Sapienza University of Rome

Ingegneria dell'Informazione, Informatica e Statistica
Ingegneria Informatica, Automatica e Gestionale "Antonio Ruperti"

Master Thesis For The Degree Of Engineering in Computer Science

A Modular Approach for the Analysis of Blockchain Consensus Protocol under Churn

Thesis Advisor
Prof. Bonomi Silvia

Candidate
Dinu Floris Ciprian
1593921

Academic Year 2021-2022

Abstract

Blockchain is an emerging technology that gained a lot of attention in the last years. Many different consensus protocols have been proposed to improve both the scalability and the resilience of existing blockchain. However, most of these solutions have been defined for rather static settings, where the system does not change or changes very slowly, mainly due to failures. Real networks might be subject to a progressive refreshment of the peers participating in the system, known as *churn*. Every blockchain protocol can be seen as the composition of three main distributed building blocks: a) overlay management protocol, b) communication layer, c) agreement primitive.

In this work is proposed a modular approach for analyzing and comparing different consensus protocols used in blockchain under churn conditions. The results from the simulations carried out with the OMNeT++ discrete-event simulation framework help to clarify how these distributed blocks are related and how different churning models impact the blockchain.

Keywords: blockchain, dependability, consensus, omnetpp, pbft, simulation, overlay.

Contents

List of Figures **iv**

List of Tables **v**

Acronyms **vii**

1 Introduction **1**
1.1 Thesis contributions 2
1.2 Thesis outline 2

2 Background **4**
2.1 Blockchain 4
2.1.1 Properties 5
2.1.2 Classification 6
2.2 State Machine Replication 8
2.2.1 Asynchrony 8
2.2.2 Byzantine Fault Tolerance 9
2.3 Broadcast primitives 10
2.3.1 Best-effort broadcast 11
2.3.2 Reliable broadcast 11
2.3.3 Atomic broadcast 12
2.4 Consensus 12
2.4.1 Proof of Work 13
2.4.2 Proof of Stake 13
2.4.3 BFT-based 14
2.4.4 Federated BA 17
2.4.5 Proof of X 19
2.5 Comparison 19
2.6 Conclusion 20

3 Modular Approach for the Analysis of Blockchain Consensus Protocol under Churn **21**
3.1 Introduction 21
3.2 Problem statement 21
3.3 Proposed Framework 22
3.4 Composition Model 22
3.4.1 Overlay Management Protocol 23
3.4.2 Communication layer 25
3.4.3 Agreement Primitive 25
3.5 Churn Model 26
3.6 Load Model 26
3.7 Metrics 26

3.8 Implemented instance 27
3.8.1 Scalable Membership Protocol (SCAMP) 27
3.9 Conclusion 29

4 Simulation Framework **30**
4.1 Introduction 30
4.2 Simulation modeling and analysis 30
4.2.1 Discrete-Event Simulation 31
4.2.2 OMNeT++ 32
4.2.3 OverSim 33
4.3 Implementation 34
4.3.1 Underlay 34
4.3.2 Overlay 35
4.3.3 Application 39
4.3.4 Churn 41
4.3.5 Measurements 42
4.3.6 Simulation composition 43
4.4 Conclusion 44

5 Analysis **45**
5.1 Introduction 45
5.2 Simulation 45
5.2.1 Input parameters 46
5.3 Results 49
5.3.1 Connectivity 49
5.3.2 Strong components 52
5.3.3 Blockchain 55
5.4 Conclusion 60

6 Future Work **62**

Bibliography **63**

List of Figures

2.1 Block structure example in a blockchain. 4
2.2 Blockchain categories. 7
2.3 PBFT process. 15
2.4 Tendermint consensus steps. 16

3.1 Framework components. 23

4.1 Simple Underlay Network modules. 34
4.2 Simple Overlay Host modules. 35
4.3 IOverlay gates. 36
4.4 SCAMP components. 37
4.5 SCAMP definition. 37
4.6 SCAMP simple module. 38
4.7 PBFT implementation components. 40
4.8 PBFT modules definition. 40
4.9 PBFT simple-modules definition. 41

5.1 Continuous churn. 48
5.2 Intermittent churn. 49
5.3 Connectivity with continuous churn. 50
5.4 Intermittent churn connectivity vs. continuous churn. 51
5.5 Connectivity 25-nodes network vs 50-nodes. 51
5.6 Strong components trend over time. 52
5.7 Primary's strong component size. 53
5.8 Time spent into the strong component. 54
5.9 Primary's strong component size trend. 54
5.10 Joins and Leaves from the largest strong component. 55
5.11 Primary blocks creation. 56
5.12 Average connectivity. 56
5.13 Primary blocks creation with churn rate = 4. 57
5.14 Percentage of added blocks with churn rate = 4. 57
5.15 Blockchain trend with continuous churn. 58
5.16 Blockchain trend with intermittent churn. 59
5.17 Blockchain comparison. 59

List of Tables

2.1 Algorithms assumptions 19

Acronyms

ABC Atomic Broadcast

BFT Byzantine Fault Tolerance

FBA Federated Byzantine Agreement

FLP Fischer, Lynch, Paterson

KBR Key-based Routing

OMNeT++ Objective Modular Network Testbed in C++

OMP Overlay Management Protocol

OverSim Overlay Network Simulation Framework

P2P Peer-to-peer

PBFT Practical Byzantine Fault Tolerance

PoS Proof of Stake

PoW Proof of Work

RBC Reliable Broadcast

SCAMP Scalable Membership Protocol

SCP Stellar Consensus Protocol

Chapter 1

Introduction

In the past decade, blockchain has become one of the most widespread technologies for storing transactions in a distributed system characterized by full decentralization, transparency, immutability, and data non-repudiation. Blockchain is an example of an emerging technology that initially solidified its progress and has just lately begun to study its theoretical basis. As a result, numerous algorithmic methods have been developed to enhance the scalability and resilience to Byzantine processes as much as possible. Nevertheless, the vast majority of existing solutions do not include a strong theoretical analysis that demonstrates their formal correctness, and the evaluation is carried out by taking into consideration rather static environments in which the system either does not change at all or changes very slowly as a result of failures. However, real networks are not static and are subject to a progressive refreshment of the system's peers. This is especially true for the networks that lie beneath public blockchains that do not require permission to access their data. This phenomenon is also known as *churn* and, if not properly examined and managed, can have a significant effect on the blockchain's correctness and performance. To the author's knowledge, there are currently no results demonstrating the impact of churn on the blockchain. In this regard, a framework that can be used to evaluate how existing consensus methods for blockchain respond to churn has been defined.

