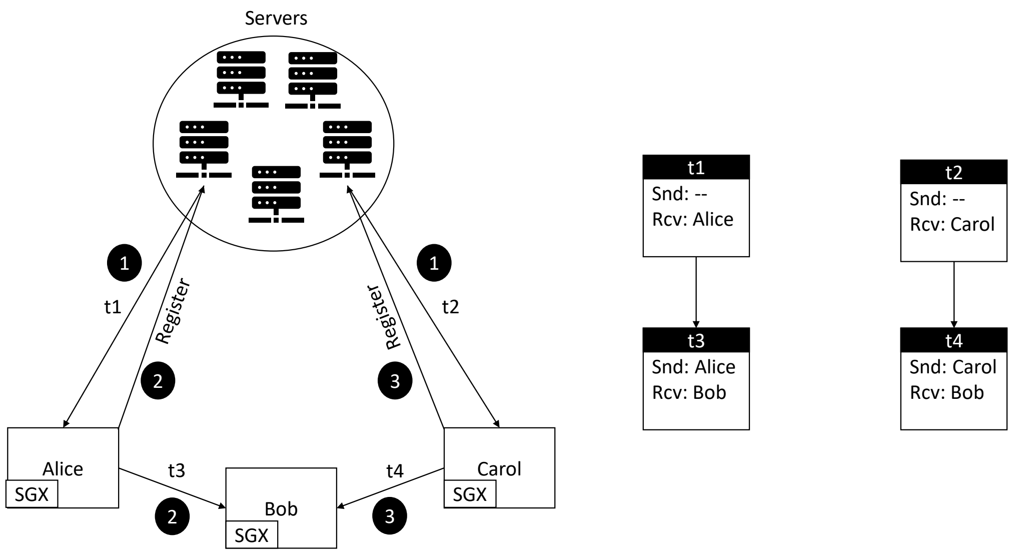

Figure 1.1: Example of a system model, with three clients.

The next step to the implementation of the blockchain would be the extension of the operations allowed by the wallets in order to support smart-contracts. As such, the system could then allow for more complex operations that simple transactions and the smart-contracts could also leverage the SGX, since its code execution could be kept private, by doing it inside the secure container.

In short, the main two goals to be achieved in this thesis are:

1. Implement a blockchain system to support the existing wallet system;

2. Extend the system to support smart-contracts.

1.4 Report Structure

This thesis consists of six chapters, being that Chapter 1 is the current one. In Chapter 2, the necessary related work for this dissertation is discussed, including subchapters about the needed general concepts about distributed systems, blockchain and trusted computing. Chapter 3 describes the overall architecture of the developed system, while Chapter 4 presents a more detailed explanation of the most important components of the system. Chapter 5 shows some performance tests that were executed and Chapter 6 presents some of the core ideas of the thesis and what could be expanded on, in the future.

Chapter 2

Related Work

This chapter is divided into three sections. It starts off by presenting a brief explanation about general concepts of the distributed systems area, in section 2.1. Section 2.2 will discuss what a blockchain is, it's central components and mechanisms, followed by some of existing solutions. The third and final section of this chapter, section 2.3, will present the concept of trusted computing, two solutions that provide it and one system that leverages one of those solutions.

2.1 General Concepts

To better understand the next section, this one discusses basic concepts related with distributed systems. Subsection 2.1.1 addresses the notion of replication, while subsection 2.1.2 discusses some of the failure models assumed when building a fault tolerant system. Subsection 2.1.3 explains the consensus problem and, the final one, subsection 2.1.4 tackles a specific type of fault-tolerance, namely Byzantine fault-tolerance.

2.1.1 Replication

Replication, more specifically data replication, consists of having copies of the same data at multiple machines. This technique is very useful for the deploying distributed systems, since it provides the following highly beneficial properties: performance enhancement, increased availability and fault tolerance [10].

By utilizing replication, it becomes possible to provide a given service in parallel, by having different clients communicating with different replicated servers. Thus, they can share the service's workload between themselves, resulting in the previously mentioned enhanced system performance.

Availability can be described as the proportion of time that a system can be accessed by it's users [10]. In a replicated system, if one of the servers become unable to provide a service, for whatever reason, one of the other ones can do it instead and, therefore, increasing the system's availability.

Regarding the third and last property mentioned, fault tolerance is the ability of a system to always behave as intended, ignoring a given number of failures and their type [10]. This means that, if we assume a maximum number of faults that can occur, it's possible to guarantee the system's correctness and to provide a given service, by having enough replicas.

2.1.2 Failure Models

A distributed system can suffer failures, whether they're caused by process crashes or by communication problems and, as such, it's critical that those systems can still function as intended. So, when building a distributed system, the types of failures that can occur must be properly classified, in order to tolerate them, and this is achieved by having a failure model. Failures can be divided into three main categories [10]:

- Omission failures;
- Timing failures;
- Arbitrary failures.

The first type of failures happens when a message between processes is not delivered and it can be further categorized, depending on whether the root of the problem is in the process or in the communication channel. If the process has stopped working and is halted, then it's either the fail-stop model (other processes can detect that a failure occurred) or the crash model (other processes can't detect the occurrence of the failure). However, if the fault is in the communication channel, the model can be classified as being the send-omission model (the process has completed the send operation but the message has not been put on the sender's message buffer) or the receive-omission model (the message has arrived to the receiver's message buffer, but was not received by the process itself).

Timing failures occur when a message is not delivered in a timely fashion. This type of failure can only happen in systems where exists an assumption of an upper bound on the message delivery time, i.e. in synchronous systems.

The most difficult type of failure to tolerate is the third one, the arbitrary failure, which is also known as Byzantine failure. This one is rather easy to explain, since it means that no assumptions can be made on the type of the error, i.e. any error can happen. In "The Byzantine Generals Problem" [19], these failures are shown and represented as the traitor generals, because they can have any kind of behavior, even a malicious one. In this dissertation, the assumed failure model is this one.

2.1.3 Consensus

Consensus is the problem of reaching an agreement about some value, by a set of processes and through communication. To solve consensus, algorithms must respect the following three properties at all times, according to Coulouris *et al.* [10]:

- Termination: every correct process must decide some value;
- Agreement: the decided value must be equal, for all correct processes;
- Integrity: a correct process will decide a given value, as long as all other correct processes proposed that same value.

A solution to the consensus problem can be used as a key component for providing multiple services regarding distributed systems. Examples of this include state machine replication, leader elections and other services that depend on the agreement of some value.

There's multiple algorithms that solve it, such as Paxos [18] and Raft [26], which make the replicated nodes reach an agreement about a given value and their implementation depend on various factors, such as the chosen failure model and whether it's an asynchronous or synchronous environment. For instance, when developing a consensus protocol, it's crucial to remember the FLP result [14], which states that it's impossible to solve deterministically the consensus problem in an asynchronous system, in which a single process can fail by crash. However, if it's assumed that no failures occur, i.e. all processes are correct, solving it is trivial matter.

2.1.4 Byzantine Fault Tolerance

The Byzantine fault tolerance concept was originally presented in "The Byzantine Generals Problem" [19], where it's explained as the following abstract problem: a set of generals need to agree on the action to execute: whether to attack or not an enemy city and the only way to communicate is through messengers. It's also important to note that generals can either be loyal or traitors, being that the first type will attempt to decide on the best course of action, while traitors will attempt to prevent the loyal generals to achieve that. So, there's two main goals to accomplish, namely the loyal generals have to decide on the same action, while the traitors cannot disrupt the previous ones decision.

All of this can be translated into a computer systems environment: the generals correspond to participating computers, or nodes, in the network, being that the loyal ones are non-faulty and the traitor ones are faulty or Byzantine, and the messengers corresponds to messages between participants. Regarding the general's action, it can correspond to the value to be agreed between nodes.

"The Byzantine Generals Problem" proves that, to solve the presented problem, there needs to exist $3f+1$ total nodes in the system, being that f corresponds to the maximum number of possible faulty nodes [19]. This means that it's impossible to have a system with 3 or less total nodes to decide on a value, in the presence of Byzantine faults.

2.1.4.1 Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance, or just simply PBFT, is a consensus algorithm with the main goal of being deployed on an asynchronous environment and it can handle faulty nodes as long as they represent less than 1/3 of the total number of peers in the system [8]. PBFT was the first algorithm to implement Byzantine fault tolerance in realistic settings, i.e. in asynchronous settings, without relying on synchronous assumptions, henceforth the name Practical Byzantine Fault Tolerance. It's important to note that, since the algorithm is a form of state-machine replication, it assumes that every node has a state and that there's a set of deterministic operations that can be performed to modify it.

In PBFT, there's a needed concept for the algorithm, namely views and its purpose is to allow nodes to know who's the primary of a given view. In other words, each view has a node that's designated primary, while the rest are backups.

Having that said, the base algorithm is divided into the following 3 phases:

- Pre-prepare;
- Prepare;
- Commit.

The client starts by sending a request to the node that he thinks is the current primary and the reason why the client can have an estimate of who's the primary is due to every message received from the replicas containing the current view. When the primary receives the request, it broadcasts to the backups a pre-prepare message with a sequence number, alongside the client's request.

After receiving the pre-prepare message and checking if it's correct, the backups start broadcasting the prepare messages and, consequently, they consider themselves prepared when they receive $2f$ prepares, from other peers. The main goal of both these phases is to order the requests of a given view, though checking in every message received if its view corresponds to the node's current view and if its sequence number is between a certain lower and upper bound.

From the moment that a node is prepared, it broadcasts a commit message and awaits for $2f$ messages of the same type, from other peers. After receiving the needed quantity of commit messages, nodes will execute the requested operation to change its state and send the result to the client, who waits until he receives $f+1$ messages to ensure its correctness.

Figure 2.1 represents this algorithm with a client and four nodes, where node 0 is the primary and node 3 is faulty.

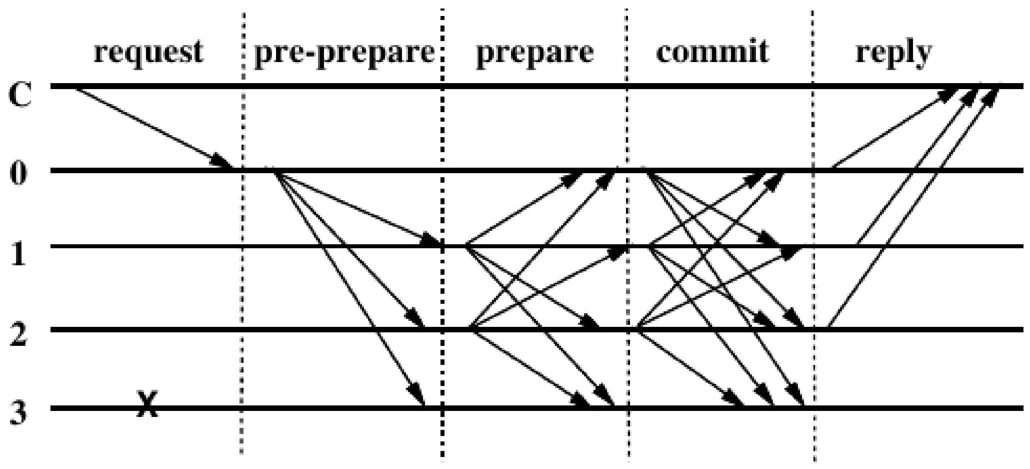

Figure 2.1: PBFT's normal execution case, i.e without primary faults (taken from [8]).

While these three phases ensure the safety of the system, i.e. that the non-faulty nodes can't execute a single wrong step, it's possible to stop it from making progress by attacking the primary node. In order to solve this problem and guarantee the system's liveness, PBFT has a protocol to change the view when a failure is detected on a primary node. Every time a backup node receives a request, it starts a timer and ends it when the request is executed. So, if the timeout is activated before the request's execution, the change-view protocol will begin.

When a node starts executing the previously mentioned protocol, the first step is to broadcast a view-change message, while ignoring other messages that aren't related to the protocol. When the primary of the next view receives $2f$ view-change messages, it proceeds to broadcast a new-view message and when the peers accept this message, they move to the next view. This means that, due to the assumption of the number of faulty nodes being less than 1/3 of the total of peers, eventually the primary node will be non-faulty, which ensures the system's liveness.

2.2 Blockchain

Blockchain is a distributed ledger technology with Byzantine fault tolerance and it's mainly based on two main ideas. The first one is the use of public key encryption, such that each user is assigned a pair of keys, one public and one private, and they are uniquely identified by it. This technique grants properties of authenticity and anonymity to the system. The second idea is a public and immutable register of transactions. This structure alongside public key encryption allows for a system where all transactions are public and, despite of this, users identities are kept secret.

A blockchain is constituted by the following components:

- Ledger;

- Consensus Protocol;
- Virtual Currency.

The first one is a data structure that stores all transactions of the system. More specifically, the ledger consists of a linked list of blocks, where each block contains an ordered sequence of transactions. A fundamental detail of this structure is that each block has a hash of the previous block, which grants the immutability property to the blockchain.

Since the goal of blockchains is to have a distributed ledger instead of a centralized one, it's crucial to have a consensus protocol, enabling the convergence of ledger's state. The chosen protocol depends wildly on the use case for a specific system, but the most common one is Proof of Work, or PoF, which will be explained in 2.2.1.

The third component is optional, since it depends on specific use cases, but in many cases, it's the focus of the whole system. Since most blockchains are utilized to manage financial transactions and wallets, it's necessary to have some kind of virtual currency. In systems that use PoF, it's utilized as an incentive for the users to contribute with computational power to mine blocks.

Blockchains can be separated in two main categories, namely public/trustless and private. The main difference between both is that in public blockchains, the identity of each participant is not known and everyone can participate, while in private blockchains there's a set of identified peers that know each other. This has great consequences on the consensus protocols chosen and can make the difference whether the system should have a virtual currency.

The remainder of this section will present and discuss some examples of blockchain systems.

2.2.1 Bitcoin

Bitcoin [24] is a public blockchain system, whose goal is to eliminate the necessity for third party intervention in transactions between two entities, while also solving the double-spending problem.

The base idea is to order all transactions made chronologically and, to achieve this, PoF is utilized. This is a consensus algorithm that consists of a challenge made to all peers, which requires them to discover the answer to a certain computational puzzle or, more specifically, requires them to discover a nonce that, alongside the block, will generate a hash with a certain number of initial zero bits. Due to the properties of cryptographic hash functions, this makes this challenge computationally difficult, while being extremely easy to verify the correctness of the result. As soon as a peer has succeeded in discovering a correct nonce, the block is added to the chain, with the hash generated by the PoF. This means that, in order to modify a block, it would be necessary to redo the PoF and to modify all the blocks chained after it. But this is not enough to achieve consensus, since it's possible that two blocks have the same parent, meaning that there can be multiple chains. So, to solve this problem, Bitcoin considers that the longest chain is the correct one and the reason for this is that the correct chain will eventually be longer than any other, as long as honest peers control the majority of the CPU power.

Due to the decentralized nature of the system, it was needed some mechanism to introduce coins to the network without relying upon a central authority. Bitcoin's solution to this was to have special transactions that reward block creators with coins and these transactions are always the first ones in the block. Besides this, there's also the concept of transaction fees, which consists of adding the value of a fee per transaction to the block value. So, the idea is to introduce a limited number of coins through special transactions and after introducing them all, each block value will depend solely on the fees. There's an interesting detail regarding the reward system, which is that if there exists an attacker that controls the majority of CPU power of the network, it's possible that he financially profits more from being honest than from invalidating his own transactions and the systems validity.

