Blockchain as a consensus service

Paulo Jorge Almeida dos Anjos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Miguel Ângelo Marques de Matos
Prof. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. José Carlos Alves Pereira Monteiro
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. José Orlando Roque Nascimento Pereira

October 2018

To my parents

Acknowledgments

This dissertation owes its existence from the proposal made by professors Miguel Pupo Correia and Miguel Matos, who supports me since the beginning gave me many great ideas, enthusiasm and advices.

I would also like to thanks my family and friends for the support that they gave me, not just, during the elaboration of this thesis, but also during my entire life.

Resumo

As criptomoedas, das quais Bitcoin e Ethereum são as mais populares, são baseadas numa abstração subjacente conhecida como blockchain. A blockchain em si é construída de modo a oferecer duas abstrações diferentes, mas complementares: um registo distribuído inviolável e um mecanismo de consenso. As propriedades de consenso da blockchain são significativamente diferentes dos algoritmos de consenso clássicos na literatura de sistemas distribuídos. Além disso, essas propriedades diferem dependendo do tipo de blockchain e do algoritmo de consenso subjacente. Nesta dissertação, exploramos a ideia de como criar um serviço de consenso sobre blockchains existentes e como as diferentes garantias que esta oferece podem ser exploradas pelas aplicações. Devido às propriedades da blockchain, a API fornecida por um serviço de consenso baseado em blockchain é mais complexo do que a fornecida por implementações de consenso mais tradicionais, como Paxos, PBFT ou Raft. O *Blockchain Consensus Service* (BCS), permite que as aplicações equilibrem desempenho e coerência, ou seja, permite que as aplicações mudem de um consenso mais caro que garante uma coerência forte, para um que garante menos coerência e é mais eficiente. Com isto, fornecemos um serviço de consenso com tolerância a falhas bizantinas, uma API clássica que oferece coerência incremental e um registo distribuído inviolável.

Palavras-chave: Blockchain, Consenso, Faltas Byzantinas, Coerência incremental

Abstract

Cryptocurrencies, of which Bitcoin and Ethereum are the most popular, are based on an underlying abstraction known as blockchain. The blockchain itself is built to offer two different but complementary abstractions: a distributed tamper-proof ledger and consensus. The properties of blockchains consensus are significantly different from classical consensus algorithms in the distributed computing literature. Moreover, those properties diverge depending on the type of blockchain and the underlying consensus algorithm. In this dissertation, we explore how to build a consensus service on top of existing blockchains, and how the different guarantees provided by the latter can be exploited by applications. Due to blockchain properties, the API provided by a blockchain based consensus service is more complex than the one provided by implementations of traditional consensus algorithm such as Paxos, PBFT or Raft. BCS, allows applications to balance performance and consistency i.e., applications can change from a costly strong consistent consensus execution to a weakly consistent and more efficient one. With this, we provide a Byzantine fault tolerant consensus service with a classic API that offers incremental consistency and a distributed tamper-proof ledger.

Keywords: Blockchain, Consensus, Byzantine Fault Tolerance, Incremental Consistency

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In 2008, Satoshi Nakamoto presented Bitcoin [1], a peer-to-peer version of electronic cash with the objective of eliminating the need for a trusted third party in on-line transactions. Bitcoin is one of the most popular on-line transaction systems. It works with digital coin and the key technology behind it is the blockchain. The blockchain itself is built to offer two different but complementary abstractions: a distributed tamper-proof ledger and consensus. The blockchain is an ever-growing chain of blocks. This chain works as a ledger because it is unique and stores all the valid transactions.

Software is at the heart of almost everything in a modern society. As a result, applications have gotten more complex. Modern applications are built from many components that cooperate to get things done. The most interesting applications can no longer be executed in isolation (can no longer be standalone applications). Almost none of the mobile phone applications can do anything without talking to other computers. Similarly, most web applications are multiple cooperating processes. Even desktop applications like text editors or calendars use networked services. All applications that need multiple machines, connected with each other, to work are a distributed system.

Distributed applications need to keep a consistent state across replicas or need replicas to come to an agreement about operations. Consensus plays a key role in the development of applications. This happens for two reasons: because most distributed applications need agreement across replicas and because many distributed systems problems are equivalent to consensus. There are a few consensus protocols like Paxos [2], Raft [3] and Practical Byzantine Fault Tolerance (PBFT) [4] that, despite being relatively simple in the normal case, are very hard to understand in the presence of faults. One thing that is more difficult than understanding the algorithms is to implement them correctly. Blockchain is not trivial to understand but its usability can be. This happens despite the eventual complexity of the underling consensus

algorithm and cryptographic methods used. Because of this we believe that a consensus service based on a blockchain system can be relatively simple to implement.

The problem that we will try to solve is: how to offer a classical consensus interface (or close to that) on top of blockchain consensus. We want to provide a classical consensus interface so that a large range of applications can use blockchain to solve consensus. Also, the methods from a classical consensus interface are well known by the community since it has been used for years in several different protocols. Besides being simpler, blockchains like the one used in Bitcoin, also admit inconsistencies for some undetermined time. These inconsistencies make room for speculation. Speculation allows applications to use the time that they are waiting for consensus replies to do useful work. By doing this, applications can save time and be more responsive and available, increasing user satisfaction.

The main issue that we need to overcome in order to provide a consensus service based on blockchain is how to handle blockchain's fork problem. Forks occur when two or more nodes in the blockchain create different versions of the chain of blocks creating several alternative histories of transactions. How can we manage this problem in an acceptable way? Fortunately for us forks happen with low probability (2%) which allows for real systems to function. We present the *Blockchain Consensus Service* (BCS), a service implemented on top of blockchains that allows applications to choose the level of consistency (or trust) that they want for every single execution of consensus by choosing between a weak and a strong consistent result.

1.1 Motivation

The main motivation for BCS was the idea of joining the properties of traditional consensus systems and blockchain systems to provide a new system with a large and interesting range of properties that this two types of systems could not provide alone.

Consensus is one of the fundamental problems in distributed systems. It has been researched for decades due to its importance and relation with other key problems in distributed systems. Problems like atomic broadcast and state machine replication can be reduced to the consensus problem or are equivalent to it [5]. Consensus algorithms are often defined in terms of three properties; validity, agreement, and termination.

The properties that blockchain offers are very interesting and they are not easy to provide. The properties may vary depending on the type of blockchain but the most usual are; anonymity, decentralization, persistency and auditable. But beyond the properties, what is interesting in the blockchain is the distributed tamper proof ledger. The ledger, together with cryptographic methods make the system reliable.

Smart contracts were developed with the objective of allowing the development of Turing complete applications inside the blockchain [6]. This means that they can be used to provide a consensus service. However, we want BCS to be compatible with already existing applications as well as for it to be as generic as possible.

A system that is able to provide the properties of consensus together with the properties of blockchains is a system that; is decentralized, tolerates Byzantine faults, is auditable and persistent. In other words, is a system that applications can trust and will be developed a small cost of adding an abstraction layer to a blockchain system. To the best of our knowledge, BCS is the first system that provides both consensus and blockchain properties with a simple consensus API. The API allows already existing applications to migrate from a traditional consensus algorithm to a consensus service based on blockchain gaining a new set of guarantees.

1.2 Topic Overview

The main topics of this work are: consensus and blockchain.

Consensus is one of the fundamental problems in distributed systems. This problem has been investigated for decades due to its importance and relation with other key problems in distributed systems. Many important distributed systems problems are reducible or equivalent to consensus. Examples are *Atomic Broadcast* and *State Machine Replication* (SMR) [5]. Informally, consensus can be stated as how to ensure that a set of processes agree upon a decision. To better understand the problem, suppose that there is a set of processes that compose a distributed file system; we can think of consensus as a way to ensure that all the replicas of the system contain the same version of all the files. When different replicas contain different versions of the same file, the consensus algorithm comes to play and forces the replicas to reach an agreement on the versions of all the files.

A blockchain is an ever-growing chain of blocks. In cryptocurrencies such as Bitcoin and Ethereum [6] transactions are stored in blocks. The blocks are hashed with a cryptographic hash function. Each block contains the hash of the previous block creating a chain of hashes that is tamper-proof. To break the sequence of hashes, an attacker would have to re-do all the blocks and change their hashes. This chain works as a ledger because it is unique and stores all the valid transactions. The transactions are stored in order. The ledger stores a global unique state of the system. This allows blockchain to solve consensus because all the nodes in the system agree on a single global history.

Remarkably, the properties of some blockchains' consensus are significantly different than classical consensus algorithms in the distributed computing literature. In traditional algorithms,

nodes exchange messages trying to reach agreement. For every instance of consensus there is a corresponding round of messages. When a round finishes, all nodes decide and agree on the same values. This decision is final. As for blockchain based consensus, there is also a node that works as a leader. This node takes decisions and communicates them by sending the next block of the blockchain to the other nodes. The blockchain constitutes a distributed ledger. Nodes consult the ledger and observe a global single history of transactions. Because blockchain works as a peer-to-peer network, a blockchain based consensus service allows the existence of a highly scalable system. The problem is that some types of blockchain systems (e.g. Bitcoin and Ethereum) admit inconsistencies for some undetermined time. These inconsistencies cause the fork problem. This problem does not exist in all types of blockchains.

Forks in the blockchain break the single global history abstraction provided by the chain of blocks. They break the chain into two or more versions creating alternative histories of transactions until they are healed. For this reason, nodes in different points of the network obtain a different history of transactions. Most of blockchain consensus algorithms, those based on *Proof-of-Work* (PoW) and *Proof-of-Stake* (PoS), are easier to understand than the classical examples already mentioned. Another important fact is the existence of large-scale applications (Bitcoin and Ethereum) that use blockchain consensus algorithms, proves that the implementations are correct and practical. Therefore, leveraging the infrastructure of big blockchain platforms is a very appealing and interesting idea.

1.3 Objectives

The objective of this project is to "wrap" a blockchain in order to provide a consensus service. Since we are going to provide a consensus service, we want it to have an interface as similar as possible to a classical consensus interface. Therefore, our solution (BCS) should assure the following system requirements:

1. *Intuitive classical consensus interface* - The API of the service has to be similar to a classical consensus interface. It should also have intuitive names for the proposed methods. This must be done in a way that allows applications that use traditional algorithm to migrate to BCS without any issues.

2. *Deal with forks* - The system must be able to deal with forks in a practical manner.

3. *Consensus* - The solution has to solve the consensus problem in a Byzantine fault tolerant system model.

4. *Byzantine faults* - The service has to be able to handle as many Byzantine failures as the classical consensus algorithm (i.e. components, other than the consensus protocol, must not add constraints to the system model).

1.4 Thesis Outline

The remainder of the document is structured as follows. Chapter 2 describes and analyses the existing systems and mechanisms present in the state-of-the-art solutions for the consensus problem and blockchain. Chapter 3 presents BCS. From the architecture of the system to the applications that can use it. Chapter 4 describes the implementation that we developed as a proof of concept for our solution. In Chapter 5 we evaluate our system regarding performance and the impact of forks. Chapter 6 presents the achievements and conclusions that we achieved with this project and our solution.

Chapter 2

Background

The solution proposed in this document takes advantage of knowledge obtained from the study of state-of-the-art systems and solutions. This chapter presents those solutions and their relationship with our work. This chapter is structured as follows. Section 2.1 presents the consensus problem, gives examples of state-of-the-art classical consensus algorithms, speculative consensus, explains their relation with the proposed solution and discusses their advantages and disadvantages. Section 2.2 introduces the blockchain abstraction. In particular, it discusses the different types of blockchains, blockchain consensus protocols, examples of real systems based on them and the discussion of their advantages and disadvantages.

2.1 Consensus

Consensus is one of the fundamental problems in distributed systems. This problem has been investigated for decades due to its importance and relation with other key problems in distributed systems. Many important distributed systems problems are reducible or equivalent to consensus. Examples are atomic broadcast and SMR [5]. Informally, consensus can be stated as how to ensure that a set of processes agree upon a decision. To better understand the problem, suppose that there is a set of processes that compose a distributed file system; we can think of consensus as a way to ensure that all the replicas of the system contain the same version of all the files. when different replicas contain different versions of the same file, the consensus algorithm comes to play and forces the replicas to reach an agreement on the versions of all the files. Consensus protocol used is the component of the system that ensures that all the replicas in the system contain the same version of all the files.

There are different variations of consensus in the literature, like *binary consensus*, *multi-value consensus*, *vector consensus* and others [5]. One of the simplest variations is the *binary*

consensus but the most interesting one is probably the *multi-value consensus* because of its more practical use. All variations of consensus can be defined in terms of three properties; *validity*, *agreement* and *termination*. Normally the difference between them is the definition of the validity property [5]. For example, a definition of the binary consensus is:

- *Validity:* If all correct processes propose the same value v, then any correct process that decides, decides v
- *Agreement:* No two correct processes decide differently.
- *Termination:* Every correct process eventually decides

Validity and agreement are *safety* properties, while termination is a *liveness* property. Safety states that "nothing bad" should happen during the execution of an algorithm and liveness states that eventually "something good" will happen (Section 2.3.1 of [7]).

We are particularly interested in the vector consensus variation (first presented in [8]) because that is the one that we will try to provide in our solution. Here the validity property is called *vector validity* and its definition is the following [5]:

- Vector validity: Every correct process that decides, decides on a vector V of size n:
  - $\forall p_i$: if p_i is correct, then either $V[i]$ is the value proposed by p_i or $\perp$;
  - at least $(f + 1)$ elements of V were proposed by correct processes.

The value $\perp$ is a default value that does not belong to the set of possible values proposed by clients (i.e. $\perp \notin$ V).

At the beginning of this section, we indicated that the SMR problem is related to consensus. To solve SMR, all client's requests must be totally ordered across replicas. To achieve this requirement is necessary to use atomic broadcast [9]. We have already seen that atomic broadcast is equivalent to consensus. Because SMR "uses" consensus and there are much more implementations of *Byzantine Fault Tolerance* (BFT) SMR protocols, that we will talk about in the next sections, than BFT consensus protocols we present a definition of SMR. SMR is defined by a set of state variables and a set of commands that modifies those variables. These commands should be atomic so that they do not interfere with other commands. The protocol is replicated in a set of servers and all the servers in that set implement the service. All the servers in the set follow the same history if the following properties are satisfied [10]:

1. *Initial state*: all correct servers start in the same state.

2. *Agreement*: all correct servers execute the same commands.

3. *Total order*: all correct servers execute the commands in the same order.

4. *Determinism*: the same command executed in the same initial state in two different correct servers generates the same final state.

Consensus is a problem simple to explain and it is straightforward to solve if, and only if, there are no failures. For instance, suppose that a well-defined set of process exists and that every process reliably broadcasts its proposed value. The processes wait until they receive all values, including their own. After that, processes can apply a deterministic function that receives all values as input and outputs one of the input values. Since the function is deterministic and all processes apply the same function with the same input, the output of the function is the decided value and it is the same for all processes.