Reviewing and analyzing the current state of the art in blockchain consensus protocols reveals that every blockchain protocol can be viewed as the composition and coordination of the following distributed building blocks:

- an *Overlay Management Protocol (OMP)* responsible for connecting replicas into a logical overlay network and preserve the connectivity of the overlay network graph.
- a Communication Layer implementing one-to-one, one-to-many or many-to-many communication primitives that allow the dissemination of transactions, blocks and consensus-related messages to all interested replicas.
- an Agreement primitive (e.g., a consensus, a leader election, a committee-based voting) that is used to select, validate and attach blocks to the blockchain consistently with other replicas in the system.

These primitives are not independent of one another; rather, they act in synergy. As a result, as the system becomes dynamic, the effect of churn affects not just the overlay network and the OMP, but also any other layer built on top of it.

1.1 Thesis contributions

This research is aimed to define a framework that can be used to analyze different blockchain solutions in dynamic settings and to compare their characteristics. The framework is composed of four main elements:

- a distributed building blocks composition model that allows to define a blockchain protocol as composition of existing distributed building blocks.
- a churn model that allows to characterize the dynamics of the system and to describe the arrival and departure distribution of processes from the system (and in particular at the OMP level).
- a load model that allows to characterize how transactions are generated by clients.
- a set of metrics that allows to analyze blockchain protocols and to compare them under several circumstances.

For the creation of the composition model, consensus protocols from the state of the art were chosen (e.g., Practical Byzantine Fault Tolerance (PBFT) [1], Stellar Consensus Protocol (SCP) [2], XRP [3], Tendermint [4]) and analyzed in order to identify the set of assumptions concerning (i) the overlay network (ii) the communication primitives used and (iii) the type of agreement implemented on top of them. Almost every algorithm either assumes a fully connected overlay network (e.g., PBFT, Tendermint) or an overlay network having the characteristics of a random graph (e.g., SCP).

Regarding communication primitives, the vast majority of papers consider a reliable communication system without elaborating on its implementation. The current state of OMPs and protocols implementing a reliable communication primitive was then evaluated. While the many available solutions can be considered equivalent from a correctness standpoint, they are not equivalent in terms of performance, dependability, and robustness. In order to define a blockchain as a composition of (i) one OMPs, (ii) one (or more) communication primitive(s), and (iii) an agreement primitive, a composition model was implemented in OMNeT++ [5]. This model facilitates the modular evaluation of blockchain algorithms, allowing for the simple modification of parameters such as churn, network size, and block size.

1.2 Thesis outline

Chapter 2 illustrates some of the most important concepts related to the topic that is discussed in this work. It begins with definitions of the blockchain technology and related distributed systems notions, introduces the consensus problem, continues with a brief overview of some state-of-the-art consensus algorithms, and finally summarizes the assumptions these algorithms make.

Chapter 3 states the problem emerged from the previous chapter and proposes a modular approach for the analysis of blockchain consensus protocols under churn settings, in which will be discussed the four main elements of the framework.

Chapter 4 introduces the practical implementation in OMNeT++ of the proposed framework, going through the definition of all the necessary components for the simulation analysis. This chapter contains the author's most significant contribution, namely the application of discrete-event simulation to the composition model identified.

The results of simulations performed with OMNeT++ are presented in Chapter 5, along with how churn and other parameters can affect not only the behavior of the OMP, but also the behavior of the layers built upon it, such as the communication or agreement primitives used by the blockchain protocol.

The final chapter provides a summary of the work accomplished, with an eye toward enhancing some of the aspects regarding the entire components stack of a blockchain protocol.

Part of this thesis work was published as a poster, named A Modular Approach for the Analysis of Blockchain Consensus Protocol Under Churn, to the 5th International Symposium on Foundations and Applications of Blockchain 2022 Conference, by the publisher Schloss Dagstuhl [6].

Chapter 2

Background

Distributed consensus systems have evolved into an essential component of modern Internet infrastructure, powering every major Internet application at some level. Furthermore, blockchain has become one of the most widely used technologies for storing any type of transaction in a distributed system that is characterized by full decentralization, transparency, immutability, and data non-repudiation.

This chapter introduces the necessary background topics for understanding and discussing about blockchain protocols, with the objective to define the assumptions made by each consensus protocol reviewed in this work.

2.1 Blockchain

Blockchain technology is being used in an increasing number of platforms, and it was first conceptualized in 2008, when Nakamoto used the blockchain as the underlying technology of Bitcoin. Bitcoin enabled secure monetary transactions without the need for a central authority such as a bank, demonstrating the power of blockchain technology. Since that time, blockchain technology has been used in a variety of applications ranging from payment processing to online voting.

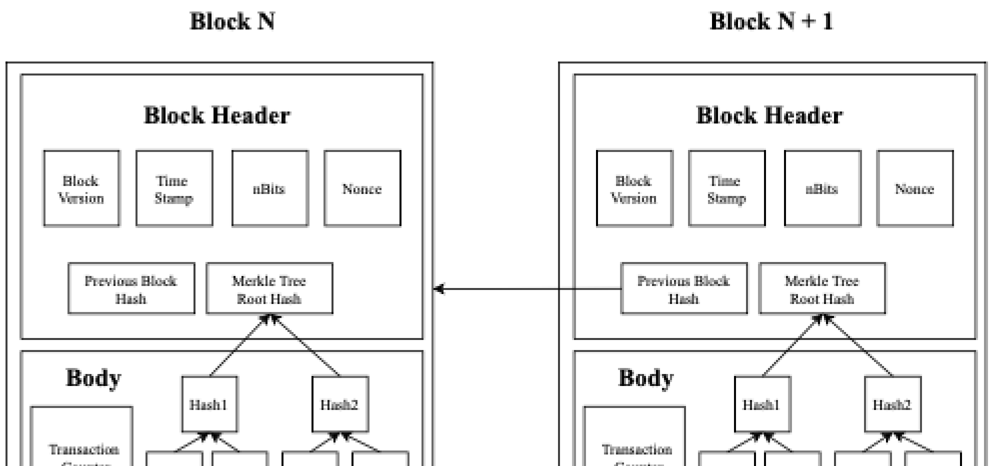


Figure 2.1: Block structure example in a blockchain.