In spite of providing many interesting properties, Bitcoin has two major problems that prevent it to grow even more and to become the mainstream solution, namely:

- Scalability;
- Energetic waste.

While mainstream payment processors achieve thousands of transactions/second, Bitcoin has a mere maximum of 7 transactions/second [11]. This is due to PoF, since it requires that there is always an interval of about 10 minutes between blocks creation to reduce the number of forks and to improve the fairness of the system.

The energy waste problem is also relative to PoF [37], since it requires that the peers solve a computationally intensive problem and, thus, the energy consumption by miners is very significative.

2.2.2 Hashgraph

Hedera Hashgraph [5, 6] is a public distributed ledger built on top of a data structure named hashgraph and it's very different from traditional blockchain systems. Hashgraph's consensus algorithm is asynchronous, non-deterministic and achieves consensus with probability one, as long as the following conditions are met:

- Assuming f as the total number of nodes in the network, the number of faulty nodes has to be lower than $f/3$;
- If a message keeps being resent, then eventually it will be received by the receiver node, even if an attacker controls the network.

It's based on two different algorithms, namely Gossip about Gossip and Virtual Voting. In the first one, nodes share between themselves information about the transactions that they have knowledge of, through gossip, and, as such, every node has a view of every other node's state. When a node receives information from a peer, it creates an event, whose constitution is shown in Figure 2.2. The transactions in an event are the ones that

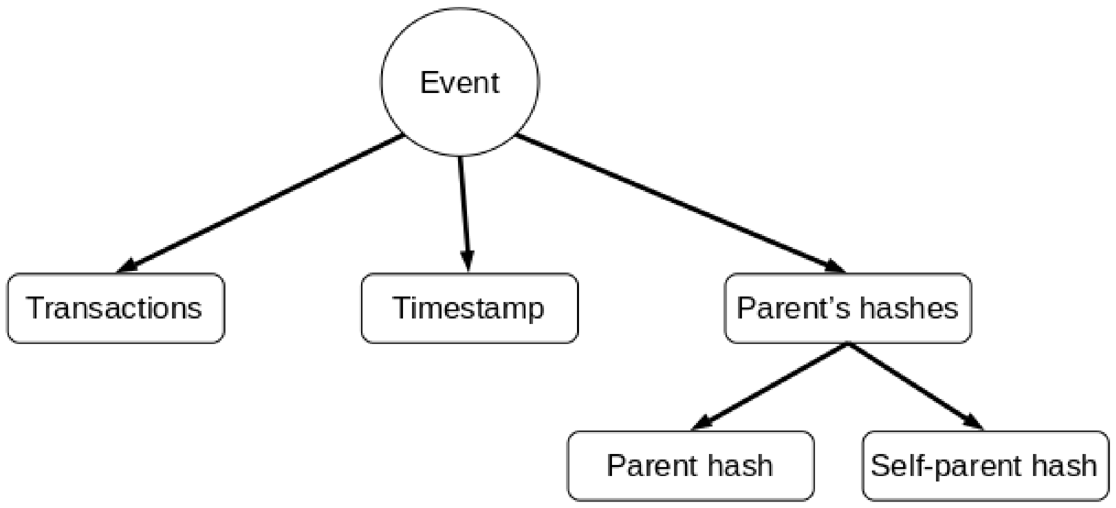

Figure 2.2: Hashgraph's event constitution.

the receiver node has no knowledge about and the timestamp represents the moment of the event creation. Regarding the parent and self-parent hashes, the first one is the hash of the sender node's latest event, while the second one is the hash of the receiver node's latest event.

Through Gossip about Gossip, the peers' view of the hashgraph will eventually converge as long as the assumptions are maintained. But there's still a problem to solve, namely the convergence of their state, or rather, the transactions' ordering. To achieve this, Hashgraph uses Virtual Voting, which consists of an algorithm that is executed locally by every peer when an event is created and is divided into three procedures, namely DivideRounds, DecideFame and FindOrder, performed in this order.

In the DivideRounds procedure, every known event is given a round number and the value is decided based on the parent's maximum number. If an event has no parents, it means that they're the first ones of the graph, so they're given a round number of 1. There's also the possibility of an event's round number being incremented, which depends on whether it can strongly see $2/3f$ witnesses of it's round. Given events x and y, x can strongly see y if there's a path in the graph connecting them that goes through, at least, $2/3f$ peers. Regarding the witnesses, they are the first events of each round and of each peer.

The second procedure, DecideFame, consists of a voting process by all identified witnesses to appoint famous witnesses. Given a round r, the witnesses of r+1 will vote on the witnesses of r and this vote consists of whether they can see them. In other words, a witness of r+1 will cast a positive vote on a witness of r if the first sees the latter, or a negative vote otherwise. Regarding the meaning of an event seeing another, it consists of a weaker version of strongly seeing, in the sense that there needs to exist one path on the graph that connects both events, without being needed to go through a given number of peers. If $2/3f$ peers agreed on a positive decision about an event, then a famous witness has been found, otherwise the events of subsequent rounds of round $r+1$ will vote with the value that has majority, until there's enough votes to make a decision. There's also the concept of coin round, which consists of a round where events vote randomly, and this serves as a measure to prevent attacks where the network is controlled in such a way that the votes are intentionally split.

The third and last procedure of Virtual Voting is FindOrder and, as the name implies, the goal in this one is to find a total order of transactions, by assigning a timestamp to events. To achieve this, there's three concepts that are crucial to understand, namely unique famous witness and received round. The first one is a famous witnesses that is unique to a specific round and a specific node. Given an event x, its received round is the first round where all of its unique famous witnesses can see x. To start off this procedure, for each event, the first step is to calculate its received round and then it's computed the set of timestamps of the unique famous witnesses' self-parents, which observed the event the earliest. As such, the consensus timestamp corresponds to the median value of that set and the last step is to order the events through their timestamps.

Since Hashgraph uses a combination of Gossip about Gossip and Virtual Voting instead of PoF, the throughput it achieves is very high, while also being DDoS resilient due to not using a leader-based algorithm. It also has a low energetic waste when compared with blockchains that use PoF, since there's no difficult computation needed to achieve consensus. But it's important to understand that Hashgraph is a private blockchain, being that the consensus algorithm depends directly on the number of peers known by each node. This means that the system, if applied to a public environment, will be vulnerable to Sybil attacks, which consist of an attacker creating a great number of nodes to overpower the system. Hedera's idea to prevent this was to adapt the base Hashgraph algorithm to use Proof-of-Stake, or PoS, so that each vote power will depend on the voter's stake in the system and to create a council, formed by multiple trusted enterprises and partners from different markets, that owns 2/3 of the total stake in the system.

2.2.3 Hyperledger Fabric

Hyperledger Fabric [2], or just Fabric, is a modular and extensible blockchain system, in which is possible to develop distributed applications using standard programming languages, such as Go and Java, and can also be described as a distributed operating system utilized for building permissioned blockchains.

Currently, most smart-contract blockchain systems use the order-execute architecture [2], as shown in Figure 2.3. This type of architecture is based on propagating the transactions, in a total order fashion, through all peers and then, executing them sequentially on every peer. Even though it's the most used architecture, it has inherent limitations, namely the execution of the transactions by all peers and the requirement of deterministic transactions. Due to this and according to Androulaki *et al.* [2], private blockchains have many issues, including the following:

- Hard-coded consensus algorithms, which should depend of the use case and specific needs;
- Transactions trust model determined by the consensus algorithm, which prevents it from be adapted to specific smart-contracts;
- Smart-contracts written in domain-specific languages;
- Sequential execution of every transaction by every peer, presenting a bottleneck in performance and vulnerabilities to DDoS attacks;
- Deterministic transactions;
- Smart-contracts ran in all nodes, compromising confidentiality properties.

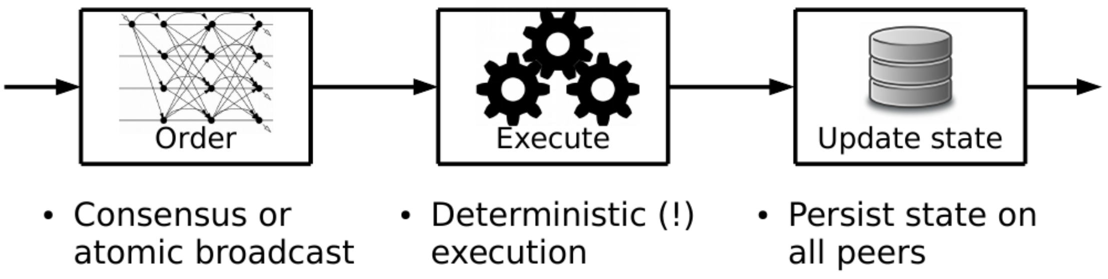
Order
Execute
Update state
• Consensus or atomic broadcast
• Deterministic (!) execution
• Persist state on all peers

Figure 2.3: Order-execute architecture (taken from [2]).

Fabric was developed with the goal of solving the previously mentioned problems, by utilizing a novel architecture, namely the execute-order-validate architecture, shown in Figure 2.4, and also by having the following set of modular components:

- An ordering service, to atomically broadcast state updates and to establish consensus on the order of transactions. It also has the function of reconfiguring the channel (one of the blockchains connected to a particular ordering service) through configuration update transactions made by the members;
- A membership service provider (MSP), to associate nodes with cryptographic identities;
- An optional peer-to-peer gossip service, to propagate the ordering service output;
- Smart-contracts written in standard languages, that are run in containers for isolation.

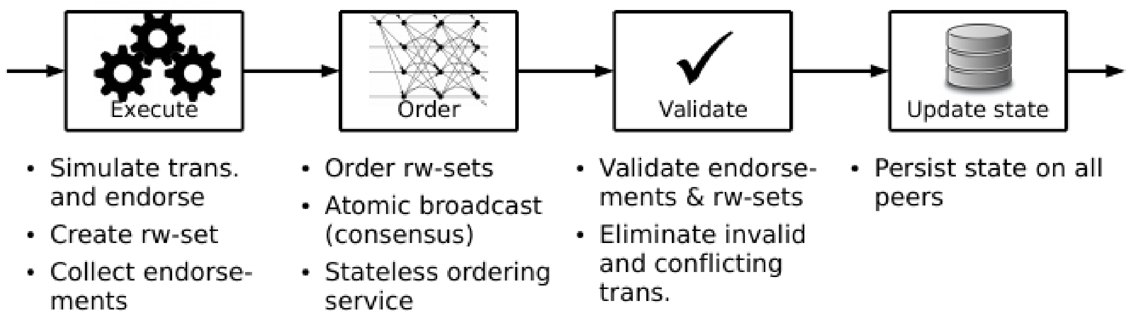
Execute
Order
Validate
Update state
• Simulate trans. and endorse
• Create rw-set
• Collect endorse-ments
• Order rw-sets
• Atomic broadcast (consensus)
• Stateless ordering service
• Validate endorse-ments & rw-sets
• Eliminate invalid and conflicting trans.
• Persist state on all peers

Figure 2.4: Execute-order-validate architecture (taken from [2]).

Regarding the execute-order-validate architecture, it has three phases and the first one is the Execution phase. An important concept for this phase is the endorsement policy, which lets each smart-contract to specify how many endorsers are needed for the transactions and these endorsers are a set of peers used to validate transactions. The Execution phase begins with a proposal submission, made by a client, to the endorsers specified in the endorsement policy and this proposal consists of the client's identity, the transaction payload, a nonce and a transaction identifier. Each endorser proceeds to simulate the proposal on the local state maintained by the Peer Transaction Manager, or PTM, which is a versioned key-value store, and produces a message constituted by a readset and a writeset, named endorsement. The readset is the set of all keys read of the local state alongside their versions, while the writeset is the set of the modified keys alongside their new values. The Execution phase ends with the collection of these endorsements by the client and, if there's enough endorsements to satisfy the policy, then the transaction is sent to the ordering service.

The next phase is the Ordering phase and the goal of this one is to find a total order of the transactions. For this, there's a set of nodes called Ordering Service Nodes and their only purpose is to execute the chosen consensus algorithm to achieve consensus on the transactions order. After an order is agreed, transactions are put in blocks and broadcast to all peers. Fabric also has an optional gossip protocol that can be used instead of a regular broadcast, if needed.

The last phase is the Validation phase and the purpose of this is to check if there's something wrong with the transactions received from the previous phase. It begins by verifying whether the transactions endorsements comply with the endorsement policy of the smart-contract executed and if they don't, then the transactions are not persisted in the system. This is verified by the validation system chaincode which is a static library utilized only for this goal. Next up is the "read-write conflict check", consisting of a comparison between the readset of the endorsement and the local state. A transaction is aborted if the versions contained in the readset are different from the ones in the local state. The final step of this phase is just to apply the transactions to the local state, or in other words, to put the block of transactions into the blockchain, by utilizing the writeset of the endorsements.

2.2.4 Tendermint

Tendermint is a platform based on a novel Byzantine fault tolerant consensus algorithm [7], inspired by the PBFT algorithm [8]. Like PBFT, Tendermint also has a concept of rounds, but with a slight difference: instead of representing a single communication step, it represents a sequence of communication steps. On every round, a node that acts as a proposer and if this proposer is a correct node and there's reliable communication between peers, it's only needed 3 steps to achieve consensus.

Tendermint's system assumptions include the possession of some voting power by every node and the existence of an indirect channel that connects all peers, which is important due to the communication being Gossip-based. Regarding time assumptions, Tendermint has two system variables, namely Δ, that represents an upper bound, and Global Stabilization Time (GST), that represents an instant, and are utilized in the following way: "If a correct process p receives some message m at time t, all correct processes will receive m before $\max\{t, \text{GST}\} + \Delta$" [7].

Regarding the consensus algorithm, it tolerates up to 1/3 of the voting power in faulty nodes' possession and it's round-robin-based, which implies in each round there can be a different proposer, as mentioned previously. Every node has to have the information of the round's proposer's identity and the proposer selection process is based on the peers voting power, meaning that the number of rounds a given node is the proposer is directly proportional to his voting power.

There's three types of messages exchanged between peers, namely Propose, Prevote and Precommit. The first one is sent by the round proposer to suggest a new value, while the other ones consist of votes for a given proposed value. A relevant detail about these messages is that the Proposal carries the value and the other two only carry an identifier of the proposed value, which usually consists of a hash.

On the start of a round, the proposer sends a proposal with a given value and waits for receiving Prevote messages with a given value from 2/3 of the peers. When peers receive this proposal, they check if the value is valid and if it is, then they will send a Prevote message containing the value and set a *timeoutPrevote*, otherwise they'll also send a Prevote message but with a *nil* value, which can also happen due to a *timeoutPropose* set on the start of the round.

When a node receives Prevote messages with a given value from 2/3 of the peers, it sends a Precommit message, sets a *timeoutPrecommit* and proceeds to wait for Precommit messages of, again, 2/3 of the peers. If *timeoutPrevote* expires, then a node will send a Precommit message with the *nil* value.

Finally, if a node receives Precommit messages with a given value from 2/3 of his peers, it will decide on that value and proceed to another consensus instance, which is called height in Tendermint. However, if *timeoutPrecommit* expires, then the node will enter a new round, potentially with a different proposer.