A more interesting challenge is to solve consensus in the presence of faults. In fact, in an environment where faults exist, consensus may be impossible to solve. It has been proven that in an asynchronous system where even a single process may crash it is impossible to solve consensus (the FLP impossibility [11]). There are numerous reasons for the existence of the impossibility but the most important is that in asynchronous systems, a crashed process is impossible to distinguish from a very slow one. Despite the FLP impossibility, there are numerous algorithms that solve consensus in the presence of faults [2], [3], [4], [12] and [10]. This is possible because the authors of such algorithms work around the impossibility. One of the most used techniques is to add a time assumption to the system model, making the system synchronous or eventually synchronous. Another popular technique is the use of failure detectors [13]. In [14] the authors present a survey about the techniques that are used to work around the FLP impossibility.

2.1.1 Crash-stop Failure Model

In the *crash-stop failure model* a process is considered to be correct if it executes the algorithm that it is supposed to. A crash failure occurs when a process stops at some point during the execution and never recovers after that point. It is the responsibility of the system designers to predict faults and to define how to respond to them. System designers usually handle crash failures by adding a failure detector to the system. A failure detector is a component of the system whose only function is to detect that processes crashed ([13] and Section 2 of [7]).

Solving consensus in the presence of failures is way harder than solving consensus in the normal case. From the several techniques used to circumvent the FLP impossibility, failure detection and randomization are some of the most popular ones in this system model. Failure

detectors circumvent the FLP impossibility because they distinguish a slow process from a crashed one. When a process crashes and it is detected, all the other processes stop waiting for the crashed process messages and proceed with one less process. Random consensus algorithm [15] sacrifice determinism to circumvent the FLP impossibility [14] and solves the problem with probability one.

2.1.2 Byzantine Failure Model

For high reliable and secure systems, the crash failure model is not sufficient to model the failures that may occur in the system. To be reliable and secure a system should be able to function even in the presence of *arbitrary* failure. So, the model that models this type of systems should include arbitrary failures. Arbitrary failures are more commonly referred to as *Byzantine* failures after The Byzantine Generals Problem paper [16]. A process is said to be Byzantine if it fails to execute the algorithm in any possible way. Causes for Byzantine behaviour go from bugs in the software to an attacker that physically owns the machines of all the failed processes. In [16], using an analogy between generals and processes, it is proven that Byzantine consensus is only possible to solve if more than two-thirds of the processes are correct. So, Byzantine consensus with three processes does not tolerate any failure. With three processes, if any of them fails, the consensus algorithm should abort, and no value can be decided in order to assure safety, otherwise, the Byzantine process could influence the outcome of consensus.

The Byzantine Failure model guarantees very similar properties to the Crash-stop Failure model with two major differences. The first difference is that nothing can be expected from Byzantine processes. The consensus algorithm cannot rely on Byzantine processes even to know that they are faulty. Because of this, all properties must be ensured by the correct processes. The other difference is related to the validity property. Depending on the definition of validity used to define consensus, a problem may arise. If validity does not clearly state that a value is only valid if it was proposed by a correct process, and instead only states that a value is valid if it was proposed by any process, values proposed by Byzantine processes may be considered valid for consensus. The problem is that consensus is not supposed to decide a value coming out of nowhere. In order to maintain this property, we must formulate validity in another way. In this context two definitions of validity arise, *weak validity* and *strong validity* (Section 5 of [7]). Weak validity guarantees that consensus does not decide a value proposed by a Byzantine process only for executions where there are no Byzantine processes. If a process fails, weak validity states that a random value can be decided. Strong validity tolerates Byzantine faults and it guarantees that if all correct processes propose the same value then that is the decided

value. If correct processes do not propose the same value, a special value is decided to indicate that no valid value could be decided.

2.1.3 Speculative Consensus

We are particularly interested in *speculative* methods because our solution is based on blockchain and the most popular blockchain consensus algorithm PoW has a problem. The problem is that PoW admits inconsistencies for some undetermined time. One way for applications to deal with these inconsistencies is through speculative executions. We discuss PoW in more detail in Section 2.2.1. Another important fact is that, normally, consensus algorithms are very slow, and some effort has been done to make them faster. Speculation is one of the methods used to try to solve this issue.

In computer science, speculation happens when a process executes some operations based on a state that may not be the correct or final. It is normally used to improve performance when the state of the system used for the speculative execution has a high probability of being the correct one. For instance, we know that: a consensus algorithm that tolerates failures will, most of the time, have runs without failures and that the cost of tolerating faults exists in every run. Ideally, we would like to pay the cost of fault tolerance only in executions where failures occur. A speculative execution is an execution that attempts to save the cost of expensive and rare executions by optimistically choosing an algorithm that is less generic (i.e. it is correct in a stricter environment) but more efficient. When a failure is detected, the state of the system is rolled backed to the point before the beginning of the execution and a more expensive algorithm that tolerates failures is executed instead. Speculation is only useful if the cost of rollback is smaller than the cost of running the more expensive algorithm all the time. The use of speculation needs to be careful because it introduces some new and difficult challenges [12] [17] [18].

Zyzzyva is one of the first protocols that uses speculation to solve the SMR problem [12]. In Zyzzyva, replicas do not reply to clients after the execution of an expensive agreement algorithm, instead, they reply immediately after they receive the order proposed by the primary replica. This may cause the state of the replicas to diverge and different replicas to give different replies to different clients. If different clients receive different replies, they have a different view of the system and this breaks the abstraction that SMR gives. The challenge is to guarantee the abstraction even when the state of the replicas diverges. To do this, Zyzzyva leverages the information that the clients have of the system. For instance, the agreement sub protocol stops working on a request once a client knows the order of requests, avoiding work that would

otherwise be needed to achieve the order in replicas. In order to know this, clients need to know what action to take when they receive a message, i.e. they need to know if they can accept that message. To help clients decide, replicas append history information to the replies that they send to them. Clients then decide whether replies are based on the same ordering of requests.

In [18], the authors present *speculative linearizability* as new property that, as the name says, is related to *linearizability*. Linearizability is a property used in concurrent systems that allows to reason about the system by looking at parallel executions in a sequential (linear) way. Speculative linearizability has a similar purpose to linearizability but applied to speculative executions. Despite the importance of speculative linearizability, we are only interested in the framework developed as a proof of concept. To understand the framework and the key ideas behind it, we need first to reason about the notion of protocol phases. In Speculative linearizability, a protocol phase is an independent algorithm. The change between speculative phases implies the change of the algorithm that is in execution. The notion of protocol phases allows processes to switch from one phase to another without having to execute an agreement protocol. The framework works in a pluggable way, where implementations of protocols only need to be adapted (i.e. wrapped in an API) to be used. The idea is to use traditional fault-tolerant algorithms as speculative phases that serve as a backup when crashes occur and to use more efficient algorithms, that do not handle failures, for most of the executions. The power of the framework is demonstrated through two examples; we will look at one of them.

The example is an implementation of the Paxos algorithm [2]. To do this, Paxos was wrapped in an interface that allows for integration with other speculative phases, making Paxos a speculation phase on its own. Speculative linearizability hides the complexity of Paxos and allows the speculation phase to be composed with any other speculation phase to build a correct implementation of consensus. Paxos is composed of a Quorum speculation phase. This phase only needs two messages rounds to solve consensus, but it does not tolerate failures or contention. The system starts with Quorum running and then if contention of failures occur, Quorum cannot decide and passes control to Paxos. The other example given by the framework is a shared memory consensus [18].

Speculative Paxos [19] is another example of speculative consensus. It is closer to Zyzzyva than to the framework presented by Speculative linearizability in the sense that it is a modification of Paxos. This protocol is composed by three sub-protocols: speculative processing, synchronization and reconciliation. Speculative processing does something similar to Zyzzyva speculative phase where the replicas rely on a previously established order to execute the protocol and respond to clients before an agreement is confirmed.

Table 2.1: Examples of consensus API's

Library	Protocol	Propose	Decide
libpaxos	Paxos	paxos_submit	call back
BFT-Smart	MOD-Smart	invoke	return of invoke
etcd	Raft	propose	readIndex
Speculative Paxos	Speculative Paxos	invoke	return of invoke

In Speculative Paxos, the order is defined by a network primitive (Mostly-Ordered Multicast). Synchronization is responsible to periodically verify that replicas executed the same operations in the same order and reconciliation ensures progress when requests are delivered out of order. Unlike most agreement protocol, Speculative Paxos assumes a good behavior from the network. This is possible because the protocol was designed for the specific case where applications run inside a data center and the network is well-structured and managed. Like the other two examples, Speculative Paxos does not externalize speculative state to clients.

We have seen what a speculative execution is and studied examples of speculative consensus, we will now discuss the relationship between this and our work. Because of the fork problem, that we will discuss in detail in Section 2.2.1, consensus based on permissionless blockchains happens to be similar to speculative consensus protocols that we have seen. It is similar because the state of the system may need to be rolled back because the main chain in the blockchain is not the same as it was before. A key difference is that in speculative executions this need to roll back the state happens because of optimizations, i.e. because the system designer chose to design the system that way. While in blockchains it happens due to forks, which are intrinsic to blockchains based on PoW and PoS, in Zyzzyva and speculative linearizability speculative executions are part of the definition of the protocol. Zyzzyva and speculative linearizability never externalize speculative state to non-speculative components. In contrast, PoW blockchains externalize speculative state and forces applications to handle it.

2.1.4 Consensus API

In abstract, consensus can be specified by two events, propose and decide. Processes have an initial value v, and to propose it they invoke $\langle Propose,v \rangle$. The consensus algorithm executes. When a value d is decided, processes receive the value through $\langle Decide,d \rangle$ (Section 5 of [7]). We call *classical consensus interfaces* to consensus interfaces with $\langle Propose,v \rangle$ and $\langle Decide,d \rangle$. We will now discuss several examples of API presented by implementations of consensus algorithms.

Table 2.1 presents examples of implementations of consensus protocols and the methods corresponding to a consensus API. The table contains four examples with the respective names,

implemented protocol, propose and decide primitives.

Libpaxos [20] is an open source implementation of Paxos. Despite not having the exact same names, its methods names are similar to the ones used in a classical consensus API. The method *paxos_submit* is the corresponding method to the propose primitive and that processes are informed of the decided value through a callback event.

BFT-Smart [21] implements MOD-Smart [22], which is a BFT SMR protocol. The SMR problem uses consensus [5]. Because of this, BFT-Smart also provides an interface similar to a classical consensus interface. In fact, it is very similar to the libpaxos interface. The difference is in the decide primitive. In libpaxos the client receives the decided value through a call back while in BFT-Smart it is received as the return of invoke method.

Etcd [23] implements Raft in order to provide a distributed reliable key-value store. What is interesting about etcd is that the Raft library is independent of the rest of the application. In etcd, the propose primitive has the same name as the one from a classical consensus API. The method equivalent to the decide primitive is called readIndex. Different from the other examples, the clients need to invoke readIndex explicitly.

Speculative Paxos [24] implements protocol with the same name [19]. This example, despite being a speculative protocol has a similar interface to the other three examples given. Invoke is used to propose values, and the decided value is given by the return of invoke.

With these examples is possible to see that implementations of consensus do not follow a classical consensus interface but they try to keep it simple and as similar as possible.

2.2 Blockchain

This section introduces blockchain and its organization is the following. Section 2.2.1 presents permissionless blockchains with particular attention to PoW and PoS. Section 2.2.2 discusses permissioned blockchains. The final Section 2.2.3 presents a discussion about the various types of blockchain consensus algorithms presented in the previous sections.

Bitcoin [1] was developed with the objective of eliminating the need for a trusted third party in online transactions and blockchain is the key technology behind it. To eliminate the need for a trusted third party, payees (those that receive money) must be able to verify the legitimacy of transactions and be sure that the payers (those that pay) do not double spent the coin.

The double spend problem can be explained as follows; entity A pays YA amount of coins to entity B for the asset XB; then A uses the same coins YA to pay entity C for the asset XC; because C is not aware that A already used the YA coins to pay B, it accepts the transaction. In the end A has gained XB and XC and only spent YA coins. The problem is that the payee

can not verify that the payer actually owns the coins. To prevent the double spend problem the payee needs to be aware of all transactions. This is where the blockchain is important. Blockchain works as a distributed ledger that registers transactions, and the payees use it to verify transitions. The blockchain gives proof that a majority of nodes agree on a single global history about the order in which the transactions were received [1]. Note that, for a majority of nodes to agree on a single global history, they have to reach a consensus. Blockchain is secure as long as correct nodes own a majority of CPU power in the network, blockchain security issues will be discussed later because they depend on the type of the blockchain.

A blockchain is composed by a chain of blocks. Each block is composed by a timestamp, a nonce, a hash of the previous block and a list of all the new transactions that were not stored in the previous blocks [6]. The timestamp proves, that the transactions existed at that time and the order in which they were received. Miners are nodes in the network that have the function to mine ("create") new blocks, producing a PoW. PoW is the process of finding a nonce that, when hashed together with the content of the block produces a hash starting with a specific number of zeros. This hash proves that the history of transactions has not been tampered with. Section 2.2.1 discusses PoW in detail. The following steps explain what happens in Bitcoin's blockchain network when a transaction takes place [1]:

1. New transactions are broadcast to all nodes.
2. Each node receives new transactions and stores them into a block.
3. Each node tries to find PoW for its block.
4. When PoW is found, the block is broadcasted to all nodes.
5. The block is accepted by nodes if all transactions in it are valid and not already spent.
6. To express their acceptance of the block, nodes work on creating the next block in the chain, using the hash of the accepted block as the previous hash.

These steps allow for two or more miners to find PoW for two different blocks creating two separated chains, *a fork in the chain*. The fork will eventually disappear when another block gets mined and only one of the chains accepts the new block. This is because nodes always chose the longest chain, the alternative chain will get discarded. We will return to the fork problem when we discuss some blockchain issues.

Ethereum [6] is another system that uses blockchain. It was inspired by Bitcoin. The authors of Ethereum claim that Bitcoin ledger can technically be thought of as a state transition system.

The ownership status of the bitcoins is the "state" of the system and exists a "state transition function" that receives a state and a transaction and outputs a new state. So, the idea of Ethereum is to create a generic state transaction system where users can do much more than transact coins, namely users can use scripts to define smart contracts. Ethereum defines *smart contracts* as systems which automatically move digital assets according to arbitrary pre-specified rules. A simple example of a smart contract can be the payment of a rent, the tenant defines that every month X coins are transferred to the homeowner, on the other side the homeowner can have a smart contract that "replies" to the payment of the rent with the equivalent of a receipt.

From the application point of view, the major difference between Ethereum and Bitcoin are the smart contracts. From the blockchain perspective, the major difference is that the blocks from Ethereum, in addition to saving all the content that Bitcoin saves, also store the most recent state. Here, state is represented by "accounts", each account contains four elements; a nonce, to prevent the repetition of transactions; ether balance with the current amount of ether in the account; the accounts contract code and storage.