A blockchain consists of a series of blocks that each maintain a record of transactions. Each block includes a header and body. As shown in the figure 2.1, the header contains the following information about the block:

- *Block version:* specifies the set of block validation rules that must be followed.
- *Merkle tree root hash:* the recursively calculated hash value of the Merkle tree root from the block's transaction hashes, as can be seen in figure 2.1.
- *Timestamp:* the time passed in seconds from January 1, 1970 until the creation of the block.
- *nBits:* the target threshold of a valid hash. The generated hash needs nBits or a lower amount of 0's at the beginning of the generated hash.
- *Nonce:* a 4-byte field, which usually starts with 0 and increases for every hash calculation.
- *Previous block hash:* a 256-bit hash value that points to the previous block.

The block header stores the hash of the preceding block, and each block can only have one block that came before it in the chain. The initial block that is added to a blockchain is referred to as the genesis block. There is no block that comes before the genesis block. A transaction counter and a list of different types of transactions make up the body. The maximum number of transactions that can be contained within a block is determined by the maximum block size as well as the size of the transactions that are allowed by the particular blockchain implementation that is being used.

2.1.1 Properties

The blockchain particular structure allows to have the following main properties:

- *Immutability:* Blockchains are resistant to alterations because data in any particular block cannot be edited without modifying all the subsequent blocks, which are each comprised of a cryptographic hash of the preceding block, a timestamp, and transaction data.
- *Decentralization:* There is no the need to trust some central authority that controls the blockchain, nor the other participants to the network. Decentralization provides a trustless environment and reduces points of weakness.
- *Distributed:* Records are replicated across several locations in the network. Clients submit transactions to the nodes implementing the blockchain, and upon validation they are aggregated into blocks and saved to the distributed blockchain.
- *Transparency and traceability:* Blockchain has emerged as a possible solution to implement traceability and transparency by creating an information trail while ensuring security and data immutability.

Based on the blockchain application built upon it, there can be many more advantages in using this kind of technology, where ideally it is very difficult to manipulate information in an arbitrary fraudulent way.

2.1.2 Classification

In general, the nodes that are participating into the system and/or in the validation process that determines which blocks must be linked to the blockchain by each honest node, are the ones responsible for maintaining the blockchain. A distinction between the two types of blockchain needs to be drawn, taking into account the parties that can join and take part to the validation process.

- *Permissionless:* Participation in these blockchains is open to any entity that runs a local instance of the protocol with their digital signature. Participants are not required to reveal their true identity. The fully replicated immutable data storage of permissionless blockchains and their open nature have the following benefits:
  - There is no need to place complete trust in any single entity in the network or a third party.
  - The digital signature of each entity in the network identifies it. Real identities are unnecessary. As a result, permissionless blockchains are pseudonymous.
  - Blockchains are fully replicated immutable data stores that aim to ensure integrity and non-repudiation.

  Permissionless blockchains do, however, suffer from several drawbacks due to their fully replicated and open nature:
  - They have poor performance in terms of processing a high throughput of transaction requests.
  - Since data saved in permissionless blockchains is fully duplicated among all peers, it is accessible to the public. Permissionless blockchains suffer from confidentiality issues if extra security mechanisms are not added to the original protocol.
  - Forking causes permissionless blockchains with the consensus protocols Proof of Work (PoW), Proof of Stake (PoS), and Proof of Authority to never reach consensus finality, resulting in longer wait times for a block to be deemed valid.
- *Permissioned:* As an alternative to permissionless blockchains, permissioned blockchains have been developed. This occurred to answer the necessity for running blockchain technology among a set of known and identified players who must be admitted expressly to the blockchain network. This type of blockchain concept is most suitable for enterprise applications. Even though the nodes do not necessarily trust each other or any third parties, they must nonetheless identify themselves. Permissioned blockchains are frequently favored over their permissionless equivalents because they enable certain use cases, such as:
  - Participation in the consensus protocol is restricted to a small number of nodes that require explicit system configuration in terms of authority permission, software, and hardware.
  - Permissioned blockchains provide greater confidentiality because sensitive transactions can be isolated from public view.
  - For permissioned blockchains forks and non-final decisions could be avoided.

Permissioned blockchains share the same drawbacks and limitations as their permissionless counterparts; in fact, the enhancements for this type of blockchain come at a cost, such as a reduction in decentralization in blockchain networks, as user roles and participation are determined by an administrative entity and transaction validation is performed by predefined nodes.

Based on this major distinction, four different blockchain categories can be drawn, visible also in figure 2.2:

- *Public:* they are permissionless in nature, allow anyone to join, and are completely decentralized. Public blockchains give all nodes on the blockchain equal access to the blockchain, the ability to create new blocks of data, and the ability to validate blocks of data.
- *Private:* a single organization controls permissioned blockchains. The central authority determines who can be a node in a private blockchain. Furthermore, the central authority does not always grant each node equal rights to perform functions. Because public access to private blockchains is limited, they are only partially decentralized.
- *Hybrid:* Permissioned blockchains, as opposed to private blockchains, are governed by a group of organizations rather than a single entity. As a result, consortium blockchains have more decentralization than private blockchains, resulting in higher levels of security.
- *Consortium:* blockchains controlled by a single organization but subject to oversight by the public blockchain, which is required to perform certain transaction validations.