2.3 Trusted Computing

Trusted Computing can be described as the ability of computer systems to protect their data from potential attacks and to perform as expected by a given specification [33], through software and hardware measures, and this concept was first introduced by the former Trusted Computing Platform Alliance (TCPA), whose successor is currently the Trusted Computing Group (TCG), when it released the first specifications of a Trusted Platform Module (TPM) [15].

This section contains the presentation of some relevant trusted computing technology, such as the TPM in subsection 2.3.1 and the Trusted Execution Environment (TEE) in subsection 2.3.2. Afterwards, two solutions that provide TEE will be discussed, ARM TrustZone in subsection 2.3.3 and Intel SGX in subsection 2.3.4, followed by an example of a database system that leverages the second solution, called EnclaveDB, in subsection 2.3.5.

2.3.1 Trusted Platform Module

As mentioned previously, the concept of TPM was introduced by the TCG [15], and it consists of a standardized dedicated microcontroller that can securely generate and store cryptographic keys, among other artifacts [30]. As such, TPM must provide a set of basic features to a computer system, namely authenticated boot, certification and encryption [33], through the set of components shown in Figure 2.5.

2.3.1.1 Authenticated Boot

One of the features that a TPM provides to a system is the authenticated boot, also known as measured boot. This consists of performing an integrity measurement of the boot sequence's components and placing a digest in the Platform Configuration Registers (PCR's), which are secure memory locations in the TPM to store measurements [35]. The goal of this feature is to establish trust on the TPM's system. Since this trust can also be expanded to other hardware and software components, it's possible to create a chain of integrity measurements [36].

2.3.1.2 Certification

The TPM has a unique key pair embedded called Endorsement Key (EK), which is provided by the manufacturer. Since this pair uniquely identifies a TPM and, perhaps, its platform, it's not directly utilized to make digital signatures. Instead, it's possible to generate multiple key pairs from the EK to perform signatures, named Attestation Identity Keys (AIKs) [36].

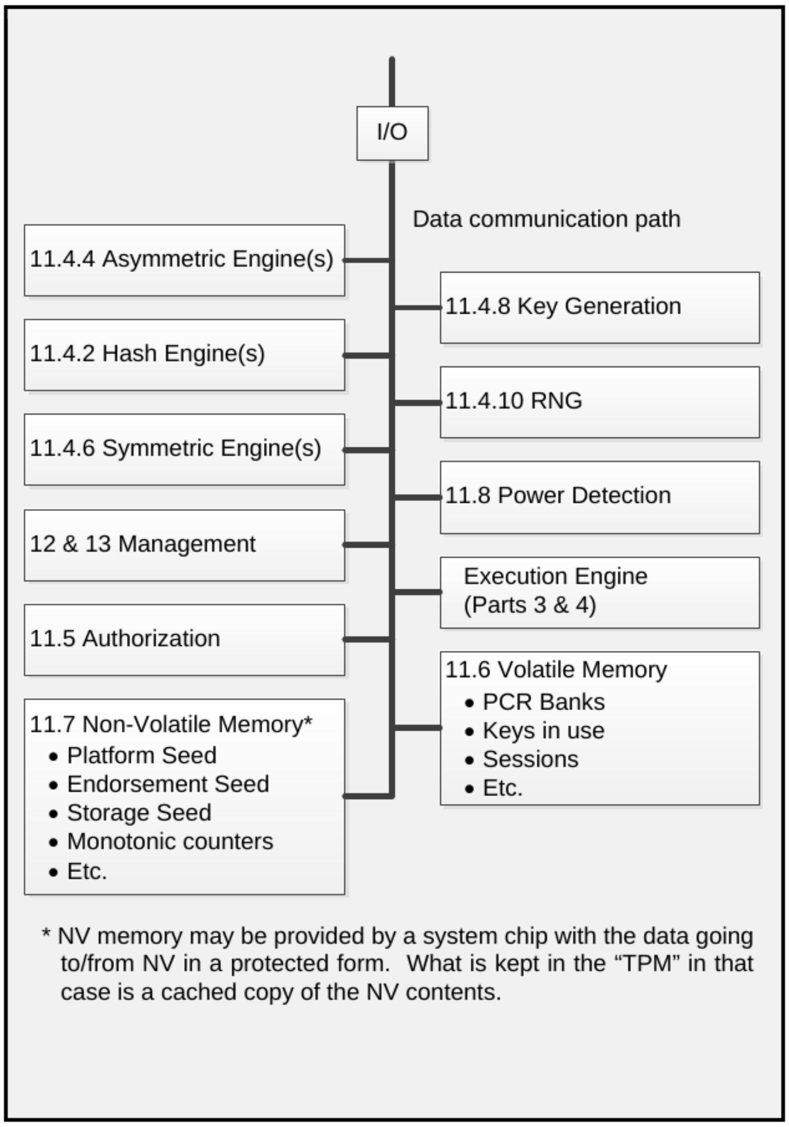
I/O
Data communication path
11.4.4 Asymmetric Engine(s)
11.4.2 Hash Engine(s)
11.4.6 Symmetric Engine(s)
12 & 13 Management
11.5 Authorization
11.7 Non-Volatile Memory*
• Platform Seed
• Endorsement Seed
• Storage Seed
• Monotonic counters
• Etc.
11.4.8 Key Generation
11.4.10 RNG
11.8 Power Detection
Execution Engine (Parts 3 & 4)
11.6 Volatile Memory
• PCR Banks
• Keys in use
• Sessions
• Etc.
* NV memory may be provided by a system chip with the data going to/from NV in a protected form. What is kept in the "TPM" in that case is a cached copy of the NV contents.

Figure 2.5: TPM 2.0 architecture (taken from [8]).

Utilizing these AIKs, the TPM can make signed certificates of the PCR's contents and thus prove to third parties that a given component is trusted by the platform, as long as they possess the public key of the AIK [36]. That is, a third party can request a signed certificate of some component of a given platform and the TPM will provide it [33]. As such, the TPM provides attestation of a platform's components through certification of the same.

2.3.1.3 Encryption

Another feature of the TPM is encryption of data in such a way that it can only be decrypted by the TPM's platform. This mechanism is called binding and utilizes a unique secret key only known by the TPM. As such, it's possible to protect data from being decrypted by any other platform [36]. But this isn't the only mechanism that the TPM provides to encrypt data. It's also possible to specify the configuration, i.e. some PCR's values, of the platform that will perform the decryption, being that this mechanism is called sealing [36]. So, if the configuration of platform changes, it won't be able do access the encrypted data even if it's the intended one [33].

2.3.2 Trusted Execution Environment

A Trusted Execution Environment (TEE) can be described as a secure area in the main processor, where data can be stored and processed securely [16]. This area is isolated from the rich execution environment (REE) [13], which consists of an area where the platform's operating system and applications are running [3], and as such, it provides applications an environment where they can perform sensitive operations. Therefore, it's possible to execute code from third parties, unlike TPM's, which only provide access to a predefined set of API's [31].

The set of security features that a TEE should offer includes platform integrity, secure storage, isolated execution, device authentication and attestation [3].

Platform integrity refers to the ability of a system to verify the integrity of its code and there are two ways of doing this in boot time [3]. The first one is authenticated boot and was already discussed in 2.3.1.1. The second one is secure boot and instead of only measuring the components like authenticated boot, it uses those measurements and compares them to predefined values set by the manufacturer. If it detects differences between the values, it means that some component was modified and it aborts the boot sequence.

A TEE providing secure storage means that data is stored in such a way that its integrity, confidentiality and access control properties are ensured [21]. To achieve this, TEE's usually have a device-specific cryptographic key [3], which is used to encrypt and decrypt data [21]. This key is confidential, set by the manufacturer and stored in a protected memory area in the processor's chip [13].

As mentioned previously, TEE is a secure area in the processor and one of the most important features is the isolated execution of code inside of it. So, TEE's must provide an API, in which sensitive code can be executed safely and isolated from the REE [13]. Besides this, applications must have a TEE code certificate to be able to access the device-specific key and other resources [3].

Another crucial feature is the device authentication and it consists of the ability of an external entity to uniquely identifying the TEE and the associated platform [3]. To accomplish this, TEEs use their device key to authenticate data and since they possess a certificate of that key signed by the manufacturer [3], they can authenticate themselves to third parties as long as these trust the manufacturer.

As explained in 2.3.1.2, attestation is the process of proving the integrity of some component to a third party and TEEs should also provide it. This is done in a similar fashion as in TPMs: send a proof consisting of the integrity measurements taken in authenticated boot, signed by the device key [3].

2.3.3 ARM TrustZone

The ARM TrustZone is a set of hardware security extensions present in ARM processors and its central mechanism is the division of the processor into two areas, namely the Secure World and Normal World [20]. When in the Secure World, the processor runs applications in a secure state, while in the Normal World applications are run in a non-secure state. The difference between these two states is that the normal one can't access some security-critical system registers and processor core bits [27].

There's also a third processor mode that is used for handling world transitions, the Secure Monitor and the switch is made by a privileged instruction, the Secure Monitor Call [32]. But, while this is the case on older processors (Arm Cortex-A), the new generation processors (Arm Cortex-M) architecture doesn't have that third mode, since transitions happen in exception handling code and this means that they happen faster.

Due to the security modes implemented by TrustZone, it can be used to provide TEE features, such as secure boot, isolated execution, secure storage and remote attestation.

The secure boot is achieved by having the processor starting in the Secure World when it's powered on and by initializing the privileged software contained in this World. After this, the processor switches to the Normal World to initialize the rest of the platform's software. During this process, at each component, there's a verification of its integrity and if it detects any modification, the boot sequence is halted. If the boot is successful, then a chain of trust is created [20]. This boot sequence is shown in Figure 2.6.

Isolated execution is achieved by executing trusted applications in the Secure World, where untrusted software can't interfere, not even a privileged one such as the operating system.

To provide secure storage, it's just needed to restrain access to some storage component in such a way that it's only accessible by the Secure World [27].

Since the ARM TrutZone doesn't specify a root of trust [39], it must depend on some component that provides it, such as a TPM. So, it's possible to provide remote attestation if there's such a component that can measure the integrity of the TEE and also give access to unique cryptographic keys [27].

2.3.4 Intel SGX

The Intel Software Guard Extensions (SGX) is a set of instructions and memory changes to the Intel architecture [23] and it allows applications to create protected containers, or more specifically TEEs, called enclaves, to handle sensitive operations and data. To accomplish this, when an enclave is created, its code and data is stored and measured in a special memory location of the Dynamic random-access memory (DRAM), namely the Processor Reserved Memory (PRM), and other software cannot access its contents

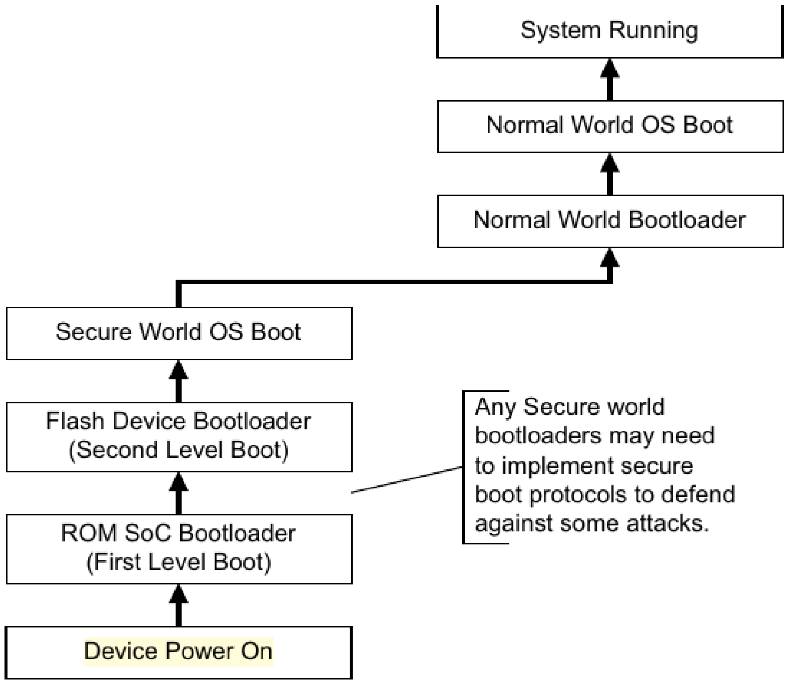
System Running
Normal World OS Boot
Normal World Bootloader
Secure World OS Boot
Flash Device Bootloader (Second Level Boot)
ROM SoC Bootloader (First Level Boot)
Device Power On
Any Secure world bootloaders may need to implement secure boot protocols to defend against some attacks.

Figure 2.6: Trustzone boot sequence (taken from [20]).

[9], even privileged software. The PRM contains the Enclave Page Cache (EPC), which consists of a set of pages, each one storing the contents of a single enclave. To track its content, the processor uses a protected structure called Enclave Page Cache Map (EPCM) and this structure has a set of entries, one for each EPC page [23]. These entries contain information about the EPC pages, such as the page validity, and are used by the CPU in order to enforce access-control on the pages [23]. As such, it's through this memory organization that the SGX provides isolated execution to applications. These memory components are shown in Figure 2.7.

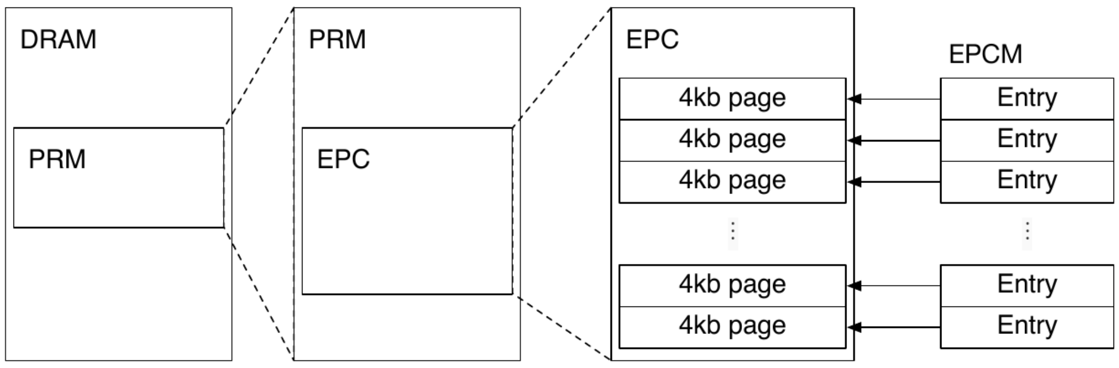

Figure 2.7: Intel's SGX memory organization (taken from [9]).

The SGX also provides attestation of its content and this is achieved by creating an attestation assertion [1]. The structure and creation process of this assertion is shown in Figure 2.8. The MRENCLAVE and MRSIGNER are two measurement registers, that are recorded when the enclave is built, being that the first one corresponds to the identity of the enclave's code and data, while the second one corresponds to an identity of an authority over the enclave [1]. An important detail of SGX's attestation is that the signature primitive can't be implemented in hardware due to its complexity and to solve this, the signature is performed by the Quoting Enclave, which can access SGX's attestation key [9]. This requires secure communication between enclaves and, therefore, SGX has two attestation mechanisms. One creates an assertion between two enclaves on the same platform, which corresponds to local attestation, while the other extends the previous one to other platforms, which corresponds to remote attestation [1].