Due to Bitcoin popularity and its proven usability as a real system, the idea of developing new applications based on cryptocurrency has become popular. Before Ethereum presented smart contracts, there were three possible approaches to building application on top of a cryptocurrency: build a new blockchain, use scripting on top of Bitcoin, and build a meta-protocol on top of Bitcoin. Building a new blockchain gives total freedom to the developers but comes with the cost of development. Scripting is easy to implement, as it does not have the costs of development, but it is limited in terms of what can be done. Meta-protocols suffer from scalability issues because it consists in building an application on top of an already complex system. Bitcoin supports a weak version of "smart-contracts" were the coins can be owned not just by users represented by their public key, but also by a script expressed in a programming language. The scripting functionality of Bitcoin is not Turing-complete, lacks state, is value-blind and blockchain-blind. Value-blind means that there is no way for a script to provide fine-grained control over the amount that can be withdrawn. Blockchain-blind means that scripts are blind to blockchain data such as the nonce and previous hash [6].

The major goal of Ethereum is: "to merge together and improve upon the concepts of scripting, altcoins and on-chain meta-protocols, and allow developers to create arbitrary consensus-based applications that have the scalability, standardization, feature-completeness, ease of development and interoperability offered by these different paradigms all at the same time." [6]. Ethereum allows developers to write their own distributed based application where they can

define rules of ownership, transactions and state transaction functions. This is possible because Ethereum implements a blockchain with a built-in Turing-complete programming language [6].

There are three types of blockchains: *public*, *consortium* and *private* [25]. On a public blockchain, any node can participate both in transactions and the consensus instances, while on consortium blockchains only a few selected nodes can participate in the consensus instances. Consortium blockchains usually involve business partnerships and they can be seen as partly decentralized. Private blockchains are controlled by their respective owners and they decide who participates in the system. A different classification is also presented by Hyperledger [26]: *permissionless* blockchains like Bitcoin's blockchain where everyone can participate, and *permissioned* blockchains where participants are controlled, have identities and form a consortium. We can see that public and permissionless is the same category and that the consortium and private blockchains belong to the permissioned category.

There are numerous forms of consensus protocols that apply the same concepts of PoW like: PoS, Proof of Existence, Delegated Proof of Stake, Proof of Activity (hybrid of proof of work and proof of stake), Proof of Importance, Proof of Storage [25], [27], etc. In Section 2.2.1 we discuss PoW and PoS respective applications, issues and relation with our work.

2.2.1 Permissionless blockchains

This section discusses permissionless blockchains. This type of blockchain is normally based on proof-of-"something" like PoW and PoS. We focus only on PoW and PoS because those two are the most used and studied types of proof.

Proof of Work

In Bitcoin's blockchain, PoW helps to solve consensus. If the majority voting was done through one-vote-per-node, it could be easily forged. In PoW the voting is one-vote-per-CPU, therefore it can not be easily forged because in practice, to simulate CPU power an attacker needs to actually have that CPU power or at least pay for it and the electricity it consumes. In abstract, PoW is very simple, one entity proves to another entity or entities that it performed some computation. In Bitcoin, this proof is done through the calculation of a hash and the process of calculation this hash is called mining [1]. This hash is the hash of the content of the block to be added to the chain, concatenated with a nonce. The hash cannot be a random hash: the value of the hash needs to be inferior to a specific well-known value. To obtain a hash that satisfies this condition, miners increment the nonce of the block, hash the block together with the nonce, using for SHA256, and check if the new hash is acceptable. When a miner finds an acceptable

hash, it broadcasts the block to the other nodes. The other nodes check if the hash is valid, and if it is, they stop mining and accept the block from the miner. PoW helps to solve consensus because it is unlikely that more than one version of the block is accepted into the chain. We can make an analogy between PoW and known consensus algorithms like Paxos. The period of time when miners are looking for PoW for a specific block corresponds to a normal consensus instance, and the miner that actually finds the hash is the *leader* of that instance. Since the leader is a single node, it receives transactions and orders them in the order it prefers, e.g., in the order in which they were received.

Different miners can receive different transactions from different orders but here is where the leader is important to solve consensus. The order of the leader is the one that is included in the new block, all the different blocks from different miners are discarded. If the block of the leader does not include a transaction, it is as if it never existed and their results will not be visible in the blockchain. It is now clear that all the nodes agree on the same order of transactions because they all accept the block from the leader. The difficulty of mining can be adjusted simply by changing the target value, i.e. by changing the number of zeros of the target hash. For instance, in Bitcoin, the difficulty is defined in a way that a new block is added to the chain every ten minutes [1].

In practice, many problems may occur during the calculation of the PoW as well as problems related to the PoW concept itself. The most relevant problems are forks and the 51% attack.

A normal fork is, in simple words, when two different chains exist. Normal forks occur when two different miners do the PoW for two different blocks at the same time. A set of processes accepts the block from one miner while the others accept the block from a different miner. An undefined number of normal forks may occur, and multiple concurrent chains may exist at the same time. Processes chose the longest chain as the correct one because it is the one with more work done and eventually only the longest chain will remain. A more serious fork problem happens when the software of the blockchain is updated.

This new fork problem is divided in two types, *soft forks* and *hard forks*. The soft fork occurs when an update to the blockchain makes the rules to accept block *more strict*. Some blocks that were considered valid will not be accepted by the new rules. If more than 50% of the mining power shifts to the new version, the system self-corrects. Nodes following old rules will accept blocks that are not accepted by the new rules but if the majority of the mining power shifted to the new version, this chain will eventually be discarded. Hard forks are the opposite, the rules to accept blocks are now *less strict* or incompatible with the old ones. This fork poses a more serious problem because nodes that follow old rules will never accept blocks

that violate the old rules. This fork is permanent unless the majority of mining power stays with the old nodes, which makes the update useless. Soft forks are backward compatible while hard forks are non-backward compatible. Forks are a serious problem in the blockchain when they happen the processes that interact with the blockchain observe different states. From a consensus perspective, it is as if the *Decide* primitive returned different values for different processes, and if the fork disappears a set of processes received a wrong decide value. Consensus is not reached.

Because forks are the main challenge in our solution, we need to understand the probability of that event. In [28] and [29] both empirical and theoretical estimates are made. In [28] the theoretical probability of a fork is 1.741% and the empirical value is 0.3%. In [29] the theoretical probability of a fork is 1.78% and the empirical value is 1.69% that was optimized to 0.78% with an improvement proposed in the paper. Those measurements and calculation where made with specific models for Bitcoin PoW blockchain and under the normal case (i.e., as far as the authors known, there was not an attack in the system).

The 51% attack can have more serious consequences than forks. Since in PoW we have one-vote-per-CPU, if a node or a set of nodes owns more than 50% of the total CPU power from all the miners, that node or that set of nodes basically owns the blockchain. A node that owns more than 50% of CPU power can change the entire blockchain [25].

In Bitcoin, nodes receive an incentive to stay correct by receiving a coin for every block that they mine, but when a node owns more than 50% of the CPU power this incentive becomes useless. In [30] it is proved that the original Bitcoin PoW is not incentive compatible and a minority of nodes could join in a pool (still with a minority of CPU-power) and thought *selfish mining* attack the blockchain. The 51% attack has been proved to be possible with only 25% of CPU power [30]. Another problem with PoW is the waste of resources. All the miners spent resources mining the next block but only one of then gets the reward. These problems with PoW blockchains motived the community to find alternatives, with PoS and BFT algorithms being the most promising.

Proof of Stake

PPcoin [31] introduced PoS as an alternative to PoW. The main motivation behind PoS is the waste of electricity in PoW. With PoS the network security (and cryptocurrency security) in the long term does not depend on energy consumption. To understand the concept of PoS, we need first to understand the concept of *coin age*. In short, coin age is the amount of currency that some entity has multiplied by the number of days that the entity possesses each coin, for

example, a person that owns 50 coins for 10 days, possess 500 coin age.

PPcoin has a hybrid design. It has two types of blocks, PoW blocks and PoS blocks. In the PoS blocks the proof is a special transaction in the block called *coinstake*. Coinstake is a transaction where the owner pays himself thereby consuming its coin age. By spending its coin age, the owner gets the right to mine a PoS block. The Coinstake transaction has three inputs: two stakes and an input called kernel. The kernel is a hash that like PoW needs to meet a certain target.

The difference between the kernel and the PoW hash is that the search space is limited while in PoW it is not limited. Because of the limited search space, the energy consumption in PoS is minimal when compared to PoW. The PoW target that the kernel must meet is a target per unit coin age consumed in the kernel; the more coin age consumed in the kernel the easier is to meet the hash target.

A blockchain with PoS solves consensus the same way that a blockchain with PoW. In PoW, the longest chain keeps a history of transactions that is the same for all nodes. In PoS the "longest" chain is the chain with the most coin age amount, every transaction in the block contributes to the total coin age of the block and every block in the chain contributes to the total amount of the chain [31].

PoS does not take any special action regarding the fork problem; it has the same issue as PoW but it has better defences against the 51% attack. It is argued that in PoS the cost of owning more than 50% of the total amount of CPU power is higher than in PoW because normally coins have more value than electricity and the system is not sustainable otherwise. To perform such attack, the attacker would need to own more than 50% of the coins. The main issue with PoS is that by trying to solve PoW problems it introduced new ones.

A problem with using coin age is that it lowers the cost of attacking the history of the entire chain. The double spending attack was made easier because now an attacker needs only to accumulate a certain amount of coins to be able to reorganize a block. This is also known as the *nothing at stake* problem [32]. There is also a problem with attackers reusing the same stake to produce multiple blocks. Because the attacker reuses the same stake, a denial-of-service attack is easy and cheap to produce. The duplicate-stake protocol prevents this problem, in this protocol each node needs to collect (kernel, timestamp) of all coinstake transactions. Nodes use this information to detect duplicated pairs and ignore the related block [31].

2.2.2 Permissioned blockchains

This section presents permissioned blockchain. Permissioned blockchain was created because certain types of applications (e.g., banking application) can not have anonymous user and information can not be public. In this type of blockchain, nodes need to be authenticated by the consortium of members. With authentication, information is only available for selected nodes and every member of the system is well known. Due to the fact that PoW and PoS blockchains store the entire history of transactions and do not require authentication from users, they can not be used to build this type of blockchain.

Hyperledger Fabric [26] is a distributed ledger platform to run smart contracts. The idea is to have a modular architecture that allows well-known implementations of algorithms to be easily pluggable in a blockchain system. Hyperledger Fabric's main characteristics are: a permissioned blockchain; smart contracts, called chaincode; a pluggable consensus component which in the current release is an implementation of PBFT protocol [4]; support for security with Certification Authorities (CA) for multiple types of certificates; a key-store interface for persistent state and other less important characteristics. For our work there are two types of nodes, validating nodes and non-validating nodes. Validating nodes are the ones that maintain the ledger, validate the transactions and run consensus instances while non-validating nodes function as a proxy for clients and do not perform transactions. We are more interested in Hyperledger's Fabric approach to consensus and how the validating nodes execute the PBTF protocol (i.e. how PBFT works) which we will discuss next.

PBFT protocol has two types of nodes, replicas and clients. Since PBFT is a SMR protocol, replicas have the responsibility to store the system state while clients can invoke arbitrary functions to change the state stored, it is important to note that these operations need to be deterministic. Because PBFT is a Byzantine fault-tolerant protocol it can not have any assumptions about how processes fail, but as already mentioned in Section 2.1 to solve consensus some assumption about the system need to be made (because of the FLP impossibility). PBFT has three assumptions about the system:

- *Bound on faults:* assume a bound $f = (n - 1)/3$ on the number of faulty replicas.
- *Strong cryptography:* an attacker can not break any of the cryptographic functions used in the system
- *Weak synchrony:* the delay of time, between a process sending a message and another process receiving the same message, has an "asymptotic upper bound".

The first two assumptions exist to assure both safety and liveness while the third one exists only to assure liveness.

All replicas start with the same state. The hard part is making sure that all non-faulty replicas execute the operations in the same order. To achieve that, PBFT uses a mix of primary-backup [33] and quorum replication [34] techniques to order requests [4]. In primary-backup replicas move through views. A view is a like a well-defined time interval where there are two clear and easy to distinguish sets of processes, the correct ones and the faulty ones. In a view, there is a leader whose responsibility is to order the operations requested by clients and inform the other replicas of that order. The leader may be faulty and for instance, send different orders to different processes. The backups detect that the leader is faulty by using timeouts and by verifying the sequence numbers assigned by it. If a process suspects the leader, it can start the process to elect a new one (view change). To elect a new leader, at least $f + 1$ process need to suspect the leader, otherwise faulty processes could sabotage any leader. When the view changes a new leader is elected. Quorums are used to disseminate information and order operations even in the presence of faults and they have two relevant properties [4]:

- *Intersection:* any two quorums have at least one correct replica in common.
- *Availability:* there is always a quorum available with no faulty replicas.

When a replica writes to a quorum it collects a quorum certificate. A certificate is a set of messages from quorum members saying that they store the information, the certificate assures the replica that the information was stored. The certificate mechanism is used to validate every step of the protocol.

When a client wants to perform operations in the system, it broadcasts the request to all the replicas. The replicas receive the request and reply directly to the client. The client waits for $f + 1$ replies, if it does not receive $f + 1$ replies from replicas, it resends the request.

Tendermint [32] is a blockchain engine that separates the consensus algorithm from the rest of the blockchain. Tendermint started as traditional cryptocurrencies like Bitcoin and Ethereum, but it evolved to be a general purpose blockchain consensus engine that can host arbitrary application states. It is similar to Hyperledger Fabric [32]. The idea is for Tendermint to be used as a plug-and-play replacement for the consensus engines of other blockchain systems. It implements a version of DLS protocol. DLS protocol is a BFT protocol [35] [32].

Blocks are determined in rounds. Each round is composed of three steps (propose, prevote, and precommit), along with two special steps Commit and NewHeight. The consensus algorithm is executed by *validating* nodes (like in the other systems presented, the validating nodes are the ones responsible for validating transactions therefore they are responsible for consensus).

Validators participate in consensus by voting on blocks. In every round, a validator is selected to be the *proposer*. The proposer is responsible to broadcast an unconfirmed block in the round. Proposers are elected in a round robin fashion and the frequency of each node is proportional to the nodes voting power. The steps work in the following way [36], [32]:

1. *Prevote*: validators receive the proposed block from the proposer and decide if they can broadcast a prevote for that block.

2. *Precommit*: if a node receives more than 2/3 of prevotes on the proposed block, it broadcasts a precommit for that block. If a node receives more then 2/3 of precommits, it enters the commit step.

3. *Commit*: another round of messages is initiated, validators collect commit messages. If the node receives 2/3 of commits, it accepts the block and broadcasts a commit for that block. At the end of commit, validators broadcast the new block and prepare for the next one (the preparation for the next block is called NewHeight).

Tendermint and Hyperledger Fabric are just two examples of permissioned blockchains. We have seen how they allow the use of blockchain without using any type of proof. They use traditional algorithms to solve the consensus problem in a blockchain. Therefore, the process of mining is not needed and problems like forks do not exist. Because they use a traditional BFT protocol, they tolerate up to one-third of Byzantine processes in the system. Another issue with BFT protocols is that they have poor scalability in the number of nodes that execute the protocol [37].

2.2.3 Consensus comparison

This section presents a discussion about blockchain consensus protocols that we have discussed. We will discuss the advantages and disadvantages of each protocol.