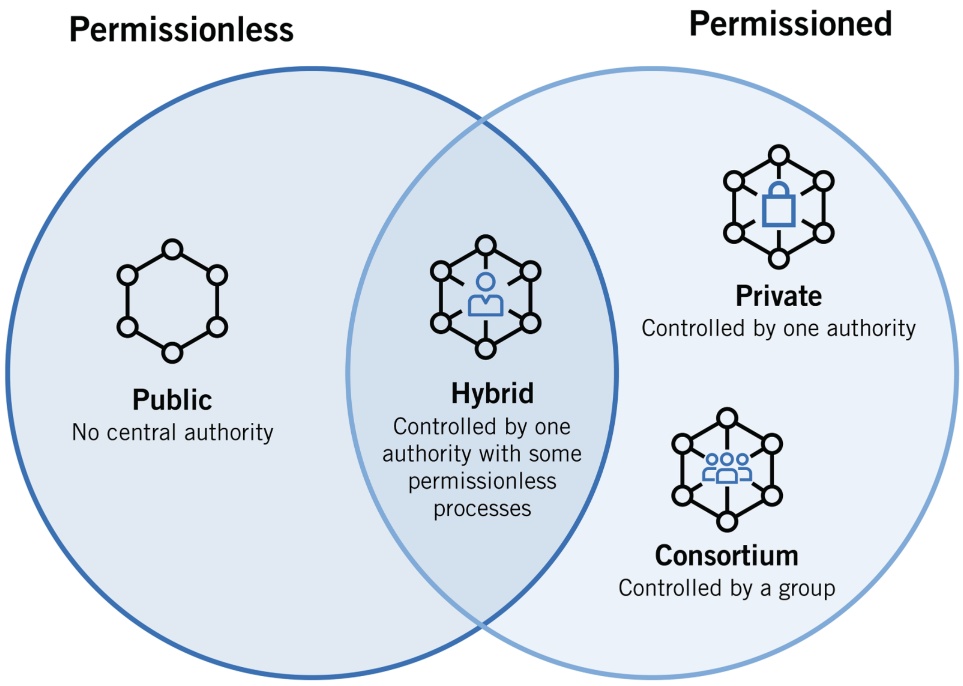


Figure 2.2: Blockchain categories.

2.2 State Machine Replication

Any blockchain relies on some kind of distributed consensus system, because at its heart it is a replicated deterministic state machine.

The Replicated State Machine model is the most studied and widely used approach to creating fault-tolerant distributed consensus. In this model, a deterministic state machine is replicated across a number of processes so that it continues to operate as a single state machine even if some of the processes fail. Depending on its legitimacy, a given transaction may or may not trigger a state transition and produce a result, which is what drives the state machine. A transaction is a database operation that either succeeds or fails immediately; it cannot be in between states. The state machine's state transition function transfers a transaction and the current state to a new state and a return value, controlling the state transition logic. In other contexts, the state transition function may be called application logic.

It is the job of the consensus protocol to arrange the order of the transactions in such a way that the final transactions log may be duplicated in its whole by each and every process. A deterministic state transition function ensures that every process will compute the same state when presented with the same transaction log.

2.2.1 Asynchrony

A fault-tolerant replicated state machine algorithm's goal is to coordinate a network of computers such that they remain in sync while providing a useful service, despite the fact that there may be errors in the system.

Keeping everything in sync requires successful replication of the transaction log; offering a valuable service requires maintaining the availability of the state machine for more transactions. These aspects of the system are traditionally known as *safety* and *liveness*, respectively:

- *Safety:* means that two different processes will agree on the same value, in addition to the value already having been proposed by one of the processes. A method that suggests a value is sometimes referred to as validity, while agreement is also referred to as consistency. In order for a process to be considered valid, it must first settle on a value and then propose that value, even if the value may not be accurate. Colloquially, safety means *nothing bad happens.*

- *Liveness:* is the property that any sent message is eventually delivered. A system that has stalled waiting for a message is *dead* because it cannot make progress. This can be thought of as *eventually something good happens*, where eventually is loosely defined as finite.

A violation of safety implies two or more valid, competing transaction logs. Violating liveness implies an unresponsive network.

Accepting all transactions is a simple way to achieve liveness. And it is uncomplicated to ensure safety by accepting none. Consequently, state machine replication techniques can be viewed as operating on a spectrum between these two extremes. Typically, processes must receive a minimum amount of data from other processes before initiating a new transaction. In synchronous environments, where assumptions are made about the maximum delay of network messages or the maximum speed of processor clocks, it is straightforward to take turns proposing new transactions, poll for a majority vote, and skip a proposer's turn if their proposal does not fall within the synchrony assumptions.

In asynchronous situations, where such assumptions about network delays and processor speeds are not guaranteed, it is considerably more difficult to manage the trade-off. In actuality, the so-called Fischer, Lynch, Paterson (FLP) impossibility result proves that distributed consensus among deterministic asynchronous processes is impossible if even a single process can fail [7]. The argument essentially demonstrates that, given that processes can fail, there are legitimate executions of the protocol in which processes fail at precisely the right moments to prohibit consensus. Consequently, there is no assurance of consensus.

Generally, the usage of timeouts to control particular transitions reflects synchronization in a protocol. In asynchronous systems where messages might be arbitrarily delayed, depending on synchronization (timeouts) for safety can result in a transaction log fork. Relying on synchronization to guarantee liveness may cause the consensus to fail and the service to become inactive. Typically, the first scenario is seen as more serious, as reconciling contradictory logs might be difficult or impossible.

Fundamentally, there are two approaches to circumvent the FLP impossibility conclusion. The first is to employ tighter synchronization assumptions - even very weak assumptions are sufficient, for example, that only crashed processes are suspected of crashing and right ones are not. Typically, this strategy employs leaders, which serve an unique coordinating role and can be bypassed if they are suspected of being defective after a timeout. Such leader-election systems can be challenging to implement in practice.

The second method for overcoming FLP is to include non-determinism, including randomization elements, such that the likelihood of reaching consensus approaches 1.

2.2.2 Byzantine Fault Tolerance

Blockchains have been referred to as *trust machines* due to the fact that they minimize counter party risk by decentralizing control over a shared database. Particularly notable is Bitcoin's ability to withstand attacks and malicious behavior by any member. Historically, consensus protocols tolerant of malicious behavior were referred to as Byzantine Fault Tolerance (BFT) consensus protocols. Due to the similarities of the difficulty to that faced by Byzantine generals attempting to coordinate an attack on Rome using only messengers, when one of the generals may have been a traitor, the term Byzantine was applied [8].

In the event of a crash fault, a process halts. A Byzantine fault can exhibit arbitrary behavior. Crash flaws are easier to handle since processes cannot lie to one another. Systems that only tolerate crash faults can operate according to the simple majority rule, and can therefore withstand the simultaneous failure of up to fifty percent of the system. If the number of failures the system can tolerate is f, such systems must have at least $2f + 1$ processes.

Byzantine failures are more complicated. In a system of $2f + 1$ processes, if f are Byzantine, they can coordinate to say arbitrary things to the other $f + 1$ processes. For instance, with $N = 3$, suppose processes A, B and C are trying to agree on the value of a single bit, and $f = 1$. C, that is byzantine, can tell A that the value is 0 and tell B that it's 1. If A agrees that it's 0, and B agrees that it's 1, then they will both think they have a majority and commit, thereby violating the safety condition. Hence, the upper bound on faults tolerated by a Byzantine system is strictly lower than a non-Byzantine one.