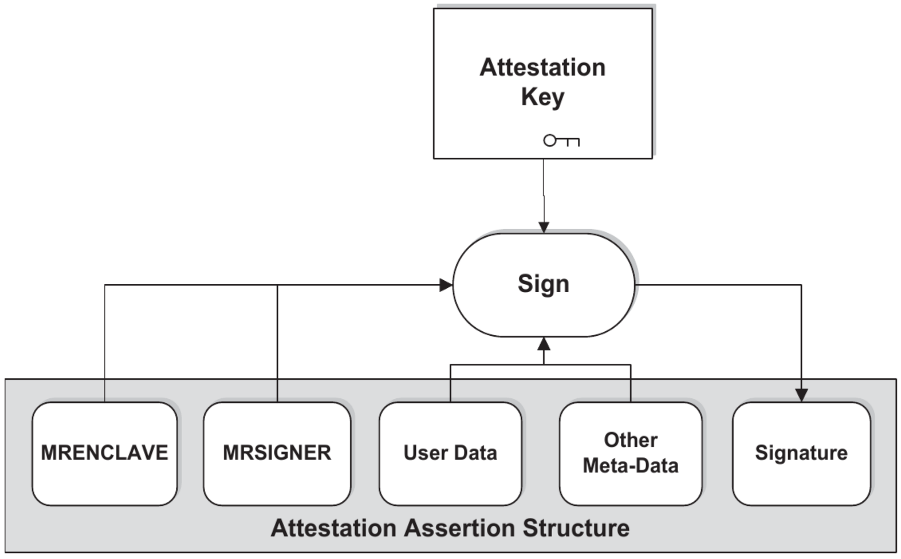

Figure 2.8: Intel's SGX attestation assertion structure and creation process (taken from [1]).

Another feature of the SGX is sealing and it consists of encrypting data to be stored outside of the enclave, which is specially important since all data stored in the enclave is lost when the enclave is destroyed [1]. To accomplish this, the SGX provides access to sealing keys, being that the used key varies depending on the sealing policy chosen. There's two policies, namely sealing to the enclave identity and sealing to the sealing identity [1]. The first one utilizes a key based on the enclave's MRENCLAVE value and therefore every enclave will possess a different key. This means that it's not possible to migrate data between enclaves. The second one utilizes a key based on the enclave's MRSIGNER. As such, keys generated by this policy will allow sealed data transfers between enclaves signed by the same sealing authority [1].

2.3.5 EnclaveDB

EnclaveDB is an in-memory database system that utilizes a TEE, more specifically a SGX enclave, to provide security properties to users data and queries [29]. In Figure 2.9, it's shown an overview of EnclaveDB's architecture.

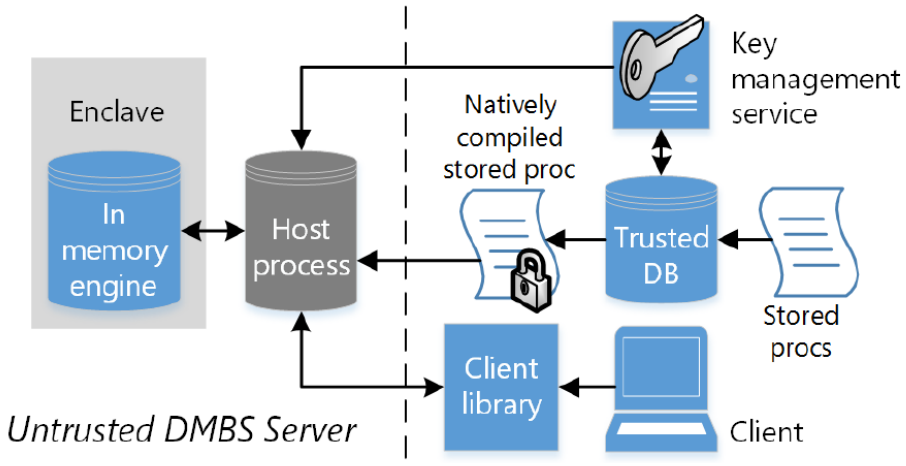

Figure 2.9: EnclaveDB's architecture overview (taken from [29]).

In EnclaveDB's threat model, it's assumed that every component of the hosting server is untrusted, with the exception of the enclave. So, a malicious administrator can't modify any in-memory data or can even compromise privileged software, such as the operating system. It's also possible that database logs are corrupted. On the other hand, the client software stack is trusted.

Due to the trust placed in the system's clients, they're able to protect queries sent to the enclave by encrypt and signing them. This way, a secure channel is established between clients and the enclave, where the database engine is being executed. But before sending the queries, clients pre-compile them, which allows to host some components on it and, by doing this, the attack surface is reduced. The process flow required to execute a query by the client is enabled by the isolated execution feature of the SGX and is essentially the following:

1. The query is pre-compiled, encrypted and signed by the client;
2. Then, it's sent to the enclave;
3. On arrival in the enclave, the query is decrypted and executed;
4. The query result is encrypted again and sent back to the client.

By storing data and executing queries inside the enclave, EnclaveDB provides integrity guarantees. But this also means that data is stored in volatile memory, since an enclave is hosted on DRAM as discussed in Subsection 2.3.4. To prevent data loss and guarantee the durability of the database tables, the system utilizes a database engine named Hekaton [12] and it maintains a persistent log that is used to enable data recovery. Due to the enclave being hosted on an untrusted server, EnclaveDB also provides a protocol to check the log's integrity.

Chapter 3

System Design

This chapter presents the overall design of the system. In Section 3.1, it will be explained the overall architecture of the system, more specifically, the internal components of the servers and their purpose. Then, in Section 3.2, the interaction between clients and the servers will be presented, highlighting the main features that expand the wallet system. Section 3.3, will present some of the thought process that lead to the inclusion of a Message Delivery protocol, while also explaining briefly how the chosen protocol functions, namely the Timestamped Anti-Entropy [22], and which modifications where done to the base protocol. To finish this chapter, Section 3.4 will point out some security-related details that where included into the system.

3.1 Architecture

As mentioned in Section 1.3, the main goal of this work is to provide a form of traceability to the transactions of a secure wallet system that leverages the Intel SGX. This wallet system allows clients to manage locally their wallet, i.e. to make and receive transactions from other clients. These transactions consist of simple monetary transfers between users. What distinguishes this system is that it uses the Intel SGX to verify the validity of every transaction. More specifically, it verifies the code of the client application to ensure that every entity can trust on the validity of the performed transactions. Through this system, clients can directly send and receive operations among themselves. However, the wallets can't store locally all the transactions that involves them, so they only maintain their current wallet balance and not much more. This is the problem that this work tackles concretely. Through the addition of servers to the overall architecture, the system provides a way for clients to register and store their transactions, thus solving the problem of discarding transactions. This means that the servers operate as a open and public transaction registry, where transactions are backed-up, while clients are free to only maintain minimal wallet state. In addition, since the operations performed by clients have validity proofs from the Intel SGX, servers can trust them and only need to verify the authenticity of said proof.

To meet the requirements of backing up this wallet system, the servers were design in a way that they can be divided into 3 layered components:

- Message Delivery;
- Ledger;
- Wallet Management Application.

The first one, the Message Delivery, is the corner stone of the system, as it provides a form of reliable communication between multiple servers and allows data replication. It disseminates and receives messages to and from other servers through the Timestamped Anti-Entropy protocol [22] (discussed in subsection 3.3.1), while also storing them in stable memory, more specifically in a database. This component interacts with the Ledger component, receiving messages to store and propagate and also delivering messages from other nodes to it. An important detail about this component, more specifically the protocol it implements, is that it provides the eventual state convergence of the membership. Basically, this means that it ensures that all transactions will be disseminated to every node that belongs to the peer network.

The middle one, the Ledger, is a data structure that behaves and functions as a graph, in the sense that it has vertices and links connecting them. The vertices are the operations and the links between them are the dependencies. Therefore, the Ledger leverages the graph-like structure, e.g. checking for dependencies of an operation before inserting it. Regarding the interaction with the previous component, when the ledger adds an operation, it serializes it and hands it over to the Message Delivery so that it can be spread through the network, while also receiving messages from it, where it extracts the operations contained within the message, inserts the operation in the graph and delivers them to the upper layer.

The third and upper component is the one that maintains the state of the applications and handles directly the transaction requests from the clients and processes them. This means that it's in charge of verifying the application logic of the wallet system, such as the authenticity of the request and state computation of each wallet. When it receives a transaction from a client, this layer verifies its validity. If it's valid, then it's converted into a Ledger operation and handed over to the Ledger. The application takes advantage of the dependencies that the Ledger operations have to use them as transaction dependencies and meeting the requirements of the system. As mentioned previously, the Ledger also delivers operations to the application, which are converted into wallet transactions in order to update the state of the wallets. An important detail about the wallet application is that it exists a special user, the Administrator, that has infinite balance. Thus, through this account, it's possible to introduce money into the system.

In Figure 3.1, it presented a diagram that represents a set of 4 servers, their internal components, i.e. the components discussed above, and how they interact with each other.

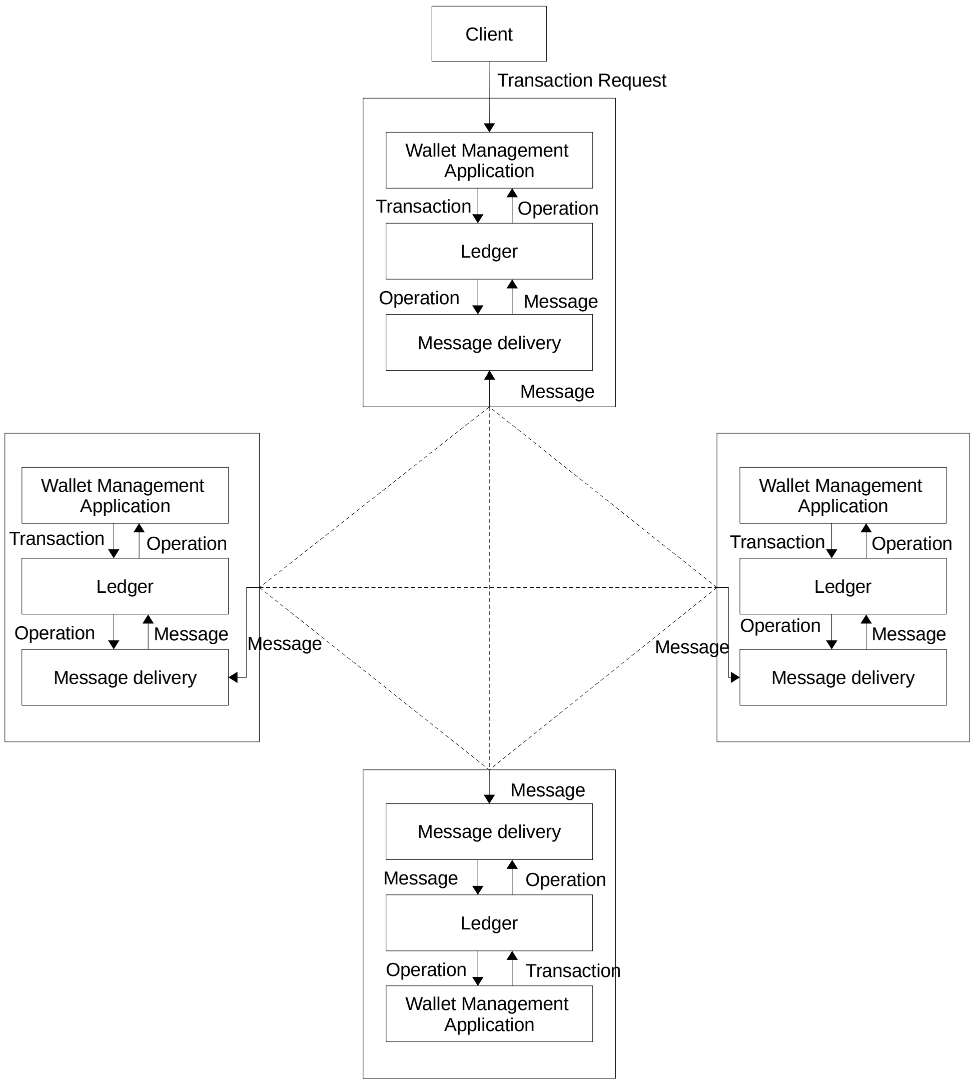

Figure 3.1: Sample system architecture diagram, showing internal server constitution and communication.

3.2 Client-Server interactions

There are essentially 3 ways for clients to interact with the servers, being that the first and most important one is the register of transactions. To be able to do it, the client must have a proof of its validity and authenticity, which consists of the set of identifiers of the past transactions that made it possible to be performed, signed with a key from the source wallet. Thus, by associating transactions this way, the clients build a blockchain that only maintains a causal ordering instead of a total one.

Since the validity of the operations is already verified by the SGX, the only condition that the servers need to check when a client tries to register a transaction is if its dependencies are also registered. If they aren't, then the transaction cannot go through. In this case, the clients has 2 options, to retry to register it regularly until it's successful or to register the missing dependencies directly.

When performing a transaction, it's possible that a server crashes before it propagates it to its peers and/or the receiver is not available at the moment. This brings up another issue: when can clients safely discard information about transactions? As an answer to this problem, the servers provide a durability factor, that represents the percentage of replicas that have successfully received a given transaction. A client can contact a server to request this factor and by doing so, they can make an informed decision whether to discard a given transaction, being that the safest option is to wait until it achieves a factor of 100%. In practice, a client can assume that a transaction is durable after its replication factor reaches a majority of servers.

If, by some reason, a client isn't available to receive transactions directly from other clients in a certain period of time, how can he still obtain them afterwards? To solve this, and representing the third API operation offered, the servers allow the clients to request transactions that affect their wallet. This means that clients that are the source of a transfer can register it on the ledger and receivers can fetch it from the servers if they haven't received them directly - a client can register any transaction it holds, even if it has been received from some other client. To speed up the durability, it's also possible for clients to register the same transaction in multiple servers.

All these interactions are deeply intertwined, since a client can't discard a transaction information without being sure that it's replicated on a large enough number of servers. On the other hand, this ensures that no operation suffers the risk of being permanently lost and that other clients will be able to, eventually, retrieve transactions destined for themselves that were not received directly from their senders.

3.3 Fault Tolerance

It's possible that a server can crash or suffer from some type of failure, which can endanger the availability of the system as a whole. This means that, to prevent it from happening, it's necessary to have some mechanism to tolerate failures, such as data replication between multiple servers. In addition to this, having multiple servers scattered over different geographical areas can improve the latency of the communication with clients. As such, it was considered crucial that the developed ledger system have some form of replication through a set of servers.

With this in mind, the developed system has a component that disseminates transactions over a peer network. This component is based on the Timestamped Anti-Entropy

(TSAE) [22], while having some minor changes, and this choice can be justified by the nature of the ledger. As explained previously, the ledger does not maintain a total ordering of transactions, but a causal ordering. Thus, the system does not require a strong-consistency model, i.e. it's possible that servers states diverge for some time, as long as they eventually converge. Therefore, through the usage of the TSAE protocol, the ledger ensures that transactions are delivered to all of its nodes, in a reliable, eventual and causality-preserving fashion. By having this message delivery protocol instead of some other strong-consistent one, the system can have a better performance.