Table 2.2 presents a summary of the main consensus methods used in relevant blockchains. Table 2.2 is inspired and adapted from Table I of [36]. First, we note that the two examples of permissioned blockchain use BFT protocols that tolerate 1/3 of Byzantine processes while in permissionless blockchain it is hard to discriminate the number of faulty processes because a single process can have the power to, by himself, attack the entire chain. The main advantage of permissionless is that they are completely decentralized and with the chain stored in a large number of participants, it is almost impossible to tamper with transactions. On the opposite side, permissioned blockchains have fewer participants and the chain is stored in fewer nodes but because the members of the system are more controlled, the number of Byzantine processes

Table 2.2: Consensus comparison

Property	**Bitcoin[1]/ Ethereum[6] (POW)**	**PPcoin[31] (POS)**	**Hyperledger Fabric[26] (BFT)**	**Tendermint [32] (BFT)**
Node identity management	permissionless	permissionless	permissioned	permissioned
Tolerated power of adversary	$<$ 25% of adversary computing power	$<$ 51% stake	$<$ 33.3% faulty replicas	$<$ 33.3% Byzantine voting power
Main advantage	decentralized	decentralized	well known and studied	control over nodes that execute consensus
Main disadvantaged	forks / selfish mining	forks /nothing at stake problem	node scalability	node scalability
Transactions Rate/s	7 / 15	unknown	3 500	10 000

in the system should be smaller. Although fully decentralized, both PoW and PoS tend to centralization. PoW tends to centralization because as the chain keeps growing the number of miners with the ability to mine will decrease. It happens because the resources needed for mining will increase with the number of blocks. In PoS centralization can be achieved through a rich miner with more than 50% of the coinage in the system.

Regarding scalability, PoW blockchains have good node scalability with poor performance, whereas BFT blockchains have good performance with a small number of nodes but have a limited scalability. PoS blockchains have better performance than PoW blockchains but it is still far from BFT performance [37].

Regarding performance, we observe that there is a clear difference between the permissioned and permissionless blockchains. The transaction rate is higher in the permissioned blockchain. This difference complies with our theoretical expectations because on permissioned blockchain based in BFT there is no need to find any type of "proof-of-something" that normally involves heavy cryptographic calculations, there is only a need to order and validate transactions and because all the nodes in the network are known this happens to be relatively fast and simple.

The values for transactions rate in the table were obtained from the following sources. In Bitcoin, Hyperledger and Tendermint the values are from the original papers and thesis respectively [1], [26] and [32]. Unfortunately, we could not find a reference for the PPcoin transaction rate.

We can conclude, as expected, that every protocol has its own advantages and disadvantages. There is not a "one-size-fits-all" in blockchain consensus.

Chapter 3

Blockchain Consensus Service

In this chapter, we present Blockchain Consensus Service (BCS), a blockchain consensus service based on blockchains. In this chapter, we discuss all the components that are part of BCS, from the system model, API, algorithms used etc... to the applications that can use it. The chapter is structured as follows. Section 3.1 presents the model of the system. Section 3.3 presents the API of BCS. Section 3.4 presents the algorithm used in BCS. Section 3.5 shows how BCS deals with forks and Section 3.6 presents a discussion about BCS and smart contracts.

We have seen in Section 2.1.4 that existing implementations of consensus try to implement a classical consensus API. We have also seen in Section 2.2 that there are numerous blockchain algorithms that solve consensus. We propose a solution to this problem by creating a generic consensus service that provides a classical consensus API on top of the blockchain. This section explains our solution. We target permissionless blockchain system because, as we have seen in Chapter 2, they provide properties that permissioned blockchains do not provide, and pose a more difficulty solution due to the possibility of forks. We discuss this in more detail in the following sections.

3.1 System Model

The system is composed by; Applications that are the clients of our system, Wrappers, Blockchain Clients and the blockchain itself. Each Application has a Library and is connected to one Wrapper. Every Wrapper is connected to one client Application and to one Blockchain Client. The Blockchain Client is connected to one Wrapper and to peers of the blockchain network. Next, we present the assumptions about our system:

- Communications: we assume that interprocess communication and remote communications are done using authenticated perfect links. Authenticated perfect links are defined

in Section 2.4.6 of [7] and provide three properties.

  - Reliable delivery: If a correct process sends a message m to a correct process q, then q eventually delivers m.
  - No duplication: No message is delivered by a correct process more than once.
  - Authenticity: If some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p.

- Membership: BCS is publicly accessible. Still, to participate, Applications must have a blockchain account and get registered in a *Certification Authority* (CA). All clients with a blockchain account have a key pair (a public key and a private key). When clients perform an operation by sending a transaction to the blockchain, the transaction must be signed using the private key. To validate that a particular transaction was created by a member of the system (not just someone with a blockchain account), clients have access to a certificate signed by a trusted CA, which has a list of public keys of all the members in the system. Whenever a Wrapper receives a transaction, it uses the certificate to validate the client's public key and verify that it is indeed a member of the system.

- Fault Tolerance: if a set composed by Application, Library, Wrapper and Blockchain Client has a crash or any other type of fault (Byzantine); it is assumed that there are always enough correct processes such that we can obtain a Byzantine quorum. That is, the number of faults that the system tolerates is limited. This eliminates the needs for a fault detection mechanism. In a system with N Applications (and respective Wrapper and Blockchain Client) our system tolerates $F = \lfloor (N-1)/3 \rfloor$ faults. We assume that the set composed by Application, Library, Wrapper and Blockchain Client can be seen as a single process in the system and thus fail simultaneously and in the same way.

- Consensus Instance: the system assumes that clients know the identifiers of the consensus instances that they want to participate. The identifiers are used to distinguish between different consensus instances.

- Checkpoint: the system assumes that there is a checkpointing mechanism to its own state. Checkpointing is a mechanism that periodically saves the state of the blockchain relevant to our system. Thus, when a new client enters the system, his Wrapper can query this checkpoint mechanism to know the state of the system at the time the checkpoint was made. After obtaining a state through the checkpoint, the Wrapper completes its state by reading the remaining blocks until it reaches the same point of the remaining Wrappers.

Checkpointing is important for two reasons. First, it prevents the potentially time-consuming reading of the Blockchain for each of the new system clients. Second, it allows Wrappers not to save information about the system from the beginning of time.

A fairly simple method to avoid the need for a checkpoint is to force Wrappers to read the entire blockchain from the beginning of time (beginning of time for our system, and not the entire blockchain) and save their state. Reading the blockchain can take days and the amount of information related to the system can be quite a lot.

- Forks: the system assumes a low probability of forks (check Section 2.2.1). The system also assumes that hard forks do not happen and that the soft forks have a very high probability of being resolved in the N blocks that are closer to the head of the chain (more about forks in Section 3.5).

3.1.1 Target Applications

Based on the system model presented in Section 3.1 and the restrictions caused by the way BCS handles forks (see Section 3.5 for an explanation about the way we handle forks and the restrictions that it causes) not all applications can use BCS.

A fork can be seen as speculative execution that needs to be aborted because it caused inconsistencies among replicas. Based on this, applications that use our system need to be able to deal with these potential inconsistencies. One of the inspirations for out idea on how to handle forks was a system called Correctables presented in [38].

Correctables [38] makes *incremental consistency guarantees* available to the applications. Choosing the appropriate consistency model is a major problem for application designers. They have to choose between *performance* and *correctness*. To have a good performance they choose to have *weak consistency* allowing inconsistent states to happen (as long as they are not very common). When correctness is a requirement of the system, *strong consistency* is used at the expense of performance. Correctables aim to create a system that helps applications to switch between weak and strong consistency, it also allows developers to choose the consistency level for a specific operation. By offering the same abstraction, we target the same class of applications identified in [38], such as a ticket selling system and an ad serving system.

However, our *strong* consistency model is not final as the one presented in [38] due to the non-zero (but very small) probability that we can observe a fork longer than N blocks. Therefore, applications must be able to deal with this low probability event. Examples of applications that, despite this problem, could use our system are stock management applications. When the stock is very high, the application can simply use the weak consistent results. When the stock is

low the application may choose to wait for the stronger consistent results. Even if the stronger consistent results turn out to be inconsistent and for example, the application sells more units of a product then the ones that are available, this is something that already happens in real-world applications. For example, it is public knowledge that flight companies sell more tickets than the ones that are available to maximize profits in case a passenger does not show up to the flight. When there are more passengers than seats, they compensate the passenger that does get to go on the plane.

We will present in Chapter 5 the application we used to evaluate our system.

3.2 Architecture

To implement a consensus service that provides a classical consensus API on top of existing blockchains; we propose the architecture defined in Figure 3.1.

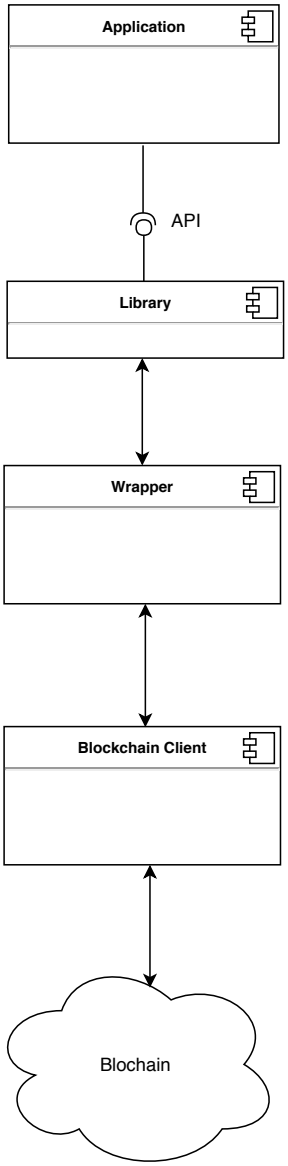

Figure 3.1: Overview of the proposed architecture

The architecture is composed by the following components:

- *Application:* It is the client of the BCS service. It uses the API provided by the Library to interact with the system.

- *API:* Bridges the gap between the application and the consensus algorithm. This API has, among others, the $\langle Propose,v \rangle$ and $\langle Decide,d \rangle$ methods or similar. We will discuss the API in Section 3.3.

- *Library:* The Library provides the API and manages communication between the Application and the Wrapper. It is responsible for the translation of operations from the Application to the Blockchain. For example, translating a propose invocation into a blockchain

transaction with the necessary information. It is also responsible for informing the client when decisions are available.

- *Wrapper:* The Wrapper has the function of execution the main algorithm (see Section 3.4). It is responsible for forwarding the transactions sent by the Library with the information of operations to the Blockchain. The Wrapper reads the blockchain and sends the results from the execution of the algorithm to the Library.

- *Blockchain Client:* The Blockchain Client is a peer of the Blockchain network and is responsible for committing the transactions to the Blockchain, to be connected to other blockchain peers and to receive the chain of blocks that compose the Blockchain.

- *Blockchain:* The Blockchain is responsible for solving consensus and store the history of all transactions. It consists in a set of nodes that, typically, are similar to the Blockchain Client (Section 2.2).

We will now discuss the proposed architecture, together with its strengths and weaknesses. The justification for the Application, the Blockchain Client and the Blockchain itself are very simple. We need an Application because it is the client of our system and we need it to invoke operation, otherwise the system would do nothing. We use a blockchain because the main idea behind our work is to leverage its properties (see Section 1.1). We use a Blockchain Client because it is the simplest way to interact with the Blockchain network.

The reason for the Library component is the following: we decided to add this extra layer to the Application because it allows us to keep the information necessary to create transactions with the owner of this information (the Application) and at the same time not force the Application to create the transactions themselves. This is important for two reasons. First, the information needed to create transactions is very sensible (it contains the private key of the blockchain account) and because of this, it can not leave the client. Second, if the applications are forced to create transactions, our system would not be compatible with already existing applications. Having the Library component, allows us to offer a simple consensus API. It is much simpler to use an already developed library than to modify the Applications in order for them to create the blockchain transactions. Still, developers who want to manage transactions themselves can ignore the Library and communicate directly with the Wrapper.

The Wrapper is the middleware responsible for transforming the consensus abstraction that the Blockchain provides for transactions into a generic consensus service.

The strongest point of this architecture is that the separation between the Library and the Wrapper. By doing this separation we allow two things. The first is that it is possible to run

the Library and the Wrapper in different machines, which means that the Application does not need to have the blockchain in its machine. The second is: in a system that is not subject to Byzantine faults, the Wrapper can serve multiple Libraries (see Section 3.1).

The weakest points about this architecture are: the Wrapper needs to run in the same machine as the Blockchain Client which means that it needs to run in the machine that has the entire blockchain (depending on the Blockchain system that is used, it can consume up to 1TB of disc space, and when the initial synchronization is occurring it also consumes a decent amount of memory). The Wrapper and the Blockchain Client need to run on the same machine because for security reasons the implementations of Blockchain clients that exist do not support secure remote connections. This happens because using a remote connection would increase the attack surface making the system more vulnerable. They do not support remote connection to force all the accesses to be made locally.

The fact that the Wrapper needs to run in the same machine as the Blockchain Client causes another issue. If we want to tolerate Byzantine faults, the Application, the Library, the Wrapper and the Blockchain Client need to be the same entity. This means that they will all fail in the same way and at the same time. This assumption simplifies our design. In a Byzantine system, components can not trust each other and because of this, if we separated the Library from the Wrapper, the libraries would need to talk to a Byzantine quorum of Wrapper because it could not trust in a single one. This would increase the complexity of the system.

We saw in Section 3.1 that our system tolerates $F = \lfloor (N-1)/3 \rfloor$ faults. When we refer a fault, the components that are included are: the Application, the Library, the Wrapper and the Blockchain Client.

3.3 API

We have seen in Section 2.1.4 the API of several different consensus libraries. We will now present BCS API.

1. $\langle join, gId \rangle$: allows clients (the applications) to join a group in the system.

2. $\langle leave, gId \rangle$: allows clients (the applications) to leave a group the system.

3. $\langle propose, id, gId, v \rangle$: allows clients (the applications) to propose a value to a consensus instance. It is a propose that receives the value to be proposed, the id of the consensus instance and the group that the instance and the application belong to.

4. $\langle decide, d \rangle$: callback that informs the application about a decision. It returns the respective

decided vector d with the accepted values. The vector contains the id of the groups, the id of the instance, the values and meta information about the values. It contains a list of the members of that instance, the number of the blockchain block that contains the last accepted value (it is the point where the decision was made) and for each value, it contains the id of the client that proposed it.

5. $\langle decideFinal,d \rangle$: callback that informs the application about a final decision. It was made by the second reading of the wrapper and provides an extra level of consistency.

Besides these methods, the Library needs to receive information about the client account in the blockchain system to create the transactions. In our prototype, this information is given through a configuration file.

3.4 Algorithm

This section introduces the core algorithm of our system. The algorithm is executed by clients and wrappers. Although Wrappers do not communicate with each other, their decisions must be the same because they all access to the same blockchain and the blockchain ensures the order of the transactions in the system (except in the exceptional case of forks).