In fact, it was demonstrated that the upper limit on f for Byzantine faults is $f < N/3$, where N is the number of processes belonging to the system. Thus, to tolerate a single Byzantine process, at least $N = 4$ is necessary. Then the faulty process can't split the vote the way it was able to when $N = 3$.

In 1999, Castro and Liskov published PBFT, which provided the first optimal Byzantine fault tolerant algorithm for practical use [1]. By processing tens of thousands of transactions per second, it established a new standard for the applicability of Byzantine fault tolerance in industrial systems. Despite this accomplishment, Byzantine fault tolerance was viewed as costly and mostly useless, and the most common implementation was difficult to develop upon. Consequently, despite a renaissance in scholarly interest, which included multiple better variants, implementations and deployments made little progress. In addition, PBFT offers no assurances if a third or more of the network violates safety standards.

2.3 Broadcast primitives

A brief overview of the broadcast primitives is required before proceeding with the topics related to this thesis. The broadcast communications abstractions are utilized to disseminate information among a set or processes. These abstractions can be differentiated from one another based on the reliability of the dissemination. For example, best-effort broadcast ensures that all processes that are correct will deliver the same collection of messages if the senders are correct. The more robust forms of reliable broadcast guarantee that this property will hold true even if the senders of the messages crash while they are in the process of broadcasting them. Even more robust broadcast abstractions are suitable for the arbitrary-fault model, and they guarantee consistency with the Byzantine process abstractions.

In this particular setting, the term "reliability" typically refers to the condition in which it is assumed that messages passed back and forth between two processes will not be lost or duplicated and will be received in the same order in which they were sent. Implementations of this abstraction typically take the form of dependable transport protocols like TCP, which are used on the internet. The application does not need to deal explicitly with issues such as acknowledgments, timeouts, message retransmissions, flow control, and a number of other issues that are encapsulated by the protocol interface because they are handled by the reliable point-to-point communication protocol. It is convenient to rely on broadcast abstractions in the many situations in which more than two processes need to operate in a coordinated manner. These situations can arise in a variety of contexts. These enable a process to send a message to another process within a group of processes, and they ensure that all of the processes in the group are in agreement regarding the messages they deliver. A naive transposition of the reliability requirement from point-to-point protocols would require that no message broadcast to the group be lost or duplicated, i.e., the processes agree to deliver every message broadcast to them. This would be the case if the reliability requirement was taken directly from point-to-point protocols.

There are a multitude of reliability guarantees regarding broadcast abstractions, that cover crash-stop processes or even arbitrary-faulty processes. Also, ordering of the messages and the set of processes that deliver the messages are part of these guarantees.

To replicate its state on other processes, a process must have access to basic communication primitives that allow it to distribute or transfer data. In this section, the focus will be on best-effort, reliable and atomic broadcast abstractions with crash-stop process abstractions.

2.3.1 Best-effort broadcast

One of the benefits of using a broadcast abstraction is that it enables a process to send a message to all of the processes in a system, including itself, in a single operation. The sender is solely responsible for ensuring the reliability of the transmission when using the best-effort method of broadcasting. Therefore, it is not necessary for any of the other processes to be concerned with ensuring the dependability of the messages they have received. On the other hand, there are no delivery guarantees provided in the event that the sender is faulty. Best-effort broadcast is characterized by the following three properties:

- *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.
- *No duplication:* No message is delivered more than once.
- *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

Validity is a liveness property, whereas the no duplication property and the no creation property are safety properties. Broadcasting a message simply consists of sending the message to every process in the system using perfect point-to-point link abstractions.

2.3.2 Reliable broadcast

Messages sent using best-effort broadcast will always be delivered, provided that the sender does not experience any problems. In the event that the sender is unsuccessful, it is possible that some processes will deliver the message while others will not. In other words, they do not concur on the manner in which the message should be conveyed. Even if the senders of these messages crash in the middle of the transmission, the semantics of a reliable broadcast algorithm ensure that the correct processes agree on the set of messages that they deliver. This is the case even though the senders of these messages.

The properties of the best-effort broadcast abstraction are extended by the specification of reliable broadcast, and this extension adds a new liveness property that is called agreement. The other characteristics have not been altered.

- *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

In essence, Reliable Broadcast (RBC) enables a message to be eventually delivered once on all correct processes.

2.3.3 Atomic broadcast

Another, more stringent primitive is atomic broadcast, which satisfies RBC and adds an additional property:

- *Total Order:* if correct processes p and q deliver m and *m'*, then p delivers m before *m'* iff q delivers m before *m'*.

Atomic broadcast is thus a reliable broadcast where values are delivered in the same order on at each process. Note this is exactly the problem of replicating a transaction log.

Intuitively, consensus and Atomic Broadcast (ABC) appear to be fairly similar to one another; however, the most important distinction between the two is that ABC is a protocol that runs continuously, whereas consensus anticipates that it will terminate. Having said that, it is common knowledge that one can be simplified into the other. Simply determining which value should be atomically broadcast first makes it very simple to reduce consensus to ABC. ABC can be reduced to consensus by running multiple instances of the consensus protocol in succession; however, certain nuanced factors need to be taken into account, particularly for the treatment of Byzantine faults.

2.4 Consensus

Blockchain is a trustless distributed ledger which allows transactions to take place in a decentralized manner. To establish any kind of trust within such networks, the nodes need to reach consensus on which blocks containing transactions are accepted into the distributed ledger.

The processes use consensus to agree on a common value among the values they initially propose. Consensus-building is one of the most fundamental issues in distributed computing. A consensus problem must be solved by any method that enables numerous processes retain a common state or decide on a future action in a paradigm where some processes may fail.

Into the wild there are multiple variants of consensus: regular, uniform, randomized with crash-stop processes, logged with crash-recovery processes, and byzantine consensus with arbitrary-fault processes.

The focus here is on a trivial consensus abstraction. The standard definition of the consensus primitive satisfies the following:

- *Termination:* every correct process eventually decides.
- *Integrity:* every correct process decides at most once.
- *Agreement:* if one correct process decides v_1 and another decides v_2, then $v_1 = v_2$.
- *Validity:* if a correct process decides v, at least one process proposed v.