3.3.1 Timestamped Anti-Entropy Protocol

The Timestamped Anti-Entropy Protocol (TSAE) protocol relies upon 3 data structures: a message log and a pair of timestamp vectors, maintained by each server. The message log is, essentially, a set of all messages generated or received by a server, being that, in this specific case, each message contains a transaction. A message is composed of 4 elements:

- Node Identifier;
- Timestamp;
- Data;
- Delivered Flag.

The node identifier is the identifier of the server where it was initially generated, while the timestamp is the timestamp of when it was generated. Both these elements are the ones that uniquely identify a message, i.e. they allow to safely produce an unique identifer for each message. This means that it's critical that the servers don't generate the same timestamp for different messages to propagate. To achieve this, the system utilizes Hybrid Logical Clock (HLC) [17], ensuring that timestamps are generated safely. Regarding the other two elements, the data represents the data that is to be disseminated to all servers and the flag indicates whether it was delivered to the upper layer or not. It's relevant to mention that the value of this flag doesn't really matter when propagating the message, it's just so that each server can control internally its delivery.

A timestamp vector, with a timestamp per server, can be defined as a snapshot of the system, summarizing a set of messages. The two timestamp vectors have very different roles in the protocol, being that one is the Summary Vector and the other one is the Acknowledgment Vector. The Summary Vector stores information about the latest messages received from every node of the membership, i.e. if entry s_i of the vector has value n_i, this mean that the current server has received all messages from s_i that have timestamp equal or lower than n_i. This allows nodes to exchange between themselves their current knowledge of the system.

While the Summary Vector provides peers with their own system view, the Acknowledgment Vector contains information about the other nodes views. Basically, for a given entry in this vector, the server stores the timestamp that represents the minimum in the Summary Vector of that entry's node. This means that a node can only directly update its own entry of its Acknowledgment Vector, while the other entries must be updated through communication with other nodes. So, it's very important that every node updates its entry of its local Acknowledgment Vector regularly, so that this information can be propagated ot other nodes. By doing this, every server gathers the information about the lowest timestamps present in their peers Summary Vector and thus has information about their system views. The reason why this vector is needed is that, by computing the lowest value in it, it's possible to know that all messages up to that timestamp were already received by every peer and, therefore, can be removed from the log. In our system, as we want to keep messages forever, messages are not removed and are kept. This information is used instead for computing the durability factor of a message/operation. This means that messages are kept in stable storage and allow to reboot a server in the event of a failure.

In order to help better visualize how the Acknowledgment Vector is built, Figure 3.2 presents a simple example.

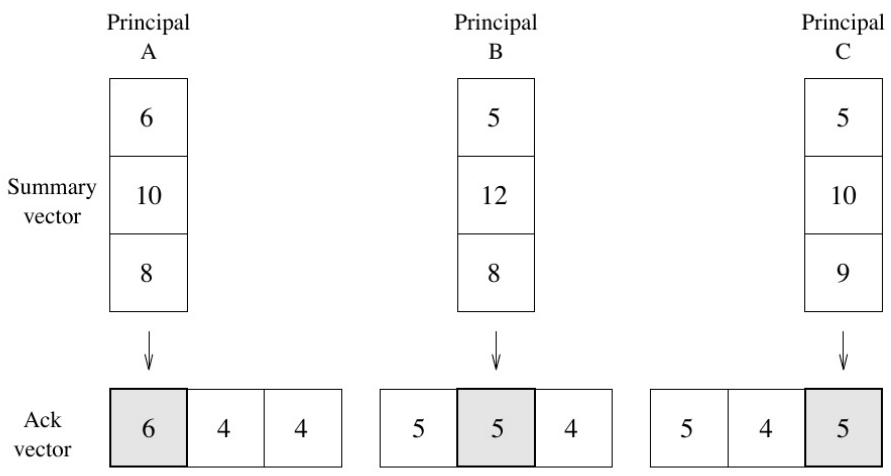

Figure 3.2: Example of both vectors and how the Acknowledgment Vector is built (taken from [22]).

Regarding the protocol itself, it can be described as periodic sessions between pairs of peers. This means that, every so often, a node will select one of its peers to start a TSAE session, which entails, essentially, 3 phases:

1. Exchange of summary vectors;
2. Exchange of log messages and vectors;
3. Exchange of acknowledgment messages.

The session protocol, presented in Algorithm 1, starts with the initiator selecting a node from the membership to be its partners, starting the first phase. In this phase, both nodes create snapshots of their Summary vectors. However, in the original protocol, they would also create a copy of the Acknowledgment vector and update both with a session timestamp. The reason of this modification will be explained ahead. This phase concludes with the exchange of the generated Summary snapshots.

After receiving the peer's summary vector, both nodes initiate the second phase. In this phase, each one determines which messages it has that are not known by its partner. They send to its peer all the locally known messages that the peer has not received yet and new copies of the summary vector and the acknowledgement vector. In the original protocol, this pair of snapshots are generated and exchanged right in the first phase. However, as a node can perform multiple TSAE sessions concurrently, this was modified to cope with possible discrepancy between the Summary snapshot and the fetched messages from the Log. Basically, if the initial protocol was utilized, the addition of messages from other sessions, during the time period between the creation of the Summary snapshot in the first phase and the message extraction in the second one, would be inevitable. So, while the first Summary copy is only used for the extraction of messages, the second one is used to update the local vector.

Before sending the vectors, each peer locally updates them. The Summary vector is updated by updating the timestamp of the local entry according to HLC. The local entry of the acknowledgment vector is also updated.

The final phase is the one where the necessary updates to the actual data structures are performed, i.e. the Log is updated with the messages received from the partner, followed by the update of both vectors. It should be noted that both vectors are updated according to the maximum of the local and received vectors. So, when the nodes end their updates, it means that both now have seen the same set of messages and have matching vectors (assuming no other anti-entropy session has occurred in the meanwhile). To signal the end of this phase and, consequently, the session, both exchange simple ack messages to signal that the updates were performed with success. The only procedure remaining is to deliver the new messages to the upper layer.

Even though the Acknowledgment Vector snapshot is not directly utilized in the session protocol other than to update the real one, it's necessary for calculating the durability factor. However, this was not the case in the original protocol, as it was used to purge messages from the log.

Algorithm 1, only presents the pseudocode of the protocol of the node that starts the session. The code for the peers being contacted for the session is essentially symmetrical to this one. It's important to note that, although not present in the pseudocode, in the case of any error during the protocol, the whole session is aborted and any updates considered without effect.

Algorithm 1: Adapted TSAE initiator protocol (based from [22]).

```
partner := SelectPartner();
// Creates local copy of the Summary Vector
localSum := Summary.clone();
// Sends the Summary copy to its partner
send(partner, localSum);
// Receives the response from its partner
partnerSum := receive(partner);
/* Fetches newer msgs, considering the partner Summary, and consistent copies
   of the Summary and Acknowledgment vectors                                  */
msgs, localSum, localAck := MessageLog.ListNewerMsgsAndVectors(partner);
// Update fetched vectors to have the session timestamp
timestamp := HLC.Advance();
localSum[NodeID] := timestamp;
localAck[NodeID] := GetLowestTimestamp(localSum);
// Exchange messages and vectors
send(partner, msgs, localSum, localAck);
partnerMsgs, partnerSum, partnerAck := receive(partner);
// Update message log with received messages
MsgLog.AddBatch(partnerMsgs);
/* Update own vectors with received vectors, through the maximum value of each
   entry                                                                      */
Summary.UpdateMax(partnerSum);
Acknowledgment.UpdateMax(partnerAck);
// Exchange an ack value to finish protocol
send(partner, "ack");
ack := receive(partner);
// Delivers messages to upper layer
DeliverMsgs();
```

3.4 Security

In terms of security, since the goal of the system is to have a public ledger where every operation is visible to everyone, there was no need to have secure and encrypted communication channels between clients and servers. Nonetheless, every transaction must have a proof of its validity and of its authenticity, to prevent illegal operations. Thus, when a client sends a transaction register request to the servers, it must include a digital signature.

However, the transaction and its signed proof aren't enough to complete the register, it's also necessary to send alongside the clients public key certificate, so that the server can attest the authenticity of the requested operation. This is due to the servers not storing internally the certificates of its users, so it's required that, for each transaction, the client sends its own certificate in the request. But there's still a crucial detail about certificates: they must be issued and/or signed by a designated Certification Authority (CA), which serves as a form of proof that the user was accepted and registered as client of the system. This is useful since the ledger is intended to be a registry of transactions for a game, which usually requires user registration.

It's also important to note that servers also have their own public key certificates, which are utilized to authenticate anti-entropy sessions. When a server tries to initiate a session, the first request must be sent alongside a signature of itself and, contrary to what happens in the register of transactions, there's no need to send the certificate, since it's assumed that the servers already hold their peers certificates.

CHAPTER 4

System Implementation

This Chapter presents the system implementation. As such, Section 4.1 will start by introducing what tools were utilized during development, while also trying to briefly explain the reasoning behind these choices. This is followed by Section 4.2, which will exhibit the state maintained and the API's of each internal component. To close this Chapter, Section 4.3 will expand on how the most important client requests are handled by the servers.

4.1 Development Tools

The first tool chosen for the development of this system was the programming language, being that the choice was Go. The main reasons for the usage of Go were its easier and different handling of concurrency when compared to some more traditional languages, e.g. Java, but it also present a very simple and direct syntax, which can boost the development time. Besides this, it also utilizes a garbage collector in its memory management, eliminating some concerns that other languages have, such as C. To complement all this, it's currently a very popular language, which means that it's possible to find many projects and discussions about it online.

To handle the communication between clients and servers, it was used REST, as it is the standard architectural style. It provides stateless communication and fits well with the poerations exposed by the server. To complement this, to structure the data exchanged between clients and servers, it was used JSON, as it's very simple to use and very popular as a data format, while also being independent and not tied to any programming language or platform.

Servers also had to communicate between themselves, to complete the Message Delivery protocol, namely TSAE. It was possible to use once more the combination of REST and JSON to achieve this, but a different choice was made. As a substitute to REST, it was used TCP directly and the reason for this is that it simplifies communication during TSAE sessions, since each one contains multiple communication exchanges, and being REST a stateless style, it didn't fit very well for this protocol as it's necessary to maintain some data for each session. Regarding the data format used for this it was chosen Gob. This consists of a package developed by the actual team of the Go language, to provide a Go-specific encoding. Since the servers that communicate between themselves are entirely written in Go, they do not need to use a language-independent format and by doing this, it makes the communication more efficient.

As any real system, this one can't really function only in memory and needs to store data on stable storage. So, the obvious decision was to use a database to handle the storage. For this, it was used SQLite and the main reason for this was that it's an embedded database. This means that, instead of creating a database server, like MySQL or PostgreSQL, it functions like an application library and needs no configuration to start to use it. This makes the deployment of the system easier and allows to focus on its actual development.

Regarding cryptographic tools, the main ones were the hash function SHA-256 and the public-key algorithm RSA. Both were used together to create digital signatures when needed. As such, servers and clients of this system are assumed to have RSA key pairs that identify them and they need to be signed by a designated CA. For development purposes, it was used a self-signed CA and the servers store their key pairs, alongside with the public-key certificates of the CA and the other membership nodes, in their own JCEKS keystores, as it provides an easy way to retrieve keys protected by password.

4.2 Server Internal Components

Now, the internal components of the system are described.

4.2.1 Message Delivery

The Message Delivery component is in charge of essentially 2 roles: the first one is to propagate messages to other servers of the group and the second is to store these messages in stable memory, i.e. in the database. In order to fulfill both of these jobs, it requires a set of elements that must be given by its upper layer, namely:

- Membership information, such as TSAE dedicated TCP endpoints and Public-Key certificates;
- RSA Private Key;
- Database connection;
- Channel for sending new messages.

The membership information is crucial since nodes need to know how to contact their peers and to authenticate messages using the certificates. The RSA private key is necessary so that it can create digital signatures that proof the communication authenticity, while the database connection is so that it can perform operations over the database. The channel is a data structure specific to the Go programming language that behaves like a channel, in the sense that data can be sent and retrieved from it by different threads. So, when this component delivers messages, in fact it puts them in this channel so that the other component can extract them.

These elements are essential to the boot the Message Delivery component and are part of the state maintained by it. This component also creates and maintains a HLC, so it can generate timestamps in a safely fashion.

The process of the Message Delivery startup is presented in Algorithm 2. First, it tries to create the necessary database tables, if they don't already exist, which includes the 3 data structures essential to the TSAE protocol discussed previously, i.e. the Message Log, the Summary vector and the Acknowledgment vector. However, while this component uses them as data structures, they're actually just interfaces that perform queries through the database connection, not retaining any data in volatile memory. So, these are what powers the whole system and, if it crashes or is rebooted by some reason, they contain all the necessary information in order to resume the state of the upper layer components. The next step is to check for any messages in the database that were already delivered to the upper application and, if there are any, retrieve them. Since this component doesn't store the HLC state in the database, it uses the retrieved messages to find the highest value timestamp that was issued by itself. The booting process finishes by redelivering the messages that the upper layer has already received previously, so that it can resume from where it stopped. An important note about this component is that it links the database connection to the Message Log and both vectors, in a way that all their operations are performed in the database.

Algorithm 2: Startup process of the Message Delivery component.

```
// Creates the database tables for the data structures
MessageLog.CreateTable(dbConn);
SummaryVector.CreateTable(dbConn);
AcknowledgmentVector.CreateTable(dbConn);
/* Retrieves messages that were already delivered, so that it can redeliver them, if
   there are any                                                                   */
messages := MessageLog.GetDeliveredMessages();
/* Searches throughout the retrieved messages for the highest timestamp issued
   by this node                                                                    */
timestamp := GetHighestTimestamp(messages);
HLC.SetState(timestamp);
// Delivers messages to the upper layer
DeliverMessages(messages);
```

This boot sequence doesn't start the TSAE protocol communication. The Message Delivery component provides an API, presented in Algorithm 3, that allows the upper layer component to start and stop these communications whenever they deem fit.

Algorithm 3: API provided by the Message Delivery component.

```
/* Starts periodic TSAE sessions and launches TCP server to handle incoming
   sessions                                                              */
Start(frequency);
// Halts TSAE protocol
Stop();
// Adds data from upper layer to be added to log and propagated
AddToLog(data);
// Calculates the durability factor of a set of message identifiers
CalcDurability(IDs);
// Lists messages more recent that the given vector
ListMessagesNewer(vector);
```

In order to initiate the TSAE protocol, this component must receive an instruction to do so, while also receiving the time frequency that it should launch a new session. As it was already mentioned previously, each node can handle more than one session concurrently and to safely handle this, it was used a locking mechanism, so that each session can perform its necessary operations without consistency issues. Since nodes can initiate and receive sessions at the same time, just using locks wasn't a good solution, since it would cause deadlocks and block the whole system. In order to solve this, it was implemented a simple lock with a timeout, i.e. the lock would be requested and the session would wait a given time for it. If the lock wasn't obtained in that time period, than the session would be aborted, preventing the deadlock issue. This lock is used when the component tries to add a new message to the log and in the anti-entropy session, right before fetching the messages that the partner hasn't received. In the first case, the lock is performed without the timeout, since it's required that the message is processed before it can reply to the client, while in the second case it's used with a timeout. While this is not an ideal solution, it was used as a measure to prevent the SQLite database locking issue, that would block the database and the system.