3.4.1 Execution at a Participant

Figure 3.2 is a sequence diagram that shows in a very simple way the sequence of steps that occur at a participant of the system. Briefly, the steps are:

1. The Application invokes $\langle Join,gId \rangle$ to join the groups with gId in the system (Step 1 in the Figure 3.2).

2. The Library creates a transaction with the information that its application wants to join the system and sends this transaction to the Wrapper (Step 2 in the Figure 3.2).

3. The Wrapper forwards the transaction to the Blockchain Client (Step 3 in the Figure 3.2).

4. The Blockchain Client commits the transaction to the Blockchain (Step 4 in the Figure 3.2).

5. The Blockchain Client replies to the Wrapper with the transaction hash (Step 5 in the Figure 3.2). The Wrapper forwards the hash to the Library (Step 6 in the Figure 3.2) and from the Library (Step 7 in the Figure 3.2) the hash goes to the client (Step 8 in the

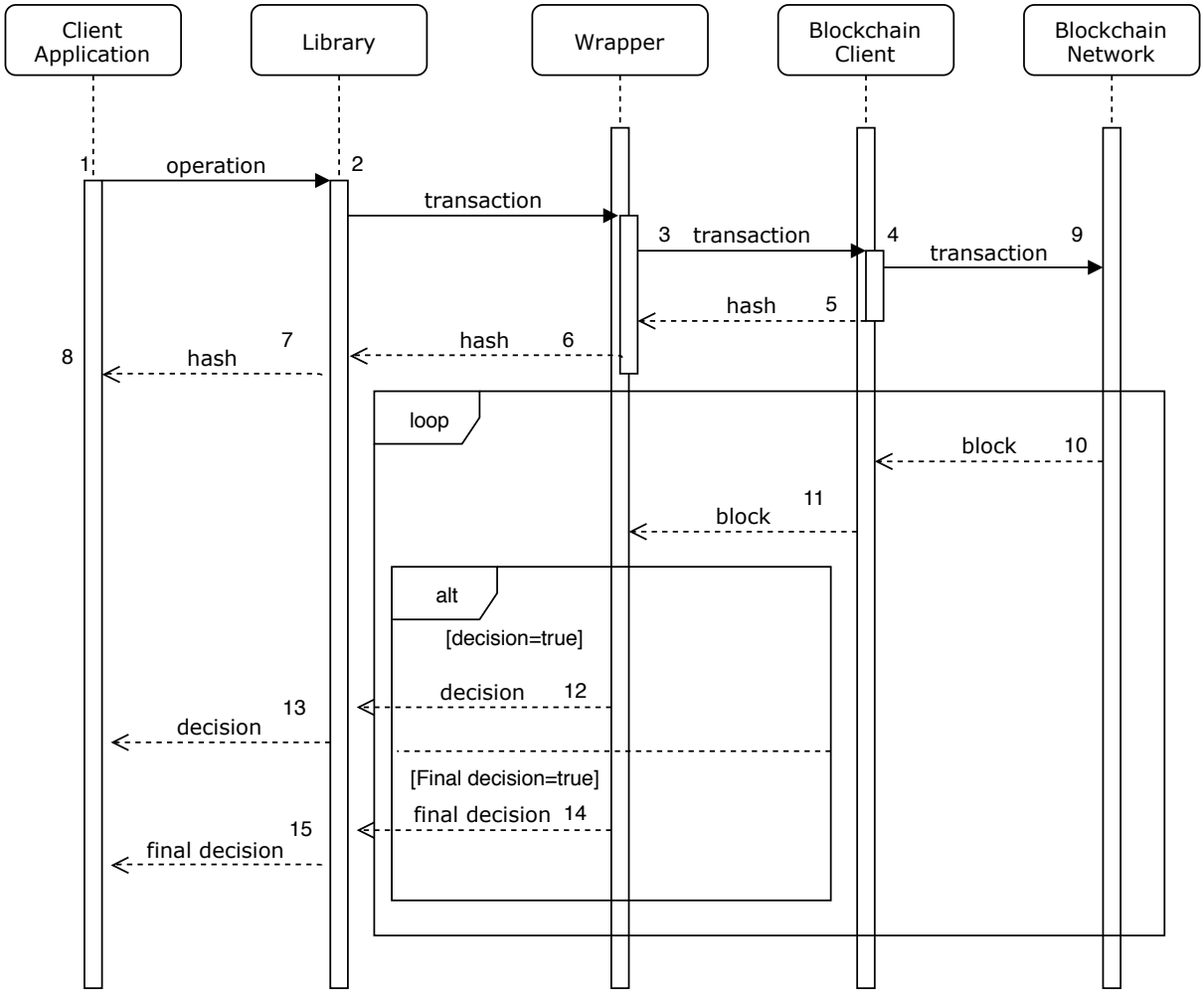

Figure 3.2: Simplified sequence diagram of the execution at a participant

Figure 3.2). The hash works as proof that the transaction is a valid (that does not mean that the transaction was inserted into a block)

6. Now the application can propose the value. It associates the proposal to a consensus instance (i.e. giving it a consensus id) and calls $\langle Propose, id, gId, v \rangle$ (Step 1 in the Figure 3.2).

7. The Library executes step 2, but for the proposal. Meaning that it creates the transaction containing the information of the proposal. Step 3, 4 and 5 are executed again by the Wrapper and the Blockchain Client respectively.

8. Transactions are included in blocks of the blockchain (Step 9 in the Figure 3.2).

9. The Blockchain Client receives the blocks from other nodes in the network (Step 10 in the Figure 3.2). The wrapper receives the blocks from the Blockchain Client and begins to interpret them (Step 11 in the Figure 3.2).

10. The Wrapper analyses the transactions with the join operations form the multiple applications in the system. With this information, the Wrapper can know how many members each instance of consensus has and who they are. When the Wrapper finds a transaction with a proposal, it starts the collection of proposals for the consensus instance with the id X that the proposal contained. From this point forward is impossible for new members to join e leave instance with id X. The join and leave operation after this point will only be valid for instance newer than instance X.

    The Wrapper keeps reading the blockchain until the number of proposals P for instance with id X meets the following condition: $P >= N - ((N-1)/3)$ where N is the number of members of the instance X. The Wrapper creates a vector ordered by the same order of the transactions with the proposals and sends this vector to the Library that he is connected to (Step 12 in the Figure 3.2).

11. The Library receives the decisions from the Wrapper and uses a callback to inform the Application of the decisions (Step 13 in the Figure 3.2).

12. The Application can now propose a value for the next consensus instance or invoke leave to no longer participate in the system. The invocation of leave works in the exact same way as the join.

13. In order to provide the final decisions the Wrapper reads the blockchain at two different positions. The first reading is done as fast as possible at the head of the blockchain (i.e. as

soon as blocks are available). The second reading is a constant number of blocks behind the first one and follows the same rules. We need two different reads to handle forks. Section 3.5 explains in detail how BCS deals with forks. When the final decision for instance X is available, the Wrapper informs the Library (Step 14 in the Figure 3.2). The library uses a call back to inform the Application (Step 14 in the Figure 3.2).

The idea behind the concept of having groups in the system is to allow different types of applications to be running concurrently. This allows different applications with their replicas to execute consensus on our system in a concurrent way.

An important detail is the beginning of the reading by the Wrapper (i.e. the block in which they begin the reading). The size of Bitcoins blockchain is more than 170GB and the size Ethereums blockchain is more than 1TB. Because of the size of the Blockchain and the fact that transactions that the system commits to the blockchain are included at the head of the blockchain, all the blocks that were added before the beginning of our system are irrelevant to us. For those two reasons, the reading should start at the block that was the head of the blockchain when our system first started. In practice the reading can start in any block, it's just a matter of configuring the system and having all the Wrappers start reading at the same block.

3.4.2 Algorithm Execution Example

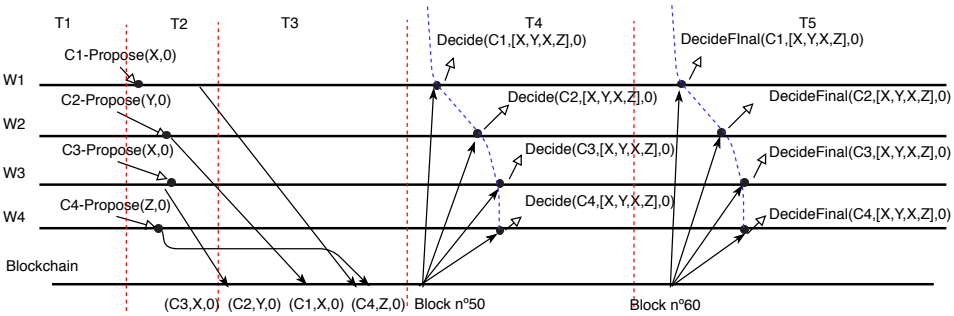

Figure 3.3: Protocol Diagram

Figure 3.3 shows an example of an execution of the algorithm in a simple and correct execution. The diagram considers 4 Wrappers (W1, W2, W3 and W4), each connected to a Blockchain Client (not represented in the diagram for simplicity) and 4 connected Applications (C1, C2, C3 and C4), each connected to his own Wrapper. This diagram assumes that the 4 Applications have already successfully joined and are part of the system and that there is only one group in the system. Because all groups are completely independent, including multiple groups in the

diagram would only make it more complicated without really adding useful information.

Now we will explain the diagram.

- In T1, applications propose their values by creating a transaction that includes the value of the proposal and the instance to which that proposal belongs to (0 in this example).
- In T2, Wrappers receive the transactions and sends them to the Blockchain Client.
- In T3 Wrappers wait for the transactions to be processed by the Blockchain network.
- In T4 Subsequently, the Wrappers receive the transactions with the Applications' proposals. The proposal is validated because all Applications have already done JOIN. Wrappers interpret client's proposals and, if there are enough proposals, they decide the set with the proposed values and inform the clients.
- In T5, after a defined number of Blocks (10 in this example) the Wrapper reads the blocks that contained the proposals and sends the final decision to the application.

Applications receive the decisions in a form of an ordered vector. After receiving the vector, is the job of the application to perform a deterministic operator over the values in the vector (e.g. choose the maximum value or the minimum value).

The Applications that joined the system belong to all the instances of the group that they join until they explicitly invoke the LEAVE operation. This will create a transaction that will be included in the blockchain and read by the Wrappers.

3.4.3 Specification

We will now present the algorithm in a more rigorous way. We will start by the Library and how it translates the commands provided in the API to blockchain transactions and send them to the Wrapper. Then we show how the Wrapper forwards the transactions to the Blockchain Client and finally we present the core algorithm of the system that is executed by the Wrapper.

Library

Algorithm 1 presents the API provided for clients to communicate with the wrapper. This is implemented in the library that runs on the client application to communicate with the wrapper. The methods of the Library are the following.

- Send: sends messages from the clients to the wrappers. This communication can be using TCP/IP sockets.

- RawTransaction: receives as input all required fields for a transaction and creates the transaction. The relevant fields for consensus resolution are; the group identifier (gId), the client identifier (cId), the transaction type (JOIN, LEAVE or PROPOSE), the proposed value (value) and the consensus instance identifier (instanceId). These fields are included in the data field of the transactions. The nonce is used to distinguish transactions (it is a field needed to create transactions). The gas price (gasPrice) and gas limit (gasLimit) are fields needed to create a transaction. They indicate how much the creator of the transaction is willing to pay for the transaction. This method performs the following steps to create a transaction; identify the next nonce to the account of the owner of the transaction, create and encode the transaction. The method then calls another method that signed the transaction. The signMessage method receives a string to be signed and the necessary cryptographic keys to sign it. It can use any cryptographic algorithm (like RSA) to sign the message as long as it is compatible with the Blockchain system used.

Funtion `Join(`*gId*`)`**:**

data = $\langle gId, JOIN \rangle$

transaction =`RawTransaction(`*cId,adressTo,gasPrice,gasLimit,data*`)`

transaction = signMessage(transaction, wallet)

hash= `Send(`*transaction*`)` to wrapperIp

return hash

Funtion `Leave(`*gId*`)`**:**

data = $\langle gId, LEAVE \rangle$

transaction =`RawTransaction(`*cId,adressTo,gasPrice,gasLimit,data*`)`

transaction = signMessage(transaction, wallet); hash= `Send(`*transaction*`)` to wrapperIp

return hash

Funtion `Propose(` *gId,instanceId,value*`)`**:**

data = $\langle gId, instanceId, value, PROPOSE \rangle$

transaction =`RawTransaction(`*cId,adressTo,gasPrice,gasLimit,data*`)`

transaction = signMessage(transaction, wallet); hash= `Send(`*transaction*`)` to wrapperIp

return hash

upon *receive decision*:

trigger *decision*

Algorithm 1: Library pseudo-code that translates commands into transactions

Wrapper

Algorithm 2 presents the methods used by the Wrapper to forward transactions. It assumes that the Library and the Wrapper know each other and that exists a communications service

between them.

The Wrapper contains a service that is always waiting for requests. The Library sends a message to this service with the transaction. The Wrapper reads the message with the transaction, forwards the transaction to the Blockchain Client, waits for the reply with the transaction hash and replies to the Library with a message containing the transaction hash in case the transaction was valid or $\perp$ if the transaction was not accepted by the Blockchain Client. The methods of the Wrapper are the following.

- Send: forwards the transactions sent by the Library to the Blockchain Client and waits for the reply with the transaction hash. The Wrapper receives the hash and sends it to the Library using the clientIp. This communication can be done TCP/IP sockets.
- upon receive transaction: The Wrapper waits for a transaction from the Library. When this event occurs, the wrapper uses a mechanism to send the transactions to a Blockchain Client running on the same host.

```
Funtion Send( transaction):
    socket.write(transaction)
upon receive transaction:
    hash = Send(transaction) to Blockchain
    Send(hash) to clientIp
```

Algorithm 2: Wrapper pseudo-code that forwards transactions

Consensus

The reading of the blockchain is presented in Algorithm 3. To help understand the algorithm we will first present its state variables. Because of the way we deal with forks, by simply reading the blockchain twice (more details on Section 3.5), all the global variables of the Wrappers were duplicated, having for each one of them a current version and a final version. The variables that hold the considered final state, which can not be changed even if a fork occurs, are the " final " variables that are identified by starting with f_, the remaining name of each variable is equal to the name of the variables of the current version. The list is the following:

- block: it is the block of the blockchain that is being parsed at the moment
- currentBlock: is the number of the current block being parsed.
- t_i: is the current transaction being processed.
- members: is an array that holds all of the members who did JOIN to the each of the groups.

- f_members: is equal to the members but it refers to the second (final) reading of the blockchain.
- instances: is a two-dimensional array that holds the instances of each group. Each instance has a set of members, a set of proposals made by the members, and a set of members that have already proposed their value called proposers.
- f_instances: is equal to the instances but it refers to the second reading of the blockchain.
- decisions: is the set of all the decisions that have already been made.
- f_decisions: is equal to the decisions but it refers to the second reading of the blockchain.