In other words, the process that defines how the participants involved in the network come to an agreement on the blocks that are to be appended to the blockchain, and therefore on the transactions that are contained within those blocks, is called a consensus algorithm. This algorithm is at the core of the blockchain technology. To get into additional details, the consensus problem necessitates that multiple processes come to a single point of agreement over a single value. Consensus protocols need to be resilient since there is a possibility that some of the processes will fail or be unreliable in other ways. The processes need to find some way to present their candidate values, communicate with one another, and come to some kind of agreement regarding a single consensus value. In the context of a blockchain, this signifies that the nodes participating in the network are required to communicate with one another and keep the truth even in the presence of defects that may be byzantine in nature.

Every participant in an open peer-to-peer network occupies a level of similar importance in the hierarchical structure. This has an effect on how agreement is reached and adds another layer of complication because there are no restrictions placed on who can join or leave the network. This makes the task of reaching consensus even more challenging than it currently is. The possibility exists that the decentralized public network will have a significant number of nodes, which will raise the bar for communication complexity, as well as the latency and attack surface. Because there is no upper bound on the amount of time it takes to update the state in this system, it is impossible to guarantee consensus in a deterministic manner.

In the following sections some of the most widespread consensus algorithms are described, with a particular eye on the assumptions they make regarding the overlay network, the communication layer and the agreement primitive. Subsequently, a brief comparison between these algorithms will be made.

2.4.1 Proof of Work

The nodes will compete against one another, and only the winner will be able to add the block to the blockchain. Participation requires to solve a hash cryptographic challenge. Everyone who is attempting to find the nonce that will solve the problem is able to do so thanks to proof-of-work, but the one who has the most processing power will have a larger chance of becoming the winner. This strategy is fraught with complications, including a high latency, a slow transaction rate, and the possibility of a security breach. However, due to the extensive hardware resources that must be possessed by the attacker, it is extremely difficult to compromise this form of consensus mechanism.

It would not be in anyone's best interest to implement Proof of Work (PoW) within permissioned blockchains. Since the blockchain in permissioned blockchains is not available to the general public, there is no requirement for the concept of persistency that is utilized in PoW. If it were employed, it would be inefficient in terms of both the amount of computational power and the amount of energy that would be required.

Proof of work was later popularized by Bitcoin as a foundation for consensus in permissionless decentralized network, in which miners compete to append blocks and mint new currency, each miner experiencing a success probability proportional to the computational effort expended.

2.4.2 Proof of Stake

Proposed as an alternative to PoW. Under Proof of Stake (PoS), proposals are made by, and voted on, those who can prove ownership of some stake of coins in the network. While eliminating the excessive energy of costs of PoW, naive implementations of PoS are vulnerable to so called *nothing-at-stake* attacks, wherein validators may propose and vote on multiple blocks at a given height, resulting in a dramatic violation of safety, with no incentive to converge.

PoS implementation within permissioned blockchains would be undesirable. A miner with the biggest network stake has the greatest chance of being selected to create a new block. In a permissioned blockchain, however, such a miner may add transactions that would not ordinarily be permitted by the authorities, so gaining greater influence in the network than the authorities.

2.4.3 BFT-based

Different protocols were designed to achieve consensus considering Byzantine settings. Generally, these protocols proceed with a series of views where each view has a proposer, known as primary. Nodes must know each other in order to participate to consensus. Paxos-like protocols are similar, require a number of nodes $> 2f + 1$, but they do not consider the byzantine model. PBFT is an example of this kind of algorithm, in which it is required that all nodes agree upon the same set of trusted validators.

PBFT

PBFT is one of the first solutions to the Byzantine Generals problem [8]. The PBFT model works by providing a practical Byzantine state machine replication that accepts malicious nodes. PBFT offers both liveness and safety provided at most $n - 1$ out of a total $3n$ replicas are simultaneously faulty. Liveness means clients eventually receive replies to their requests. Safety means the replicated service satisfies linearizability, that is, it behaves like a centralized implementation that executes operations at one time. PBFT works under the following assumptions:

- The distributed system is asynchronous: Nodes are connected by a network, and the network may fail to deliver messages, delay them, duplicate them, or deliver them out of order.
- Faulty nodes are allowed to behave freely, provided that their failures are independent. To ensure this, each node must have unique implementations of the service code and operating system, as well as have distinct administrator and root passwords.
- To cause the maximum harm to the replicated service, a powerful adversary can coordinate malfunctioning nodes, delay transmission, or delay correct nodes. However, an attacker cannot continuously delay proper nodes. In addition, the adversary and the compromised nodes under its control are computationally constrained, preventing them from subverting the cryptographic mechanisms employed between the nodes to prevent spoofing and replays, and to detect damaged messages.

The nodes in a PBFT network, known as replicas, are comprised of a primary node, known as the leader, and backup nodes. All nodes are in constant communication with one another in an attempt to reach a consensus state. For each broadcast, each node must demonstrate that the message originated from a particular peer node using public-key signatures and the message's integrity using message authentication codes (MAC). Each round of consensus in PBFT algorithm works as described in the following steps:

1. A client sends a request to the leader to invoke an operation. This starts a three-phase protocol to atomically multicast the request to the replicas. The three phases are pre-prepare, prepare, and commit.
2. In the pre-prepare stage the leader multicasts the request to the backups.

3. In the prepare stage, the backups that accept the pre-prepare message send an acknowledgment message to all other nodes.

4. After the nodes are prepared, in the commit stage, the nodes send the commit message to all other nodes. If a node receives valid commit messages from more than $n-1$ nodes, then they carry out the client request and send the reply to the client.

5. The client waits for $\frac{n-1}{3}+1$ replies from different replicas with the same result to ensure it has the correct result, that is, matching replies from one more node than the faulty number of nodes. This is the result of the operation.

An illustration of this process can be seen in figure 2.3 where the node numbered 3 is the faulty node.