Once the protocol is up and running, it's possible to propagate messages to peer servers. To achieve this, the component provides an operation to other components to add data that they want to disseminate, which essentially creates a new message containing this data and stores it in the Message Log, more specifically, in the database. After this, it's only a matter of time until this message is sent to another node through a TSAE session.

These operations are enough to allow data dissemination and replication. However, there's still a pair of operations that aren't relevant for the TSAE protocol, but that are very important in the scope of the whole system, namely the calculation of the durability factor and the retrieval of messages given a timestamp vector.

The first one utilizes a set of message identifiers, given by the upper layer, to fetch those messages from the database and compares their timestamps to the Acknowledgment Vector. It's important to highlight that it allows to receive a set of identifiers instead of only one because it's possible that the same data is contained in 2 different messages, which is an issue that must be controlled by the upper layer. In a summarized form, this operation compares each message timestamp to the ones present in the Acknowledgment snapshot and sees how many members have seen at least one of the retrieved messages. This was developed with the assumption that the membership is fixed, which means that this value should never decrease between requests.

The second operation utilizes a vector to retrieve messages that are more recent than the timestamps contained on it. This one is simpler than the previous one, as it uses the Message Log to make a query that fetches the wanted messages. When clients request their received transactions, which are more recent of a given vector, it's this operation that this components executes.

4.2.2 Operation Ledger

The Operation Ledger maintains the registry of the application's operations. These operations are composed by the following elements:

- Origin Identifier;
- Timestamp;
- Dependencies;
- Parameters.

Both the origin identifier and the timestamp ensure the unique identity of an operation, since there can't exist a pair of operations that have the same values on both. From the operation, the system generates an identifier following three steps:

1. JSON Serialization;
2. Hash creation with SHA-256;
3. Hexadecimal encoding.

This allows to encode the operation identifier in a string for easy usage. Regarding the dependencies element, it consists solely of the identifiers of the operations that need to exist in the Ledger so that a given operation that depends on them can be added. This is the one that effectively transforms this component into a data structure resembling a graph, by having links, i.e. dependencies, between operations. The parameters field is a map that allow applications to customize the operation in such a way that it fits their business logic. The upper layer component just needs to add a string as a key and whatever data that they want to associate to it to create a parameter. By having this abstraction, it's possible to use the Ledger component to support many different applications, as long as they require a graph-like data structure that maintains dependencies between operations.

The Ledger component needs to maintain a set of elements that constitute its state, namely:

- Operation Graph;
- Set of associations between operation identifiers and lists of message identifiers;
- Message Delivery component;
- Channel for delivering new operations.

The operation graph is, essentially, an hashmap that maps the identifiers of operations to the actual operations. Algorithm 4 exhibits its functionalities.

Algorithm 4: API provided by the Operation Graph.

```
// Tries to add an operation, considering its dependencies
AddOperation(operation);
// Retrieves operation by identifier
GetOperation(id);
// Checks the existence of an operation with the given identifier
Exists(id);
```

Regarding the second element, the Ledger must maintain a set of associations that links each operation identifier to the list of message identifiers where the operation is stored. Since an operation can be contained in different messages, it means that the Ledger can receive two messages with the same operation and thus needs to store both message identifiers.

The Ledger also keeps a reference to the Message Delivery component for propagating operations, as detailed in the previous Section.

Finally, the Ledger keeps a channel to deliver operations to the upper level.

The boot process of this component is quite straightforward, since it only initializes its elements and passes down to the Message Delivery component all elements it needs to startup.

The API that the ledger provides is presented in Algorithm 5.

All these methods are self-explanatory, but there are a couple of details worth mentioning. The first one is that the Ledger uses JSON to serialize operations to pass them into the Message Delivery component. The second one is that, to calculate a given operation durability, the Ledger retrieves all the message identifiers related to that operation and hands them over to the lower component to perform the actual computation. The last one is that it provides to the upper application the responsibility of activating the TSAE protocol, meaning that it can decide not to do so. In this case, the Ledger will behave like a normal data structure that doesn't replicate its state with other nodes. If it does activate

Algorithm 5: API provided by the Ledger component.

```
// Adds an operation, taking into account its dependencies
AddOperation(operation);
// Computes the durability factor of an operation by its identifier
GetOperationDurability(id);
// Retrieves operations using a timestamp vetor
QueryOperationsByVector(timestampVector);
// Starts the TSAE protocol of its Message Delivery component with a given
   interval
StartTSAE(interval);
// Halts the TSAE protocol
StopTSAE();
```

the replication module, it's launched a separate thread whose sole purpose is to handle incoming messages from the Message Delivery channel and to insert them in the Ledger.

4.2.3 Wallet Application

Finally, the component that represents the top layer of the system is the Wallet Application and it handles the clients requests directly. As the name implies, it also is in charge of all the application logic of the system, which includes processing transactions and maintaining the wallets balance state. A transactions consists of the following elements:

- Sender Identifier;
- Receiver Identifier;
- Timestamp;
- Value;
- Dependencies.

The identifiers allow to identify the wallets of both parties involved in the transaction. Similarly to the Ledger operations, the timestamp and the sender identifier are the elements that allow to uniquely identify each transaction, using the same identifier generation process. The value element simply indicates the quantity of monetary funds exchanged between both clients. The dependencies element contains all transactions identifiers that were needed to perform a given transaction, i.e. the list of transactions that generated the funds that the sender is using for the transfer.

Through these elements it's possible to convert a transaction into an Ledger operation and vice versa, being that the match between both sets of elements can be observed in Table 4.1.

The application, requires a set of information that must be provided on a configuration file. This includes:

Table 4.1: Match between the elements of a transaction and an operation.

Transaction	**Operation**
Sender Identifier	Origin Identifier
Timestamp	Timestamp
Dependencies	Dependencies
Value	Parameters["Value"]
Receiver Identifier	Parameters["ReceiverID"]

- Membership information;
- Aliases of the CA and the Administrator in the JCEKS keystore;
- TSAE protocol interval value.

The membership information consists of all the necessary data about each node (server) of the system, such as their TSAE protocol TCP endpoints. Additional configuration includes the endpoint to deploy the REST server, the filepath and password of the keystore and also the password for the node private key pair. The other two information elements consist of the aliases used to retrieve the certificates from the JCEKS keystore, while the TSAE protocol interval value is used to start the periodic sessions.

The application maintains the following information to function correctly:

- Public-key certificates of the CA and the Administrator;
- Set of wallets;
- Ledger;
- Set of associations between transaction and operation;

Since clients need to have certificates signed by a CA, servers must keep the CA certificate in order to authenticate the transaction register requests. Besides the CA, the Administrator certificate is also required, since it’s used to authenticate its requests and to process them in a different fashion than the requests from normal clients. The set of wallets used by the wallet application to maintain the balance of each client, executing the transactions received, whether they came directly from the clients or from the TSAE protocol. In fact, these set of wallets doesn’t have much utility, other than allowing clients to consult their own balance or the balance of another account. The set of associations mentioned is very similar to the Ledger component: when transactions are stored in the Ledger, they are converted to operations, meaning that their identifiers change. This means that the wallet application must make associations between the identifiers of transactions and their corresponding operations, so that it’s possible to perform insertions or queries.

The boot process of the application starts off by using the configuration file to load the keystore and retrieve its own key pair and both the CA and the Administrator certificates, followed by a verification to see if all certificates present in the keystore are signed by the CA. The keystore also has the certificates of the membership nodes, but the application doesn't need them, they are there so it can pass them to the Message Delivery component. This process is shown in Algorithm 6.

Algorithm 6: Boot process of the Wallet Application component.

```
// Loads keystore, private key and necessary certificates
keystore := LoadKeystore(config.Filepath, config.KeystorePassword);
// NodeID is the identifier of the own node
ownPrivKey, ownCert := keystore.LoadPrivateKeyAndCert(NodeID,
  config.Membership[NodeID].KeyPairPassword);
CACert := keystore.GetCert(config.CAAlias);
AdminCert := keystore.GetCert(config.AdminAlias);
// Verification of CA signature in certificates
VerifyCASignature(CACert, AdminCert);
membershipCerts := [];
foreach member in config.Membership do
    cert := keystore.GetCert(member.Alias);
    VerifyCASignature(CACert, cert);
    membershipCerts = membershipCerts ∪ cert;
end
// Initializes ledger
ledger := NewEmptyLedger(config.membership, membershipCerts, dbConn);
```

With everything initialized, the applications provides the server with an API that handles directly the requests from clients. It basically consists of 3 main operations that were already mentioned in the previous Chapter: transaction register, durability factor calculation and retrieval of received messages by timestamp vector. It also includes an additional operation that retrieves the balance of a wallet. This API is presented in Algorithm 7.

Algorithm 7: API provided by the Wallet Application component.

```
// Tries to add transaction to ledger, if request is authenticated and all
   dependencies are satisfied
RegisterTransaction(Transaction, Signature, SenderCertificate);
// Calculates the durability factor of the transaction with given identifier
GetTransactionDurability(TransactionID);
// Fetches the transactions received by a given wallet using a timestamp vector to
   query them
GetReceivedTransactionsByVector(WalletID, Vector);
// Gets the current balance of the wallet with the given identifier
GetWalletBalance(WalletID);
```

4.3 Client Requests Processing

Now, it will be discussed the behavior of the system when handling its most important types of client requests.

Table 4.2 presents the REST endpoints that the system provides for its clients.

Table 4.2: REST endpoints for each request and their HTTP methods.

Request	HTTP Method	Endpoint
Transaction Register	POST	/transaction
Durability Factor Calculation	GET	/transaction/{id}/durability
Received Transactions by Vector	POST	/received_transactions_by_vector

4.3.1 Transaction Register

A transaction register request includes of the following 3 elements:

- Transaction;
- Client public-key certificate;
- Signature of the transaction.

When this request arrives at the server, it's handled by the Wallet Application and the first thing it does is to check if the certificate present in the request was issued or signed by the designated CA. This is followed by checking if the proof contained in the request is authentic, using the certificate and transaction.

Security concerns out of the way, the next step is to convert this transaction into a Ledger Operation. In this process, as the identifiers of a transaction and its corresponding operation are not the same it's necessary that the set of dependencies are accordingly updated. Basically, when converting a transaction to an operation, it's necessary that the transaction dependencies are also changed to its operation counterparts and vice versa. This is the reason why the application stores the identifier matches between transaction and operations. An interesting detail is that, by doing this, it means that the application itself already verified if transaction dependencies are present or not, since it's not possible for the application to have associated an operation identifier to a transaction if it's not already present in the Ledger.

Retrieved the operation dependencies and created the operation itself, it's passed down to the Ledger, and serializes it to JSON, passing it down again to the Message Delivery component. Here, the serialized operation is put inside a message and added to the Message Log and to the database. When the next TSAE sessions are performed, this transaction will be propagated to other peers. After this, the Ledger can finally add the operation and associate its identifier to the message identifier that the bottom component generated. Once again in the application, there's only a couple of things remaining to do. The first one is to update the balances of the wallets involved in this transaction. The second and last one is the association between the identifiers of the transaction and the operation, finishing the processing of the request.

Algorithms 8, 9 and 10 represent the procedures described in each component, while Table 4.3 presents the possible answers given to the clients.

Algorithm 8: Pseudocode of the RegisterTransaction function, in the Wallet Application.

```
// Verifies if cert is signed by CA
VerifyCASignature(CACert, request.cert);
// Verifies if transaction signature is authentic
VerifySignature(req.Transaction, req.Signature, request.cert);
// Fetches operation dependencies identifiers
operationDependencies := [];
foreach dependencyID ∈ request.Transaction.Dependencies do
    operationID := GetOperationIDByTransactionID(dependencyID);
    operationDependencies = operationDependencies ∪ operationID;
end
operation := req.Transaction.ToOperation(operationDependencies);
// Add operation to Ledger
Ledger.AddOperation(op);
// Update wallets balances
if !IsAdmin(req.Transaction.SenderID) then
    Wallets[req.Transaction.SenderID].AddToBalance(−req.Transaction.Value);
end
if !IsAdmin(req.Transaction.ReceiverID) then
    Wallets[req.Transaction.ReceiverID].AddToBalance(req.Transaction.Value);
end
// Associate the operation identifier to the transaction
transactionID := req.Transaction.GenerateID();
operationID := operation.GenerateID();
AssociateOperationID(transactionID, operationID);
```

Algorithm 9: Pseudocode of the AddOperation function, in the Ledger.

```
// Verifies if its required dependencies are registered or not
CheckDependencies(Operation);
// Serializes operation so it can be added to the Message Log
operationJSON := SerializeToJSON(operation);
// Adds operation to TSAE Message Log
messageID := TSAE.AddToLog(operationJSON);
// Adds operation to Ledger and associates message identifier to it
RegisterOperation(operation, messageID);
```

Algorithm 10: Pseudocode of the AddToLog function, in the Message Delivery component.

```
// Creates a message with data, which in this case is the serialized operation, and
    an HLC timestamp
timestamp := HLC.Advance();
message := NewMessage(data, timestamp);
messageID := message.GenerateID();
// Handles the insertion of the message in the log and the update of the
    timestmap vectors
HandleNewMessage(message);
return messageID;
```

Table 4.3: Possible responses to the transaction register request.

HTTP Status	Error	Body
200	-	-
400	Unregistered dependencies	Set of unregistered identifiers
500	Unexpected Error	-

4.3.2 Durability Factor

To obtain the durability factor of a transaction, the request only needs its identifier.

Unlike the previous request, this one doesn't need to be authenticated, since the system is supposed to be open and public. The application only retrieves the operation identifier that matches the transaction and sends it to the Ledger. Again, if the application doesn't find the match, it means that the transaction with the given identifier has not been registered yet. The Ledger only checks that the operation exists and retrieves the identifiers of the messages that contained the given operation, passing them to the Message Delivery component. At last, it's in this component that the calculation is performed. It starts by fetching the messages with the received identifiers and by taking a snapshot of the Acknowledgment Vector from the database. As explained before, this information is used to compute the durability factor, which is returned to the upper levels, and finally to the client.

Algorithms 11, 12 and 13 present the pseudo-code of the processing and Table 4.4 exhibits the possible responses to clients.

Algorithm 11: Pseudocode of the GetTransactionDurability function, in the Wallet Application.

```
// Retrieve operation identifier matching the transaction
operationID := GetOperationIDByTransactionID(transactionID);
// Request Ledger for the durability of the operation
durability := Ledger.GetOperationDurability(operationID);
return durability;
```

Algorithm 12: Pseudocode of the GetOperationDurability function, in the Ledger.

```
// Verifies that the operation with the given identifier exists
Exists(operationID);
// Retrieves set of message identifiers that are associated with operation
messageIDs := GetMessageIDsByOperationID(operationID);
// Requests the Message Delivery component for the durability factor
durability := TSAE.CalcDurability(messageIDs);
return durability;
```

Algorithm 13: Pseudocode of the CalcDurability function, in the Message Delivery component.

```
// Retrieves messages through set of identifiers
messages := MessageLog.GetMessagesByIDs(messageIDs);
// Generates a snapshot of the current state of the Acknowledgment Vector
AcknowledgmentSnapshot := AcknowledgmentVector.Snapshot();
// Increment counter when it's confirmed that a member has seen at least one of
    the messages
memberCounter := 0;
foreach member ∈ Membership do
    foreach message ∈ messages do
        if AcknowledgmentSnapshot[member] ≥ message.Timestamp then
            memberCounter = memberCounter + 1;
            break;
        end
    end
end
// Compute the percentage of members that have seen the message
durability := (memberCounter / length(Membership)) * 100;
return durability
```

Table 4.4: Possible responses to the transaction register request.