We have seen the list with the variables, but we also need to understand the content in the transactions. Every transaction contains a set of with the following fields:

- $t_i.type$:- is the data present in the transaction that allows the wrapper to know if the transaction corresponds to a JOIN, LEAVE, or PROPOSE.
- $t_i.client$:- is the identification of the client that created the transaction and invoked the corresponding operation. In our prototype, it is the address of the blockchain wallet owned by the client.
- $t_i.instanceId$:- is the identifier of the consensus instance that the PROPOSE corresponds to.
- $t_i.gId$:- is the identification of the group to which the operation corresponds.
- $t_i.value$:- is the proposed value.

Next, present an overview of the main blocks of the algorithm:

- upon start: when a wrapper starts executing and joins the system, it uses the checkpoint mechanism to update its state. After updating his state, the wrapper starts reading the blockchain until it gets to the most recent block. Two readings of the blockchain are made, one to have the decisions as fast as possible and a final one to confirm the decisions made.
- ReadCheckPoint: The Wrapper uses the checkpoint mechanism to populate the system state variable. To know: who the members of each instance are, which instances of consensus are incomplete (the ones that do not have enough proposal to reach a decision).

- upon block appended to the blockchain: if a new block B is added to the blockchain it is necessary to process it. And since we have already received a new block, we can confirm that the B−MAX_FORK_SIZE block information is final. Two blocks are processed, the N block (the current one) and the B−MAX _FORK_SIZE block.

- processBlock: receives a block from the blockchain and iterates through all transactions. For each transaction in the block, it checks whether the transaction belongs to our system or not. If it belongs, its type is checked (JOIN / LEAVE / PROPOSE) and for each type, the required action is taken. In the case of JOIN / LEAVE members are, respectively, added/removed to the system. In the case of PROPOSE, a function is invoked that verifies that the proposal is valid and takes the necessary decisions. At the end of each block, all decisions taken in the block are sent to the client.

- isConsensusTransaction: receives a transaction and accesses the data field of the transaction to verify its type and check if the transaction was committed to the blockchain by a client of our system.

- ReadPropose: receives a transaction referring to a proposal. It makes the necessary checks to know if PROPOSE is valid and takes decisions according to the state of the consensus instance to which the PROPOSE belongs. If a decision has already been taken for the consensus instance to which the proposal refers, the proposal can still be added to the decisions if the proposal belongs to the same block of the decision. The block of a decision is the block of the last proposal needed to make the decision.

- SendDecisions: The wrapper informs its client (if he is a member of that consensus instance) of the decision.

```
upon start :
    ReadCheckpoint()
    blockCounter= Blockchain.getBlockHeight()
    for i = currentBlock + 1;  i < blockCounter;  i + + do
        block = Blockchain.getBlock(i)
        ProcessBlock(block,members,decisions,iStarted,pJoins,pLeaves,proposals,false) f_block
          = Blockchain.getBlock(block.height-MAX_FORK_SIZE)
        ProcessBlock(f_block,f_members,f_decisions,f_iStarted,f_pJoins,f_pLeaves,f_proposals,true)
    end
Funtion ReadCheckpoint():
    members=checkpoint.getMembers()
    instances=checkpoint.getInstances()
    decisions =checkpoint.getDecisions()
    f_members=checkpoint.getMembers()
    f_instances=checkpoint.getProposals()
    f_decisions =checkpoint.getDecisions()
    currentBlock=checkpoint.getBlockHeight()
upon block being appended to the blockchain :
    ProcessBlock(block,members,decisions,iStarted,pJoins,pLeaves,proposals,false)
    f_block = Blockchain.getBlock(block.height-MAX_FORK_SIZE)
    ProcessBlock(f_block,f_members,f_decisions,f_iStarted,f_pJoins,f_pLeaves,f_proposals,true)
Funtion ProcessBlock(block,members,decisions,instances):
    foreach t_i ∈ block.transactions do
        if isConsensusTransaction(t_i) then
            switch t_i.type do
                case JOIN do
                    members[t_i.gId]= members[t_i.gId] ∪ {t_i.client}
                end
                case LEAVE do
                    members[t_i.gId]= members[t_i.gId] \ {t_i.client}
                end
                case PROPOSE do
                    ReadPropose(t_i,instances, decisions,members)
                end
            end
    end
    SendDecisions(decisions)
Funtion isConsensusTransaction(t_i):
    data=t_i.data
    if ¬(data == NULL) then
        type = data.type
        if ¬(type == NULL) then
            return type==JOIN || type==LEAVE || type==PROPOSE
    return false
```

```
Funtion ReadPropose(t_i,decisions,members,instances):
    if t_i.client ∈ members[t_i.gId] then
        if decisions[t_i.gId][t_i.instanceId] == ∅ then
            //If there is no instance with the given instanceId we need to
            //create a new one
            if instances[t_i.gId][t_i.instanceId] == ∅ then
                instances[t_i.gId][t_i.instanceId].members=members[t_i.gId]
            instance= instances[t_i.gId][t_i.instanceId]
            if t_i.client ∈ instance.members && t_i.client ∉ instance.proposers then
                instance.proposers=instance.proposers ∪ t_i.client
                instance.proposals=instance.proposals ∪ t_i.proposal
                //check if the number of proposals is enough to have a decision
                if (#instance.proposals >=
                 ⌊(#instance.members − ((#instance.members − 1)/3))⌋ then
                    decision.vector=instance.proposals
                    decision.block=t_i.block
                    decision.members=instance.members
                    decision.proposers=instance.proposers
                    decisions[t_i.gId][t_i.instanceId]=decision
        //if there is a decision but the proposal belongs to the same block
        //as the decision and the client is a member of the instance,
        //then his value can be accepted
        else if decisions[t_i.gId][t_i.instanceId].block == t_i.block &&
         t_i.client ∈ decisions[t_i.gId][t_i.instanceId].members &&
         t_i.client ∉ decisions[t_i.gId][t_i.instanceId].proposers then
            decisions[t_i.gIdId][t_i.instanceId].vector=decisions[t_i.gId][t_i.instanceId].vector ∪
             t_i.proposal
            decisions[t_i.gId][t_i.instanceId].proposers=decisions[t_i.gId][t_i.instanceId].proposers ∪
             t_i.client
Funtion sendDecision(decisions):
    foreach d_i ∈ decisions do
        if ¬d_i.sent then
            send(d_i)
            d_i.sent = true
    end
```

Algorithm 3: Consensus Algorithm

3.5 Forks

The main challenge in our approach was how to deal with forks. Forks pose a problem to the resolution of consensus because when they occur, they cause different Wrappers to read different blocks that can have different orders in the transactions.

There are two main mechanisms for The first

The most used mechanism for dealing with forks in the blockchain requires that: to confirm a transaction as permanent, it is necessary to wait for a certain number of blocks, say N blocks, to be added to the blockchain [39]. That is, a transaction t included in the block with number X is confirmed when the block with number $X + N + 1$ is included in the blockchain. In the Bitcoin blockchain, the value typically used for N is 6 [40], in Ethereum the recommended value is 12 [40].

Therefore, in concrete, we deal with forks by having the Wrapper reading the blockchain twice. The first reading always reads the head of the blockchain while the second reading is a fixed number of blocks behind. This works because forks have a low probability of occurrence and they happen at the head of the blockchain (remember the probability of forks from Section 2.2.1). The fact that the second reading is not done at the head of the blockchain makes it less probable to be affected by forks (i.e. it will be less affected by those forks cause by two miners producing different blocks at the same time). The second reading is not immune to forks, but they already have a low probability at the head of the chain and the more the second reading is behind, the lower the probability of it to be affected by a fork.

It is also important to clarify the difference between the state of the blockchain and the state of our system. The state of the blockchain is composed by; transactions, clients, the balance of clients' accounts, the order of transactions, among others. While the state of our system is composed by; the consensus instances, the members of each instance and the system's clients (the Applications). The state of the system depends on the state of the blockchain because of the order of the transactions. Thus, an inconsistency in the state of the blockchain (forks), leads to an inconsistency of the state of our system.

Note that the checkpoint that we talked about in this section and the checkpoint that we explain in Section 3.1 are different. The checkpoint in Section 3.1 refers only to our system and it is a checkpoint on the state of the system (which applications, which members of each application, which instances of incomplete consensuses among others ...). The checkpoint referred to in this section to solve forks is done on the blockchain as a whole and would continue to compel the Wrapper to read the whole blockchain to obtain the state of the system. These two types of checkpoint are compatible and have the potential to be used together.

3.6 Discussion

The goal of this dissertation was to take advantage of the properties offered by Blockchain systems and at the same time provide the properties of Byzantine consensus. Both [36] and [25] list the properties provided by a Blockchain system. We have seen how the system works, now we will see the properties that the system offers. Because or system uses Blockchain it also provides same of the Blockchain properties. The properties that our system provide that derivative directly from Blockchain are:

- *Decentralization:* Blockchains have their own distributed consensus algorithm that maintains consistency in the network eliminating the need for a central entity that ordered the transactions. Because of this, and because we did not add any centralization point to the system, our system is also decentralized.

- *Persistency:* It is nearly impossible to delete or rollback transactions in the blockchain. To do it an attacker would need to own some serious computational resources (see Section 2.2.1). Because the operations in our system are done through transactions, our system also has persistency.

- *Auditability:* It is possible to audit a blockchain because transactions can be easily tracked and verified. Again, because the operations in our system are done through transactions, our system also has auditability. In fact, in the absence of a fork, it fairly simple for anyone that knows our algorithm to read the blockchain and check how every decision was made.

Note that our system, as well as Blockchains, is auditable, distributed and has persistency all thanks to the distributed tamper-proof ledger that is the core of the Blockchain.

We have seen in Section 2.1 that consensus is defined by three properties, agreement, validity and termination. Therefore, in order for our system to offer a consensus service, we need to provide those properties. In a permissioned Blockchain there is no problem in assuring them but in a permissionless blockchain, forks pose a problem. We have seen in Section 3.5 that the theoretical probability of forks is below 2%. Our solution in a permissionless Blockchain can provide those properties with more than 98% probability. Also, with the second reading, the probability of a fork is even small and gets close to zero. We can think that the 2% probability of something to go wrong in the first reading, and even lower for the second reading, is a fair price to pay for the extra properties that the Blockchain provides and that traditional consensus systems do not provide.

Because the system provides a vector consensus service we define agreement, validity and termination in the same way as in Section 2.2.1. Therefore, if we have N processes (Application,

Library, Wrapper and Blockchain Client we have already stated in Section 3.1 are a single process) and tolerate f failures, we have the following:

- Vector validity: Every correct process that decides, decides on a vector V of size n:
  - $\forall p_i$: if p_i is correct, then either $V[i]$ is the value proposed by p_i or $\perp$;
  - at least $(f + 1)$ elements of V were proposed by correct processes.
- *Agreement:* No two correct processes decide differently.
- *Termination:* Every correct process eventually decides

Our system provides vector validity because every correct Wrapper in the system collects more than $((N + f)/2)$ proposals. The missing values to complete the total number of N, are the $\perp$ values. In practice, we only collect values from the blockchain.

In the absence of a fork, every Wrapper reads the same blocks from the blockchain, and all correct Wrappers will interpret the blockchain in the same way, so they will all decide the same ensuring agreement.

In our system, in the absence of a fork, all the operations from a correct process will reach the Blockchain and eventually every correct Wrapper will read them. The only case where our system may not decide is in the presence of fork, namely when the Wrapper reads the wrong blocks from the wrong chain and those blocks do not contain the transactions with the proposals (or the transactions are not in the same order). Then the fork is solved and the transactions with the proposal are included in blocks that the Wrapper had already read (the first time). The second reading will fix this problem and all the Wrappers will have the same final decisions.

Note that, while improbable, forks that are longer than the number of blocks that the second reading is behind the first can happen. If this happens, Wrappers will read the wrong blocks from the wrong chain and those blocks may not contain the transactions with the proposals (or the transactions may not be in the same order). Then the fork is solved and the transactions with the proposal are included in blocks that the Wrapper had already read (the first and the second readings). The Wrapper will never read those proposals, the system will stop, and no decision is ever made for that consensus instance.

Now that we presented BCS, we will explain why we did not use smart contracts to implement a blockchain based consensus service (see Section 2.2). The main reasons for us to not use smart contracts in our solution are the following.

- The purpose of smart contracts is to be standalone entities, meaning that after they are deployed, they should be independent of other entities unless the contract pre-defined rules

depend on them.

- The use of smart contracts forces us to use a blockchain that supports smart contracts, and this would make our solution less generic.
- The difference in the portability of the application. a) Smart contracts use a specific language with semantics not always well defined and b) there are many applications that need consensus and could benefit from being in the blockchain. With the smart contracts, they would have to be rewritten, with our approach this is almost transparent allowing existing applications to be easily migrated to the Blockchain.

This chapter presented BCS. It presented BCSs: system model, target applications (Section 3.1), architecture (Section 3.2), API (Section 3.3), main algorithms (Section 3.4), fork handling mechanism (Section 3.5) and discussion (Section 3.6).

Chapter 4

Implementation

In this chapter, we describe the implementation details of the proof of concept. Our implementation used the Ethereum blockchain [6], `Java` as a programming language and the `Web3j` [41] library to query the Blockchain Client.

Ethereum is one of the most popular Blockchains used for development. The reasons for this are: it was the first one to introduce smart contracts, it is at the time of the creation of this document the second most valuable cryptocurrency (after Bitcoin) and more importantly it was created to be, not just a cryptocurrency, but a Blockchain framework that allows the development of any arbitrary application [6]. Because of that, it is reasonably documented which was an important criteria for us.

The use of Ethereum allowed us to use public test networks to evaluate our system in a realistic environment. The test network that we use was `Rinkeby` [42]. The Rinkeby test network uses an alternative to PoW called *Proof-of-Authority* (PoA). In PoA only certain nodes can mine transactions in a semi-trusted environment that is not as energy consuming as the public environment [27]. In this network, blocks are generated at an average rate of one block every 15 seconds. Because blocks are mined in a semi-trusted environment instead of the public environment of the Ethereum, the block generation rate does not have large variations with a standard deviation of half a second [42].

We also tested BCS in a private network and in another Ethereum test network named Ropsten [43]. We did not present results for these two tests because we had some problems and that we believe that they were not realistic. The problem with the private test network was that we needed to have nodes that mine running in our test machine. This is an issue because the latency between all the miners was near zero and the difficulty of mining need to be very low (with is not realistic) for us to be able to have several miners ruining in the same machine. The problem with the Ropsten test network is its size (around 25GB). The larger size of Ropsten

did not allow us to test the system with a reasonable number of clients.

We choose Rinkeby for two reasons. The first is the size of the chain, from all the test networks in Ethereum, it is the one that has the smallest size. The size is important because it was the main constraint in our experiments (we will discuss this in detail in the next paragraphs). At the time of the writing of this dissertation, the Rinkeby test network had more than 3 million blocks that occupied around 17GB of disk space. The second reason is that we were able to obtain a significant amount of Rinkeby ether with allows us to test the system without having to be worried about ether to pay for the transactions.