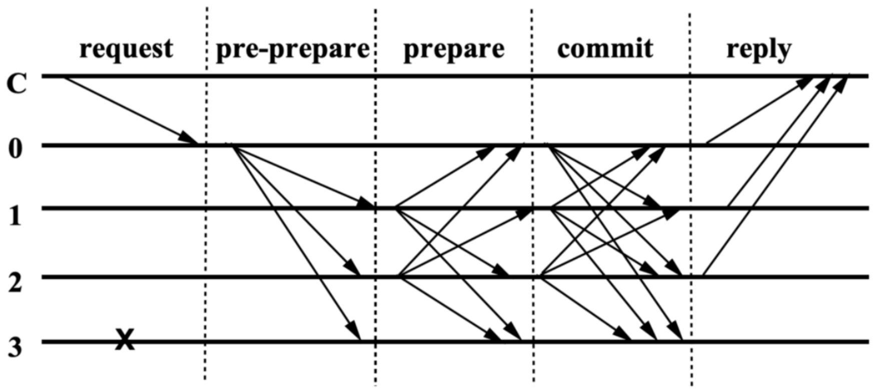


Figure 2.3: PBFT process.

C represents the Client. 0 represents the leader node and the rest of the numbers represent the backup nodes.

The nodes must begin in the same state and be deterministic, meaning that the execution of an operation in a given state and with a particular set of parameters must always return the same outcome. Due to the fact that nodes are constantly communicating, the model is only effective with a small number of nodes; consequently, it is a consensus mechanism that is ideal for permissioned blockchain systems.

Tendermint

Tendermint is the first blockchain system to demonstrate how the BFT consensus can be obtained in a PoS environment. It consists of two primary components: the Tendermint Core consensus engine and the Application Blockchain Interface, its underlying application interface (ABCI). The Tendermint core is responsible for deploying the consensus algorithm, whereas the ABCI can be used to deploy any blockchain application written in any language.

The consensus algorithm is dependent on a collection of validators. It is a round-based algorithm that selects a proposer from a group of validators. In each round, the proposer suggests a new block with the most recent height for the blockchain. The proposer is chosen via a deterministic round-robin procedure that ultimately relies on the validators' vote strength. On the other hand, the voting power is proportionate to the validators' security deposit.

The consensus algorithm consists of three steps (propose, pre-vote, and pre-commit) in each round bound by a timer equally divided among the three steps, thus making it a weakly synchronous protocol. These steps signify the transition of states in each validator. Figure 2.4 illustrates the state transition diagram for each validator. At the beginning of each round, a new proposer is chosen to propose a new block. The proposed block needs to go through a two-stage voting mechanism before it is committed to the blockchain.

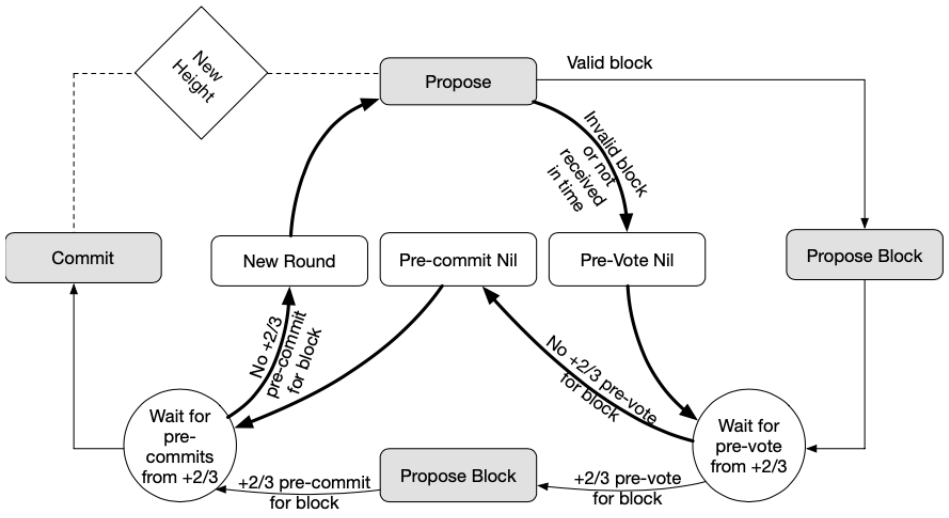

New Height
Propose
Valid block
Invalid block or not received in time
Commit
New Round
Pre-commit Nil
Pre-Vote Nil
Propose Block
No +2/3 pre-commit for block
No +2/3 pre-vote for block
Wait for pre-commits from +2/3
Wait for pre-vote from +2/3
+2/3 pre-commit for block
Propose Block
+2/3 pre-vote for block

Figure 2.4: Tendermint consensus steps.

When a validator receives the proposed block, it validates the block at first, and if okay, it pre-votes for the proposed block. If the block is not received within the propose timer or the block is invalid, the validator submits a special vote called Prevote nil. Then, the validator waits for the pre-vote interval to receive pre-votes from the super-majority (denoted as $+\frac{2}{3}$ of the validators. A $+\frac{2}{3}$ pre-votes signifies that the super-majority validators have voted for the proposed block, implying their confidence on the proposed block and is denoted as a Polka in Tendermint terminology. At this stage, the validator pre-commits the block. If the validator does not receive enough pre-votes for the proposed block, it submits another special vote called Precommit nil. Then, the validator waits for the pre-commit time-period to receive $+\frac{2}{3}$ pre-commits from the super- majority of the validators. Once received, it commits the block to the blockchain. If $+\frac{2}{3}$ pre-commits not received within the pre-commit time-period, the next round is initiated where a new proposer is selected, and the steps are repeated. To ensure the safety guarantee of the algorithm, Tendermint is also coupled with locking rules. Once a validator pre-commits a block after a polka is achieved, it must lock itself onto that block. Then, it must obey the following two rules:

- it must pre-vote for the same block in the next round for the same blockchain height,
- the unlocking is possible only when a newer block receives a polka in a later round for the same blockchain height.

With these criteria, Tendermint ensures that consensus is secure when fewer than one-third of validators display byzantine behavior, ensuring that contradictory blocks are never committed at the same blockchain height. In other words, Tendermint assures that under this assumption no fork will occur. Since Tendermint prioritizes safety over availability, it has one specific flaw. It requires

100% uptime of its $+\frac{2}{3}$ (super-majority) validators. If more than one-third ($+\frac{1}{3}$) are validators are offline or partitioned, the system will stop functioning. In such cases, out-of-protocol steps are required to tackle this situation.