HTTP Status	Error	Body
200	-	Durability Factor
404	Transaction Not Found	-
500	Unexpected Error	-

4.3.3 Received Transactions by Vector

For the completion of this request, the client must send the following two elements:

- Wallet identifier;
- Timestamp vector.

The goal is to obtain the transactions that have the given identifier as their receiver, while also being more recent than the timestamps contained in the vector. This allows clients to fetch transactions addressed to them since a given snapshot.

The Wallet Application starts by passing down the vector to the Ledger and this one does the same to the Message Delivery Component. This component uses its Message Log to list all messages newer than the provided vector, through a query to the database, which consists in selecting messages based upon a combination of their node identifiers and their timestamps. After this, the messages are sent to the Ledger, where they're used to extract its operations and to create an updated timestamp vector. Then, back in the Wallet Application, the last step is to convert the operations back into transactions and select the ones for which the receiver match the given one. Finally, the transactions and the updated vector are sent back to the client.

It's possible to observe the procedures discussed above in Algorithms 14, 15 and 16. Since the actual query is implemented in the Message Log data structure, it was also included Algorithm 17. As the previous requests, it's also shown the possible responses to clients, in Table 4.5.

Algorithm 14: Pseudocode of the GetReceivedTransactionsByVector function, in the Wallet Application.

```
/* Passes vector to the Ledger to obtain the required operations and the updated
   vector                                                                    */
operations, updatedVector := Ledger.QueryOperationsByVector(req.Vector);
/* Converts operations to transactions and selects the ones that have a receiver
   identifier equal to the given wallet identifier                           */
receivedTransactions := [];
foreach operation ∈ operations do
    transactionDependencies :=
     GetTransactionsIDsByOperationsIDs(operation.Dependencies);
    transaction := OperationToTransaction(operation, transactionDependencies);
    if transaction.ReceiverID == req.WalletID then
        receivedTransactions = receivedTransactions ∪ transaction;
    end
end
return receivedTransactions, updatedVector;
```

Algorithm 15: Pseudocode of the QueryOperationsByVector function, in the Ledger.

```
/* Hands over the vector to the Message Delivery component so it can fetch the
   necessary messages                                                          */
messages := TSAE.ListMessagesNewer(vector);
// Initializes new vector with the same values as the received one
updatedVector := vector;
operations := [];
/* Extracts operations from messages, stores them and uses them to update the
   vector sent by the client                                                   */
foreach message ∈ messages do
    operation := ExtractOperation(message);
    if operation ∉ operations then
        operations = operations ∪ operation;
    end
    if updatedVector[message.NodeID] < message.Timestamp then
        updatedVector[message.NodeID] = message.Timestamp;
    end
end
return operations, updatedVector;
```

Algorithm 16: Pseudocode of the ListMessagesNewer function, in the Message Delivery component.

```
// Uses the vector to extract messages from the log
messages := MessageLog.ListMessagesNewer(vector);
return messages;
```

Algorithm 17: Pseudocode of the ListMessagesNewer function, in the Message Log. Since the Message Log creates SQL queries, this pseudocode does not represent the actual implementation. However, the logic of the chosen messages is the same.

```
messages := GetAllMessages();
requiredMessages := [];
foreach message ∈ messages do
    /* If the message timestamp is higher that the one in the vector, then it means
       that is more recent                                                         */
    if message.Timestamp > vector[message.NodeID] then
        requiredMessages = requiredMessages ∪ message;
    end
end
return requiredMessages;
```

Table 4.5: Possible responses to the received transactions by vector request.

HTTP Status	Error	Body
200	-	Transactions and Updated Vector
500	Unexpected Error	-

Chapter 5

Evaluation

This Chapter presents an analysis and evaluation of the developed system. Therefore, it begins with Section 5.1, which has the goal of defining the environment used in evaluation. It's followed by Section 5.2, where it explains in detail the tests that were performed, finishing the Chapter with Section 5.3 that presents and discusses the results, trying to highlight the reasons that explain the performance obtained.

5.1 Environment Setup

The environment utilized for the system experiments was a cluster of nodes, i.e. machines, with the goal of separating physically each server. More specifically, 4 nodes were used, all with identical specifications, including an AMD EPYC 7281 CPU, 16 cores/32 threads, a 128 GiB DDR4 2666 MHz RAM, 1.8 TB HDD main disk, a network connection speed of 2 x 10 Gbps and the Ubuntu 19 OS.

Each one of these machines was used to run a server, for a total of 4 servers used in the experiments. Since it was needed to have clients performing requests to the servers, each node also hosted a number of clients, which would vary depending on the experiment.

There are a couple of notes that are worth mentioning. First, an Administrator account was used to inject funds in the system, allowing the clients to perform the needed transactions. Second, the clients did not use the Intel SGX, as they are very simple implementations intended only for testing purposes. Third, with the objective of emulating a real-life scenario, it was intended to simulate network latency between servers and clients, but, due to technical difficulties, this was not achieved.

5.2 Test Descriptions

As already discussed in previous Chapters, the main goal of this system is to provide an efficient solution for backing up the operations carried out by clients of the secure wallet system, leveraging the causal ordering of transactions. Therefore, the main aspect intended for testing was the performance of the servers when handling transaction register requests and, as such, 2 metrics were measured, namely the average latency of each operation request and the throughput that the system was able to achieve. In the experiments we analyzed how the number of servers, the interval between anti-entropy sessions and the ratio of read and write operation influenced the performance of the system.

The other aspect that was also tested was the durability factor. More specifically, the focus of this test was to observe how much time, on average, it took for transactions to achieve a value of durability superior to 50% and this was performed for different numbers of servers and intervals of anti-entropy sessions.

5.3 Test Results

This section will describe the experiments performed to the system.

5.3.1 Transaction Register Tests

The experiments regarding transaction requests were all performed with various numbers of clients communicating with the system, in order to observe how both latency and throughput were influenced and how much concurrency the servers could handle. In each experiment, a given number of clients was connected to each server. We varied the number of clients from 1 to 16, with the maximum throughput being achieved between 8 and 16 clients. Each experiment ran for a period of 2 minutes, in which the clients would continuously perform requests to the system.

The goal of the first experiment was to analyze how both latency and throughput would change for different numbers of servers, being that it was decided to use 1, 2 and 4 servers, as we had access only to 4 machines. As it's shown in Figure 5.1, the throughput increases significantly with the increase of servers available, which is to be expected since it means that there's more processing power. The latency isn't affected as much by the quantity of servers, however it's possible to observe that when the throughput stops increasing, it starts to increase drastically. This is also to be expected, since when the system starts to receive more requests that it can handle, requests start to queue on the server side. Regarding the anti-entropy sessions, 10 second intervals were always used in this test.

In the second experiment, we kept the number of servers constant (4 servers), and varied the interval between anti-entropy sessions between 10, 30 and 60 seconds. The purpose of doing so was to understand the impact that the anti-entropy protocol and its sessions had on the overall performance of the system. The results of this are presented in

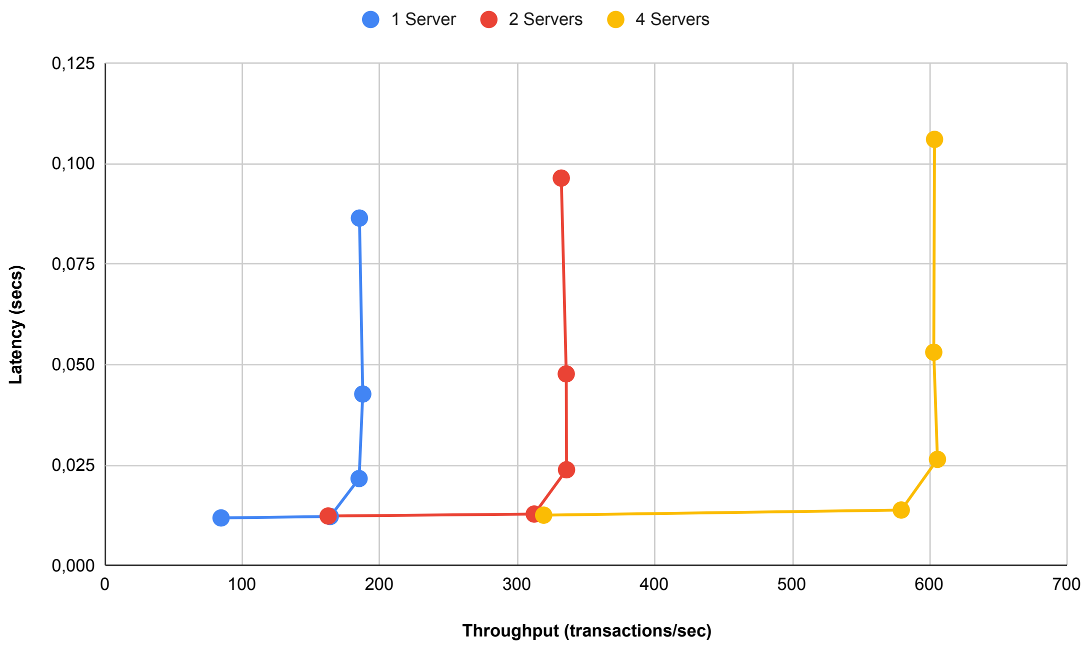

Figure 5.1: Throughput and Latency results from the transaction request experiment with different numbers of servers.

Figure 5.2 and it shows not much of a difference on either the throughput or the latency. The reason for this is that the intervals were large enough to not impact significantly the system, as they should be, since the anti-entropy protocol is computationally costly.

The third experiment consisted of deploying different workloads of write and read operations to the system, being that the write operations corresponded to the submission of a transaction request, while the read operations corresponded to simple fetches of transactions by their identifiers. The goal was to test the performance of the system when handling a write-heavy workload, a read-heavy workload and a balanced workload. As such, the chosen workloads were:

- 90% write operations and 10% read operations;
- 50% write operations and 50% read operations;
- 10% write operations and 90% read operations.

The results of this experiment are shown in Figure 5.3. It's possible to observe that the write-heavy workload result was very similar to the first experiment results, since that was a 100% write operations workload and thus very similar to this one. The balanced and the read-heavy workloads presented great improvements in both latency and throughput values, the first being lower and the latter being higher, meaning that processing the read operations is much easier that to process write operations. This is natural, since

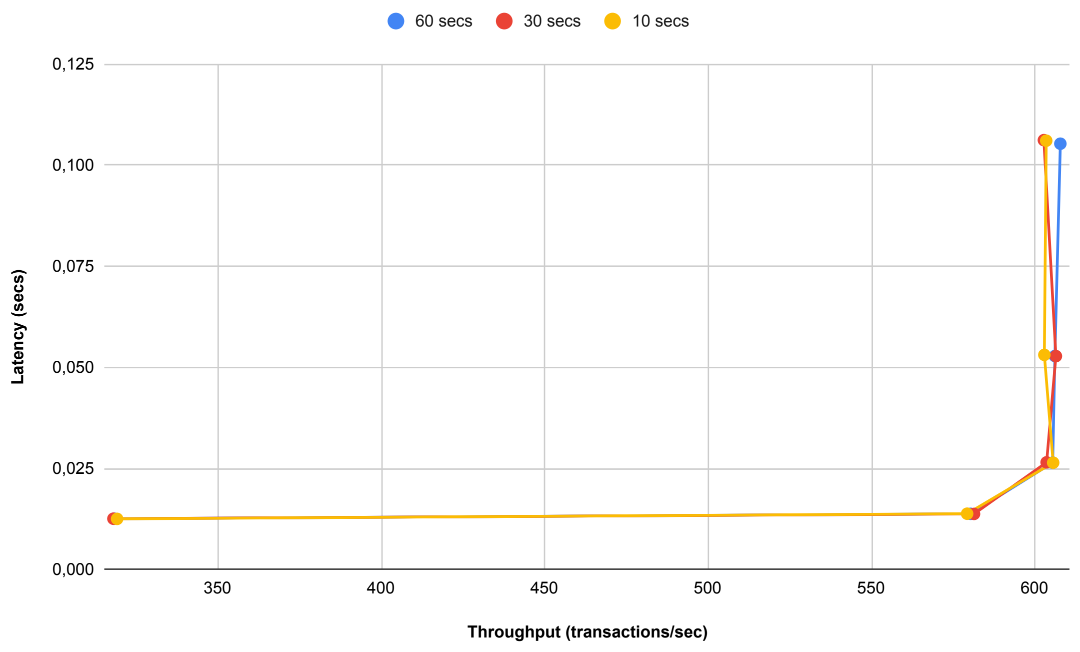

Figure 5.2: Throughput and Latency results from the transaction request experiment with different frequencies of anti-entropy sessions.

the write operations involve writing to the database, while the read operations can fetch transactions from memory.

5.3.2 Durability Tests

As stated previously, the aim of this experiment was to check how much time, on average, a transaction needs until it achieves more than 50% durability. We run the experiments with 3 and 4 servers, while also changing the frequency of anti-entropy sessions between 10, 30 and 60 seconds. This way, it would allow to analyze how would the average time change depending on multiple factors. The experiment in itself was rather simple: each server hosted a client, which would perform a transaction and continuously query its durability factor until it achieved a value superior to 50%. This process was then repeated 10 times to produce an average time, for each combination of session frequency and number of servers. It's important to highlight that what was measured in this experiment is not the actual time that a transaction took to be replicated to a majority of servers, it was the time that the servers took to perceive a transaction to be present in the majority, which is slightly different.

The results of these tests are presented in Figure 5.4. In these, it's clear that the lower the frequency of sessions is, the slower transactions are propagated through the system, resulting in an higher average time to achieve the desired durability. Similarly, the more servers there are, the longer it will take to achieve a given durability factor. Even though it may look like that the higher the anti-entropy sessions frequency, the better, it should

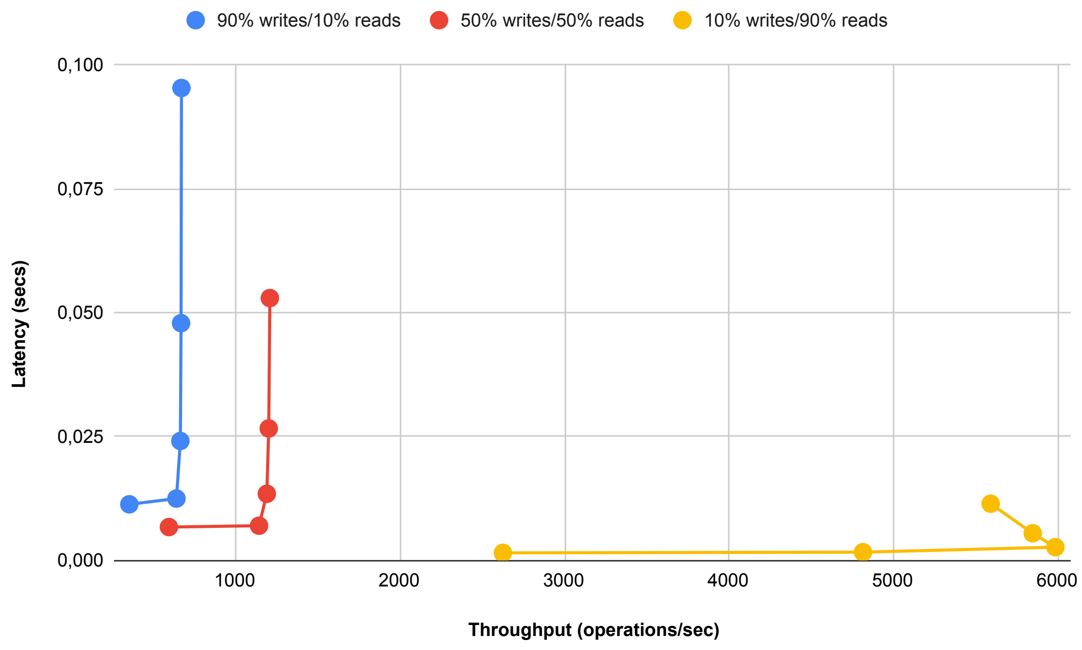

Figure 5.3: Throughput and Latency results from the transaction request experiment with different workloads of write and read operations.

be noted again that anti-entropy is a costly protocol and should be not performed too frequently, which can cause a hit on the performance of a system. So, this emphasizes the need for an additional dissemination protocol to complement it. This way, it could be possible to maintain an high interval between anti-entropy sessions, while also speeding up the rate at which transactions are propagated, thus achieving a given durability value quicker.