Since we choose Ethereum, we had to choose an Ethereum client. For that we choose `geth` because it is one of the official implementations of an Ethereum client developed by the Ethereum foundation, the entity responsible for developing the Ethereum blockchain system [44]. Another reason for choosing geth is that we found a library called `web3j` [41] that offers a complete implementation of Ethereums JSON-RPC client API defined in the Ethereum yellow paper [45] over *Hypertext Transfer Protocol* (HTTP) and *Interprocess Communication* (IPC). This allows applications to do all types of operations on the Blockchain, from the creation of transactions to the reading of blocks, reading of transactions and the creation of personalized filter, to filter certain blocks and transactions and others.

We implemented the Library, the Wrapper and a demonstration application. In the Library, we use the method RawTransaction.createTransaction from web3j to create the transactions and TransactionEncoder.signMessage to sign the transactions. The metadata from the operations was converted into a JavaScript Object Notation (JSON) string. The bytes from this string were included in the data field of Ethereum transactions.

The communication between the Library and the Wrapper was done through bidirectional SSL sockets (using the javax library). The Wrapper communicates with geth using IPC sockets (in particular web3j UnixIpcService) and transactions were sent using the method `ethSendRawTransaction(transaction).send()`

The reading of the blockchain done by the Wrapper was implemented by creating two subscriptions using `catchUpToLatestAndSubscribeToNewBlocksObservable` event subscription from web3j. The first subscription corresponds to the first reading of the blockchain and the second subscription to the second one. Each subscription runs on an independent thread and to ensure that the second read is N blocks behind the First we forced the tread of the second subscription to sleep and only executes when it is indeed N blocks behind.

We saw in the Section 3.5 that the number of blocks that are needed to confirm a transaction

as permanent in Ethereum is 12 blocks. So, in our implementation, the value of N is 12.

The implementation consists of 400 lines of Java code for the test application, 500 lines of Java code for the Library and 900 lines of Java code for the Wrapper.

This chapter presented the implementation details of the proof of concept. It describes the tools used, from programming languages to the blockchain system used.

Chapter 5

Evaluation

This chapter presents the evaluation of our solution. Recall that, the objective is to provide a consensus service based on a blockchain system. With this in mind, our evaluation was designed to answer the following questions:

1. Is it possible to have a consensus system based on a permissionless blockchain system?
2. In what way do forks impact the system? What is the frequency of incorrect first decisions and incorrect final decisions?
3. What is the cost of waiting for a final decision? How long is the wait between the initial decision and the final decision? How often does the initial and final decision differ?

To help us have a better understanding of our system and to provide some clues about the answers to these questions, we thought of some metrics. Figure 5.1 helps us understand some of this metrics by showing a simple sequence diagram of the system that we will use to explain what are some of the metrics. The metrics are the following:

- Transactions request latency: the amount of time it takes for a transaction created by a client operation to reach the blockchain and for the transactions hash to be returned to the client has a form of confirmation. It is the time that passed between 1 and 2 of the Figure 5.1.
- Initial decision latency: the amount of time elapsed on a client since it proposed a value for a consensus instance until he receives the initial decision from that instance. It is the time that passed between step 1 and step 3 of the Figure 5.1.
- Final decision latency: the amount of time that passed on a client since it proposed a value for a consensus instance until he receives the final decision from that instance. It is the time that passed between step 1 and step 4 of Figure 5.1.

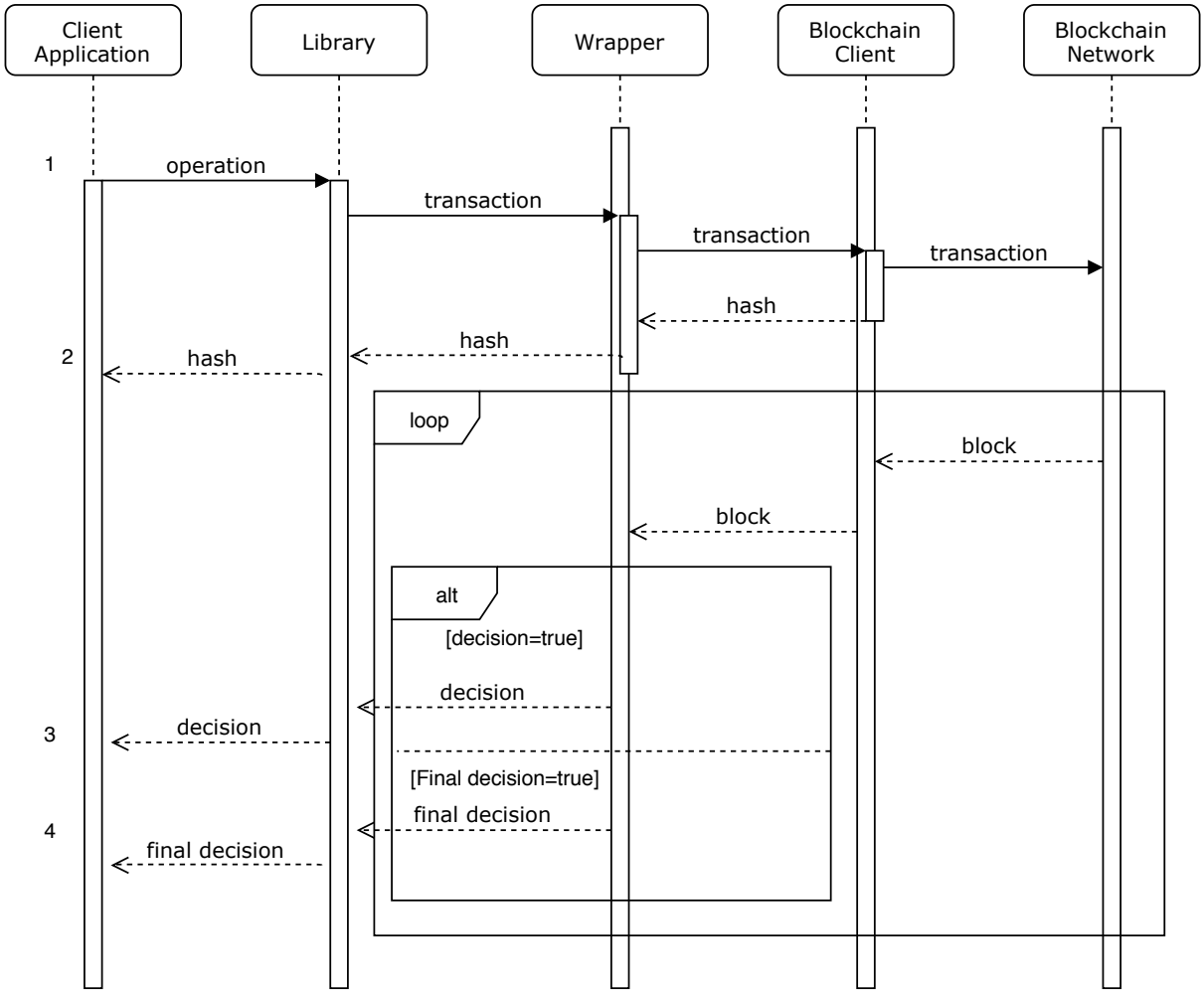

Figure 5.1: Simplified sequence diagram of the solution

- Number of blocks per decision: the number of blocks that were created since the block that contains the first value in the decisions until the block of the last value included in the decision.

- Wrong decisions: the number of times that a single client received a final decision different from the initial one. This can also be seen as the number of times that a single client was affected by a fork.

- Wrong final decisions: the number of times that for the same instance, at least one client received a different decision from the other thus violating agreement. This can also be seen as the number of times that a single client was affected by a fork longer than N, the defined number for the second reading, blocks.

- Instances with a wrong decision: the number of instances where at least one client receive a different initial decision.

- Number of clients that received a wrong decision per instance: the number of clients that received a wrong decision in an instance with a wrong decision. We will use this metric to understand how many clients were affected by the fork that caused the wrong decision.

The idea with the evaluation is to have a notion about the impact of dealing with forks and the general performance of the system. Normally this type of evaluation is done to compare different solutions, but we know that our solution is not comparable with, for instance, a Paxos, Raft or SMR implementation. Those systems are capable of executing the equivalents of our consensus instance under a second ([3], [21]), while our system is limited by the blockchain block generation rate (in the order of tens of seconds). In fact, a consensus service based on blockchain will never have a better performance than a "pure" consensus service for the simple reason that the blockchain itself uses consensus and add an extra layer of complexity. However, the consensus of permissionless blockchains can scale to a much larger number of participants. Figures 5.3 and 5.2 make it very simple to understand why our system will never perform better than other consensus algorithms. Figure 5.2 represents a simple consensus algorithm that provides a consensus service while figure 5.3 represents a blockchain based consensus service with the blockchain and consensus services on top of blockchain as extra layers.

We tested our system with a permissionless blockchain and not with a permissioned one because generally permissioned blockchains provide stronger properties and because of that, the fork problem does not exist. Because there are no forks in permissioned blockchains, they are less changeling (see Sections 2.2.2 and 2.2.1).

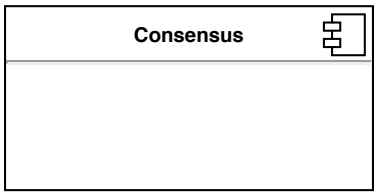

Figure 5.2: Consensus

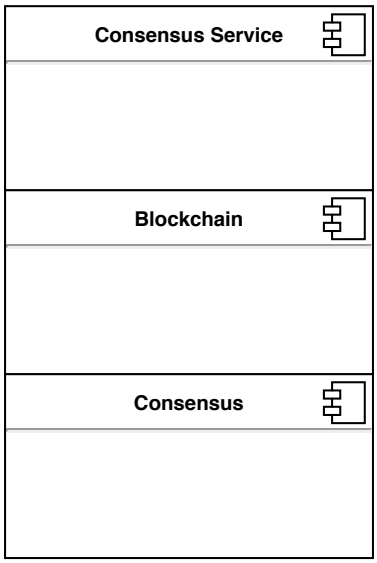

Figure 5.3: Blockchain consensus

5.1 Test Application: Ticket Selling System

As our use case application, we implemented a simple ticket selling system inspired in Listing 5 of [38]. The application sells tickets to events (e.g. concerts, movies etc...) and is available to vendors. Vendors can create events with a certain amount of tickets and can sell tickets to those events.

In the application, every client knows the existence of one event and how many tickets that event has. The tickets in the application are all equal The Application proposes to the system the number of tickets that it wants to sell. When the decision is delivered, the application checks if the number of tickets that remains is above zero. If the remaining amount of tickets is above zero then all the replicas of the application, that successfully proposed a value, can sell the tickets that they requested. If there are not enough ticket to fulfil all the requests in the vector, the application sells the tickets by order. This way a replica of the application can know if it can or not sell the tickets. The application has a threshold on the number of tickets and when that threshold is reached the application waits for the finals decisions before deciding about selling the tickets. For a stock of tickets above the threshold, applications can safely sell the tickets without waiting for a final decision thus improving latency.

5.2 Test Description

We tested our system in the Ethereum test network, Rinkeby [42] (see Chapter 4). In this network, blocks are generated at an average rate of one block every 15 seconds. Because blocks are mined in a semi-trusted environment instead of the public environment of the Ethereum main network, the block generation rate does not have large variations with a standard deviation of half a second [42].

We executed two experiments, one with 7 clients and another with 14 clients. We choose to start with 7 because we thought that for our minimum number of clients we would like to tolerate more than 1 fault. With seven clients, the system is able to handle 2 faults. Also, seven was the maximum amount clients that we were able to run in the machine that we used for development.

The experiments were run in a Supermicro Superworkstation 7039a-i with 128 Gb of memory ram (four units of DDR4-2666, Registered ECC with 32 Gb). The disc is an Intel DC S4500 Enterprise series SSD, 480GB, 2.5" TLC SATA-6Gb. It has two 4U active CPU heat sink and two CPUs Intel Xeon Silver 4114, LGQ 3647, 2.2GHz, 10C/20T.

In this machine, we had only 220GB of disk available. With this amount of disk available, the maximum number of clients that we were able to execute was 14. This happens because each client needs to have its own copy of the blockchain.

In Ethereum, every transaction has a transaction fee (so does Rinkeby). The creators of the transaction define the maximum amount of ether that they are willing to pay [6]. Due to transaction fees, the creators of the blockchain blocks give priority to transactions with the highest transaction fees. Because of this, in our system, and due to the fact that we had access to a large amount of Rinkeby ether, we decided to pay a for transactions a value substantially higher than the average value paid in the network. According to [42] the gas price is at 1 Gwei (1 Gwei corresponds to 0.000000001 ether) and the gas limit of 0.00009 Gwei. For the transactions in our system, we defined the gas limit to 0.0001 and gas price at 1.0001 Gwei. Another detail is that Rinkeby is an Ethereum test network so the number of transactions in it is very low when compared to the main Ethereum network. Most of the blocks in Rinkeby do not reach their limit. These two facts make it very likely that the transactions in our system were included in the block to be created. Other transactions were not prioritized over ours.

In the two experiments, we executed 24 runs of 100 instances of consensus for a total of 2400 instances of consensus. Every time that we started a run, clients had a new Ethereum account with the initial balance of 0,05 ether. Starting with new accounts for every execution allow us to test the system without us having to worry about nonces of the transactions (see Section 2.2.1).

All transactions had the same destination address (0x816641acb67a07517059d62655c7f0d02732b39c) meaning that all transactions in our system were sent to the same account. In [42] is possible to search by the destination address to view all the transactions that our system created during the experiment and even during development and testing.

5.3 System Performance

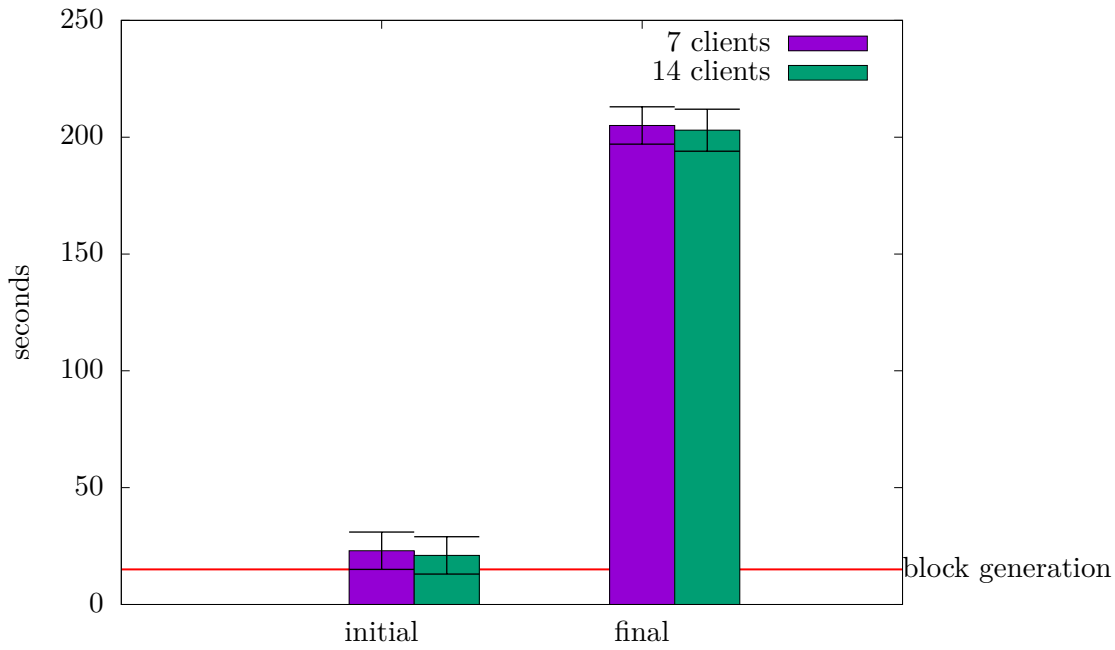

Figure 5.4: Decision latencies

Figure 5.4 is a histogram that shows the latencies of the initial and final decisions in our experimental measurements. The red line represents the average time that the Rinkeby test network takes to generate a block. The red line is placed at $y = 15$, but this value is not a constant, it is an average and the standard deviation is around half a second. This does not mean that there can not be blocks mined in less time. The time that the network takes to generate one block represents the minimal amount of time for a decision to be made.

Table 5.1: Latencies Raw values

Number of clients	Initial Decision	Final Decision	Block Generation	Transaction Request latency	Number of blocks per decision
7	23,67 ± 8	205 ± 8,2	15 ± 0,5	0,02 ± 0,09	1.27 ± 0,47
14	21,74 ± 8,2	203 ± 9,4	15 ± 0,5	0,025 ± 0,1	1.14 ± 0.36

Table 5.2: Distribution of number of blocks per decision

Number of clients	Number of blocks per decision	1 block	2 blocks	3 blocks
7	1.27 ± 0.47	73.54%	25.47%	0.99%
14	1.14 ± 0.36	86.47%	13.12%	0.41%

Table 5.1 shows the raw values used in Figure 5.4 plus the obtained values for what we called transactions request and the number of blocks per decision.

From Figure 5.4 and Table 5.1 we can observe some interesting facts about our system. The first is that the difference in seconds between the delivery of the initial decision and the final decision is around 180 seconds. This large gap makes total sense and meets our theoretical expectation due to the fact that the reading of the blockchain that finds the final decisions is 12 blocks behind the initial reading. Note that, the blockchain only creates one block at a time. On average, the Rinkeby network takes $15 * 12 = 180$ seconds to generate 12 blocks.

An observation is that the time for the decisions to be delivered did not increase when the number of clients increased, in fact, it appears to stay stable. Unfortunately, we can not be sure about this because tests with a greater number of clients are necessary to prove if this observation is valid or not. Another problem with this observation is that for a larger number of clients, the number of blocks per decision will increase. This will happen for two reasons. One is that with more clients, the number of clients that are late will increase and the values that they propose will appear later in the blockchain. The second one is that for a large number of clients, it is impossible for all the values needed to make the decisions to be included in the same block because blocks have a limit in the number of transactions that they can contain. This means that a decision whose values have to fit in two blocks instead of just one will take at least two times the time needed to generate a block ($2 * 15 = 30$ seconds). If the decision occupies three blocks it will take at least three times the time needed to generate a block ($3 * 15 = 45$ seconds) and so forth.

The problem with the number of blocks that a decision occupies can already be observed in our measurements. Table 5.2 measures for the number of blocks per decision. It presents the average and standard deviation for the number of blocks per decisions and the percentage for the number of decisions that occupied 1, 2 and 3 blocks (no decision occupied more than 3 blocks). Although they are less frequent, decisions that occupied more than one block raised the average number of blocks per decision to a value bigger than one, having an impact on the time measured. In fact, the number of decisions per block appears to be reason for the fact that

the measures observed for 14 are faster.

In our system, considering the number of blocks per decision to be around 1.2 and that the block generated rate is 15 seconds we have that the expected value to be near $1.2 * 15 = 18$ seconds. The observed values are a little higher than the theoretical calculation. But if we take into account the standard deviations of the measures we can see that the observed time, for both the initial and final decisions, is not far from what can be expected.

5.4 Fork Measures

After we have seen and discussed the performance of our system we will know discuss the metrics about the forks. Table 5.3 presents the values obtained in our experiments.

By looking at the values in Table 5.3 we observe that by doubling the number of clients and obviously doubling the total number of decisions made by clients, the number of wrong decisions also doubled. It makes perfect sense that with more clients we would measure more wrong decisions but what is interesting is that the percentage of wrong decisions per total decisions made remained stable (with 7,8 % for 7 clients and 7,1% for 14). We can also observe that the number of instances with at least one wrong decision grew very little. We explain this by the fact that despite having more clients (which is more propitious to having more wrong decisions), soft forks in the network have nothing to do with the number of clients that our system has. The frequency of soft forks in the network is independent of the number of clients in BCS. Having more clients will cause the number of wrong instances to increase because we have more clients that can be affected by a soft fork. Analysing the values for the wrong clients per instance, we see that normally a wrong decision affects more than one client. This means that the fork that caused it affected other nodes in the network.

Finally we can observe that the number of wrong final decisions is 0. This means that throughout the entire set of tests the second reading in our system was always correct. The second reading is always correct because we used the value for transaction confirmation in the Ethereum blockchain (as discussed in Section 3.5).

Table 5.3: Fork related metrics

Number of clients	Clients decisions	Wrong initial decisions	Wrong final decisions	Instances with wrong decisions	Wrong clients per instance
7	16 091	1 262	0	477	2.64 ± 1.34
14	33 877	2 407	0	505	4.77 ± 2.55

This chapter presented the experiments performed in order to evaluate BCS. It describes how the experiments were performed and the conclusions that we made by observing the results.

Chapter 6

Conclusions

This dissertation presents BCS, a blockchain based consensus service, that provides incremental consistency to applications. Furthermore, our solution bridges the gap between classic consensus systems and blockchain systems. This is achieved with an interface inspired by classical consensus APIs. Incremental consistency allows our system to work around the fork problem of permissionless blockchains. The problem still exists and may cause inconsistencies, but applications are aware of this and can handle it in a convenient way.

Along the document and particularly in the background (Chapter 2), an overview is presented about the several distributed systems related with the solution being proposed. Those systems were analysed regarding their ability to address problems related to; consensus, consistency, speculative executions and Byzantine fault tolerance. We took special attention to Blockchain systems and discuss the advantages and disadvantages of each one. We know that a solution implemented with a permissioned blockchain is simpler, but we think that a solution that uses permissionless blockchain provides a more interesting system.

A blockchain based consensus service allows the existence of a highly scalable system. We know this is true because of the existence of large-scale blockchain applications such as Bitcoin and Ethereum. For instance, Bitcoin has tens of thousands of participants which proves that the system scales very well. The use of Blockchain also provides very interesting features. A distributed tamper proof ledger that stores all the results from the execution of consensus. To the best of our knowledge, there is no other consensus service that provides a distributed tamper proof ledger.

6.1 Achievements

Before we look at what we achieved, it is important to remember the objectives of this work in Section 1.3. Our goal was to provide: an intuitive classical consensus interface, deal with forks, a consensus service and tolerate Byzantine faults.

We believe that our goals were accomplished with success. In Section 3.3 we can observe a simple API that is coherent with our discussion about a classical consensus API presented in Section 2.1.4.

In Chapter 3 we present our solution that offers a consensus service to client applications. In that same chapter, we explain how the system is able to deal with forks and how the system tolerates Byzantine faults.

In Chapter 5 we prove through our experiments that a consensus service based on blockchain works despite the existence of fork. In our experiments, we did not observe any final decision that was wrong.

6.2 Future Work

Despite the success of work, there is always room for improvement. We will now present a few ideas that we had during the development of our solution. There are some concepts that we explained as part of our solution but were not implemented. Because we did not implement those concepts, we can not be sure that they are possible (there is a large gap between theory and practice). The checkpoint mechanism is one of these concepts.

The implementation of a checkpoint mechanism that offers the functionality described in Section 3.1 would be very interesting. This could be done by implementing a blockchain with support for checkpoint similar to the system presented in [46]. The system presented in [46] implements a checkpoint mechanism on the blockchain itself allowing its size to be constant (which could make reading the entire blockchain for all new clients easier). It works in the following way: periodically, we have a new beginning of time or a new Genesis block (Genesis block is initial blockchain in the blockchain) that contains the balance of all accounts. However, this is useless for our case because it implies the loss of transaction history and our system is based precisely on the history and order of transactions. One way to apply this idea to our system would be to have a similar mechanism, but instead of saving the account balances, the Genesis block would keep the members of each instance of consensus and some other information. The problem with this idea is that it would be necessary to implement a new blockchain, different from existing ones that offered this checkpoint mechanism.

An alternative idea about the implementation of the checkpoint mechanism in BCS is to periodically save the state of the Wrapper in the blockchain. This could be done by periodically, one wrapper is chosen to create a special transaction that contains his view of the state of the system. With the state of one Wrapper saved into the blockchain, all new members could just read this transaction to obtain the state of the system, instead of having to read a potentially large number of older blocks. To do this, a new protocol would have to be created to overcome the challenges of this idea (e.g. the state of the system will probably not fit in just one transaction).

The checkpoint for BCS or the checkpoint for blockchain do not seem to be trivial to implement. Because of this, and the optimizations that it would provide to our system and blockchains in general, we believe that further research about these two topics is needed.

Another idea that we had was to extend our incremental consistency into a system with a real incremental consistency where clients define the level of confidence that they want. Instead of having a second reading of the blockchain that is a fixed number of blocks behind the first, clients could define how many blocks of confirmation they want and how many readings they want. For example, a client says that he wants to receive the decision after one block, after twelve blocks to have a strong guaranty and after fifty blocks because that is the number that it takes for him to trust the decisions as a final one. We think that this is possible to achieve but may have a big cost because it could cause the Wrapper to read the blockchain multiple times. Another argument is that we are not sure that the extra operations provided to clients would be that useful and compensate for the extra costs

Finally, we think that could be interesting to test our solutions in other blockchains system (different then Ethereum), in a system that uses PoS or permissioned blockchains. Our solution implemented in a permissioned blockchain would have much better performance.

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[3] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm." in *USENIX Annual Technical Conference*, 2014, pp. 305–319.

[4] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[5] M. Correia, N. F. Neves, and P. Veríssimo, "From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures," *The Computer Journal*, vol. 49, no. 1, pp. 82–96, 2005.

[6] V. Buterin *et al.*, "Ethereum: a next-generation smart contract and decentralized application platform," *white paper*, 2014.

[7] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

[8] A. Doudou and A. Schiper, "Muteness detectors for consensus with byzantine processes," in *in Proceedings of the 17th ACM Symposium on Principle of Distributed Computing,(Puerto.* Citeseer, 1997.

[9] J. Sousa and A. Bessani, "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation," in *Dependable Computing Conference (EDCC), 2012 Ninth European.* IEEE, 2012, pp. 37–48.

[10] M. Correia, N. F. Neves, and P. Verissimo, "Bft-to: Intrusion tolerance with less replicas," *The Computer Journal*, vol. 56, no. 6, pp. 693–715, 2012.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[12] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative Byzantine fault tolerance," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 45–58.

[13] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[14] M. Correia, G. S. Veronese, N. F. Neves, and P. Verissimo, "Byzantine consensus in asynchronous message-passing systems: a survey," *International Journal of Critical Computer-Based Systems*, vol. 2, no. 2, pp. 141–161, 2011.

[15] F. Fagnani and S. Zampieri, "Randomized consensus algorithms over large scale networks," *IEEE Journal on Selected Areas in Communications*, vol. 26, no. 4, 2008.

[16] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[17] E. B. Nightingale, P. M. Chen, and J. Flinn, "Speculative execution in a distributed file system," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 191–205, 2005.

[18] R. Guerraoui, V. Kuncak, and G. Losa, "Speculative linearizability," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 55–66, 2012.

[19] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing distributed systems using approximate synchrony in data center networks." in *Networked Systems Design and Implementation*, 2015, pp. 43–57.

[20] D. S. Marco Primi. (2013) Libpaxos. [Online]. Available: http://libpaxos.sourceforge.net/

[21] J. Sousa, A. Bessani, and E. Alchieri. (2013) Bft-smart: High-performance Byzantine fault-tolerant state machine replication. [Online]. Available: http://bft-smart.github.io/library/

[22] J. Sousa and A. Bessani, "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation," in *Dependable Computing Conference (EDCC), 2012 Ninth European*. IEEE, 2012, pp. 37–48.

[23] X. L. Blake Mizerany and Y. Qin. (2017) etcd. [Online]. Available: https://github.com/coreos/etcd/tree/master/raft

[24] D. Ports. (2016) Speculative paxos. [Online]. Available: https://github.com/UWSysLab/specpaxos

[25] I.-C. Lin and T.-C. Liao, "A survey of blockchain security issues and challenges." *IJ Network Security*, vol. 19, no. 5, pp. 653–659, 2017.

[26] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference.* ACM, 2018, p. 30.

[27] S. D. Angelis, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone, "Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain," January 2018. [Online]. Available: https://eprints.soton.ac.uk/415083/

[28] K. Finlow-Bates, "Adding trust to cap: Blockchain as a strong eventual consistency recovery strategy," 2017.

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

[30] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *International conference on financial cryptography and data security.* Springer, 2014, pp. 436–454.

[31] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," *self-published paper, August*, vol. 19, 2012.

[32] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Master's thesis, The University of Guelph, 2016.

[33] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Proceedings of the 2nd international conference on Software engineering.* IEEE Computer Society Press, 1976, pp. 562–570.

[34] D. K. Gifford, "Weighted voting for replicated data," in *Proceedings of the 7th ACM Symposium on Operating Systems Principles.* ACM, 1979, pp. 150–162.

[35] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[36] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *Big Data (BigData Congress), 2017 IEEE International Congress on.* IEEE, 2017, pp. 557–564.

[37] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *International Workshop on Open Problems in Network Security.* Springer, 2015, pp. 112–125.

[38] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Incremental consistency guarantees for replicated objects," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, no. EPFL-CONF-223662, 2016, pp. 169–184.

[39] J. Nelson, M. Ali, R. Shea, and M. J. Freedman, "Extending existing blockchains with virtualchain," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[40] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba, "A taxonomy of blockchain-based systems for architecture design," in *Software Architecture (ICSA), 2017 IEEE International Conference on.* IEEE, 2017, pp. 243–252.

[41] C. Svensson. (2016) Web3j. [Online]. Available: https://web3j.io

[42] Rinkeby: Ethereum test network. [Online]. Available: https://www.rinkeby.io/#stats

[43] Ropsten: Ethereum test network. [Online]. Available: https://ropsten.etherscan.io/

[44] E. Foundation. (2017) Geth. [Online]. Available: https://github.com/ethereum/go-ethereum/wiki/geth

[45] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[46] M. Amelchenko and S. Dolev, "Blockchain abbreviation: Implemented by message passing and shared memory," in *Network Computing and Applications (NCA), 2017 IEEE 16th International Symposium on.* IEEE, 2017, pp. 1–7.