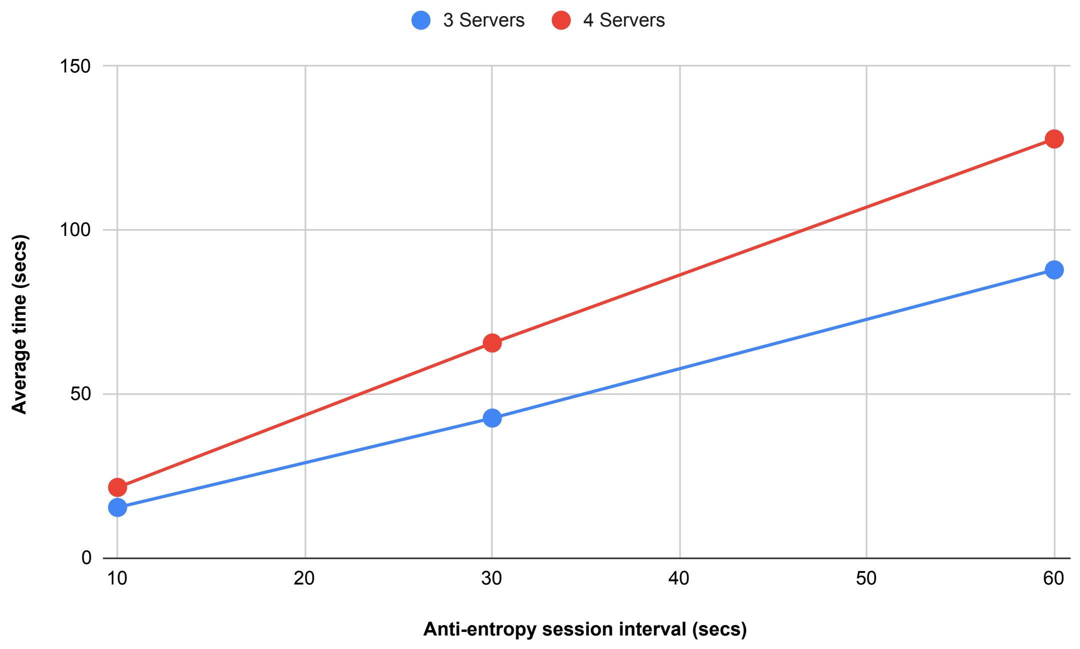

Figure 5.4: Average time for transactions to achieve a durability factor higher than 50% per anti-entropy sessions interval and number of servers.

Chapter 6 Conclusion

In this dissertation, the main goal was to develop a system that extends the secure wallet system, by acting as an open and public registry of transactions. A core element in this work was the Intel SGX, which ensured that every operation performed by a client is valid. This was done by authenticating the client's code and therefore, it enabled anyone to trust the validity of the transactions performed by clients of the wallet system. Utilizing this validity, the focus of this thesis was to implement a system consisting of a set servers that would allow clients to register and fetch transactions. These servers are comprised of three layered components: Message Delivery, Ledger and Wallet Application. The Message Delivery is the bottom layer component and disseminates transactions to other nodes using the anti-entropy protocol. The Ledger is the middle layer component and it's essentially a data structure that stores operations and its dependencies, forming a graph-like structure. This layer receives new operations from the Message Delivery, effectively becoming a distributed ledger. The upper layer component is the Wallet Application and its job is to handle client requests and to maintain the wallets state. It uses the Ledger to store and disseminate the transactions received.

Even though the system doesn't provide an API with many functionalities, it allows clients to solve important issues with their wallets. The first problem it solves is that clients have no form of backing up their transactions, meaning that, after processing them to calculate the balance of the wallet, they are discarded. This means that the information about transfers are lost permanently. As such, the servers provide clients with a transaction register operation. Another problem in the wallet system is that there's no way for a client to receive transactions performed in moment that it was disconnected from the network and this is solved by the blockchain, by providing a query operation to retrieve transfers. The last problem is that a client can't discard a registered transaction right away, since the server that it communicated with can still crash. With this in mind, it was implemented a durability factor, that allows clients to see how many nodes have replicated its transaction, which helps the client do discard them when they are received by a majority of servers.

The results of the experiments performed and presented on this thesis show that the system scales well by increasing the number of servers and that the anti-entropy protocol sessions impose a low overhead on the overall performance.

The prototype built on this thesis exemplifies a form of leveraging the properties of secure hardware to solve some of the inherent scalability issues of blockchain systems in general. In practice, the SGX allowed to implement a blockchain replicated on a set of servers that didn't require a total ordering of the operations. Instead, it uses an application-defined causal ordering, thus not requiring a strong-consistency model and having an improved and efficient performance that it wouldn't have if the blockchain constraints weren't relaxed. In essence, it shows that the usage of secure hardware is a good starting point to develop blockchains that might be geared for some real-world scenarios of permissioned ledgers.

6.1 Future Work

The system does implement the required functionalities to support the secure wallets, but it could still be further improved in a number of ways.

As stated previously, the chosen database was SQLite, mainly due to its ease of use, but, while it allows multiple readers at the same time, it does not handle concurrent writes, meaning that every write is serialized. This can represent a bottleneck in the system performance and, thus, it would be interesting to replace this database with a different one that can handle such writes. Another way to improve the performance of the system would be to add another dissemination protocol, to complement the implemented anti-entropy protocol. This would allow to perform the anti-entropy less regularly and to improve the speed at which the durability of a transaction grows. As verified in the experiments, the number of servers in the system also made the durability factor to increase slower, due to the nature of the anti-entropy protocol. So, in a system that has to handle a very high number of clients, it becomes even more interesting to use another dissemination protocol. In addition to these, a membership protocol to add and remove nodes from the system could also present a great improvement in operation processing.

However, the performance isn't the only thing that can be improved in this system. In terms of features, the system offers a solid and minimal basis: registering transactions, the computation of the durability factor and querying transactions by identifier and by timestamp vector. These could be further expanded on by adding some more complex queries to allow clients to navigate the ledger in various forms.

Currently, the application that utilizes the ledger only stores and processes simple monetary transfers between clients, which could be expanded by having other types of operations stored in the ledger. Due to the nature the ledger, it's possible to store many different types of operations, allowing for developers to build their own applications on top of it. A particularly application for this would be a blockchain system with smart-contracts. Ledger operations could then be used to store the smart-contracts, that would be then disseminated to all nodes of the system. Even though the implementation of smart-contracts was an initial goal, it was not accomplished in this thesis and, as such, it remains for future improvements.

Bibliography

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. "Innovative Technology for CPU Based Attestation and Sealing." In: *Hasp'13* (2013), pp. 1–7. DOI: 10.1.1.405.8266.

[2] E. Androulaki, A. Barger, V. Bortnikov, S. Muralidharan, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Murthy, C. Ferris, G. Laventman, Y. Manevich, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains." In: *Proceedings of the 13th EuroSys Conference, EuroSys 2018* 2018-Janua (2018). DOI: 10.1145/3190508.3190538. arXiv: 1801.10228. URL: https://dl.acm.org/doi/pdf/10.1145/3190508.3190538?download=true.

[3] N. Asokan, J. E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A. R. Sadeghi, S. Schulz, and C. Wachsmann. "Mobile trusted computing." In: *Proceedings of the IEEE* 102.8 (2014), pp. 1189–1206. ISSN: 00189219. DOI: 10.1109/JPROC.2014.2332007.

[4] P. Bailis, A. Ghodsi, J. M. Hellerstein, and R. Waldo Emerson. *Bolt-on Causal Consistency*. 2013. ISBN: 9781450320375.

[5] L. Baird. "Swirlds Hashgraph Consensus Algorithm." In: *Swirlds Tech Report Swirlds-Tr-2016-01* (2016), pp. 1–28. ISSN: 0028-0836. URL: https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf.

[6] L. Baird, M. Harmon, and P. Madsen. "Hedera: A governing council and public hashgraph network - The trust layer of the internet." In: *Whitepaper* (2018), pp. 1–27. URL: https://s3.amazonaws.com/hedera-hashgraph/hh-whitepaper-v1.1-180518.pdf.

[7] E. Buchman, J. Kwon, and Z. Milosevic. "The latest gossip on BFT consensus." In: (2018), pp. 1–14. arXiv: 1807.04938. URL: http://arxiv.org/abs/1807.04938.

[8] M. Castro. "Practical Byzantine Fault Tolerance." In: *Proceedings of the Third Symposium on Operating Systems Design OSDI '99* February (2001), pp. 1–172. URL: http://pmg.csail.mit.edu/papers/osdi99.pdf.

[9] V. Costan and S. Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86. ISSN: 02508095. DOI: 10.1159/000088809. URL: http://eprint.iacr.org/2016/086.

[10] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011.

[11] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer. "On Scaling Decentralized Blockchains: (A Position Paper)." en. In: *Financial Cryptography and Data Security*. Ed. by J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff. Vol. 9604. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 106–125. ISBN: 978-3-662-53356-7 978-3-662-53357-4. DOI: 10.1007/978-3-662-53357-4_8. URL: http://link.springer.com/10.1007/978-3-662-53357-4_8 (visited on 01/10/2020).

[12] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. "Hekaton: SQL Server's Memory-Optimized OLTP Engine." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 1243–1254. ISBN: 9781450320375. DOI: 10.1145/2463676.2463710. URL: https://doi.org/10.1145/2463676.2463710.

[13] J.-e. Ekberg, K. Kostiainen, and N Asokan. "The Untapped Potential of TEEs on Mobile Devices: What is a TEE?" In: *Financial Cryptography* 7859 (2013), pp. 293–294. DOI: 10.1007/978-3-642-39884-1_24.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process." In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382. ISSN: 1557735X. DOI: 10.1145/3149.214121. URL: https://dl.acm.org/doi/pdf/10.1145/3149.214121.

[15] E. Gallery and C. J. Mitchell. "Trusted computing: Security and applications." In: *Cryptologia* 33.3 (2009), pp. 217–245. ISSN: 01611194. DOI: 10.1080/01611190802231140.

[16] *Introduction to Trusted Execution Environments*. URL: https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf (visited on 02/06/2020).

[17] S. Kulkarni, M. Demirbas, D. Madeppa, B. Avva, and M. Leone. "HLC: Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases." In: *Opodis* (2014).

[18] L. Lamport. "Paxos Made Simple." In: *ACM SIGACT News* 32.4 (2001), pp. 51–58. ISSN: 01635700. DOI: 10.1145/568425.568433. URL: https://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/past/03F/notes/paxos-simple.pdf.

[19] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401. ISSN: 15584593. DOI: 10.1145/357172.357176. URL: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/The-Byzantine-Generals-Problem.pdf.

[20] A. Limited. *ARM Security Technology Building a Secure System using TrustZone ® Technology*. Tech. rep. 2005. URL: http://www.arm.com.

[21] H. Löhr, A. R. Sadeghi, and M. Winandy. "Patterns for secure boot and secure storage in computer systems." In: *ARES 2010 - 5th International Conference on Availability, Reliability, and Security* (2010), pp. 569–573. DOI: 10.1109/ARES.2010.110.

[22] D. Long, P. Mcdowell, and R. Golding. "Weak-Consistency Group Communication and Membership." In: (1993).

[23] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. "Innovative instructions and software model for isolated execution." In: (2013), pp. 1–1. DOI: 10.1145/2487726.2488368.

[24] S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System." en. In: (), p. 9. URL: https://bitcoin.org/bitcoin.pdf.

[25] S. Omohundro. "Cryptocurrencies, Smart Contracts, and Artificial Intelligence." In: *AI Matters* 1.2 (2014), pp. 19–21. DOI: 10.1145/2685328.2685334. URL: https://doi.org/10.1145/2685328.2685334.

[26] D. Ongaro and J. Ousterhout. "In Search of an Understandable Consensus Algorithm." en. In: (), p. 18. URL: https://raft.github.io/raft.pdf.

[27] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey." In: *ACM Comput. Surv.* 51.6 (2019). ISSN: 0360-0300. DOI: 10.1145/3291047. URL: https://doi.org/10.1145/3291047.

[28] *PoET 1.0 Specification — Sawtooth v1.0.5 documentation*. URL: https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html (visited on 02/13/2020).

[29] C Priebe, K Vaswani, and M Costa. "EnclaveDB: A Secure Database Using SGX." In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 264–278. DOI: 10.1109/SP.2018.00025.

[30] M. D. Ryan. "Trusted Platform Module (TPM) introduction." In: October (2009).

[31] M. Sabt, M. Achemlal, and A. Bouabdallah. "Trusted execution environment: What it is, and what it is not." In: *Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015* 1 (2015), pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[32] N. Santos, H. Raj, S. Saroiu, and A. Wolman. “Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications.” In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 67–80. ISBN: 9781450323055. DOI: 10.1145/2541940.2541949. URL: https://doi.org/10.1145/2541940.2541949.

[33] W. Stallings and L. Brown. *Computer security : principles and practice*, p. 820. ISBN: 9780133773927.

[34] M. Swan. *Blockchain: Blueprint for a new economy*. "O’Reilly Media, Inc.", 2015.

[35] T. C. G. TCG. “TPM Library Part 1: Architecture.” In: (2014).

[36] A. Tomlinson, K. Mayes, and K. Markantonakis. “Introduction to the TPM.” In: 2007, pp. 155–172. DOI: 10.1007/978-0-387-72198-9_7.

[37] H. Vranken. “Sustainability of bitcoin and blockchains.” In: *Current Opinion in Environmental Sustainability* 28 (2017), pp. 1–9. ISSN: 18773435. DOI: 10.1016/j.cosust.2017.04.011. URL: http://dx.doi.org/10.1016/j.cosust.2017.04.011.

[38] G. Wood and Others. “Ethereum: A secure decentralised generalised transaction ledger.” In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[39] S. Zhao, Q. Zhang, G. Hu, Y. Qin, and D. Feng. “Providing Root of Trust for ARM TrustZone Using On-Chip SRAM.” In: *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*. TrustED ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 25–36. ISBN: 9781450331494. DOI: 10.1145/2666141.2666145. URL: https://doi.org/10.1145/2666141.2666145.