To encourage the proposer and the validator to participate in the consensus process, for instance, an incentive mechanism can be included. A node can become a validator by posting a specified amount of collateral. In the event that the corresponding validator misbehaves, the deposit is destroyed, deterring the validator from launching any network attacks. All Proof-of-Stake (PoS) attacks can be effectively countered by combining the consensus algorithm with a carefully crafted reward and punishment system.

2.4.4 Federated BA

Each node in a Federated Byzantine Agreement (FBA) system is not need to be known and validated in advance, membership is open, and control is decentralized. Nodes can determine whom to trust based on external criteria. Before considering a transaction finalized, a node awaits the consensus of the vast majority of other nodes. Individual node decisions give rise to quorums at the system level. These protocols have minimal latency and trust flexibility.

FBA was created for decentralized networks, which implies that no single authority determines which nodes are permitted to participate in the consensus process. To participate in the consensus process, however, other nodes must incorporate the new node into their quorum slice. A central authority could still determine whether or not nodes are permitted to add nodes to the consensus process. FBA is therefore compatible with both permissioned and permissionless blockchain networks, but it is best suited for systems that require a large number of validators.

SCP

Stellar presents a new model known as FBA. It accomplishes robustness through quorum slices and individual trust decisions made by each node, which collectively define quorums at the system level. Similar to how different networks' peering and transit decisions now unite the Internet, slices connect the system together. The stellar consensus protocol is an FBA-specific architecture. It presupposes a list of members that is unanimously accepted and supports open membership. SCP has low computing and financial requirements, hence decreasing the entry threshold. FBA is a model suitable for worldwide consensus. In FBA, each participant knows of others it considers important. It waits for the vast majority of those other to agree on any transaction before considering the transaction settled. In turn, those important participants do not agree to the transaction until the participants they consider important agree as well. The transaction will be impossible to roll back. FBA's consensus can ensure integrity of a financial network and a decentralized control. SCP's safety is optimal for an asynchronous protocol, in that it guarantees agreement under any node-failure scenario that admits such a guarantee. SCP has four key properties:

- decentralized control.
- low latency.
- flexible trust, users have the freedom to trust any combination of parties they see fit.

- asymptotic security, no amount of computing power can overtake the network-

Protocols such as PBFT require all parties to agree on an exact participant list. Membership in byzantine agreement systems is typically determined by a central authority or through closed negotiation. Ripple, for instance, publishes a starter membership list that participants can edit for themselves, with the hope that the edits are either inconsequential or replicated by an overwhelming majority of participants. However, in practice, a great deal of power is concentrated in the maintainer of the starter membership list. Tendermint takes a different method by basing membership on evidence of stake, yet resource ownership is again related to trust. SCP is the first byzantine agreement protocol that provides each participant with the greatest freedom in selecting which combination of other participants to trust.

An FBA system runs a consensus protocol that ensures nodes agree on slot contents. A node v can safely apply update x in slot i when it has safely applied updates in all slots upon which i depends and, additionally, it believes all correct functioning nodes will eventually agree on x for slot i. A node cannot later change its mind. FBA determines quorums in a decentralized way, by each node selecting what we call quorum slices.

A node can have one or more slices. Nodes may choose slices according to arbitrary criteria, such as reputation or financial arrangements. A quorum is the minimum number of nodes required to reach consensus. A quorum slice is the subset of a quorum that convinces a certain node to concur. It is possible for a quorum slice to be smaller than a quorum. Because all nodes accept the identical slices, traditional systems do not differentiate between slices and quorums. In certain circumstances, no individual node may have complete knowledge of every other node in the system, yet consensus should still be attainable. A quorum intersection is required at all times. The overlap cannot consist of misbehaving nodes.

In this protocol the two fundamental properties of safety and liveness assume the following definitions:

- Safety: a set of nodes in an FBAs enjoy safety if no two of them ever externalize different values for the same slot.
- Liveness: A node in an FBAs enjoys liveness if it can externalize new values without the participation of any failed nodes.

So, the difference with PoW and BFT protocols is that Stellar has open membership, there is not a central authority that governs the admission to the validators list. A node that joins, participates in consensus after choosing a quorum slice. Open membership implies decentralization. Regarding the preference between liveness, safety and fault tolerance, SCP prefers safety, unlike Bitcoin that prefers liveness.

XRP

Ripple is a network created to enable the transfer of cash between multinational banks and organisations. Ripple utilizes a distributed ledger system in which all Ripple nodes maintain a record of all funds. In Ripple, each node maintains its own so-called Unique Node List (UNL). Those transactions are included on the ledger if 80% of the nodes in the nodes UNL concur with a candidate set of transactions. According to the Ripple white paper, the ledger will remain accurate so long as no more than 20% of nodes are flawed.

The Ripple Protocol Consensus Algorithm (RPCA) works with frequent voting rounds to decide what transactions are added to the ledger. The RPCA works in 4 steps each round:

1. All valid transactions that the node has received are put into the candidate set of the node.
2. All nodes collect the candidate sets of the nodes in their UNL and votes on each transaction.
3. The transactions that receive sufficient votes are moved to subsequent rounds. Transactions that receive insufficient votes are either discarded or included in the candidate set of the next round.
4. For transactions to be added to the ledger at least 80% of the nodes UNL must agree on a transaction. The ledger is closed when all transactions that receive a minimum of 80% of the votes in a UNL are added to the ledger.

Ripple is a permissioned blockchain system. The network has high expectations of trust (80% of the network must be trustworthy) this is only realistically possible in a controlled and permissioned environment.

2.4.5 Proof of X

PoCapacity techniques that monopolize disk capacity are intriguing due of their superior resource efficiency compared to PoW. A participant must verify that it has saved a portion of a file for future usage before PoRetrievability can be satisfied. With PoAuthority, individuals earn the privilege to become validators, therefore they are incentivized to defend the transaction process since they do not want their identities associated with a poor reputation.

2.5 Comparison

In this section will be summarized some of the characteristics of the above cited algorithms along with the assumptions they make about the overlay network they use, about the communication layer and what type of agreement primitive they use.

Implementation	***Overlay***	***Communication***	***Agreement***
PBFT	fully-connected network	best-effort broadcast	BFT
Tendermint	Peer-to-peer (P2P)	reliable broadcast	BFT + PoS
Stellar	random graph	reliable broadcast	FBA
XRP	random-graph	reliable broadcast	FBA

Table 2.1: Algorithms assumptions

PBFT assumes a distributed system that is asynchronous, with nodes being connected by a network that may fail to deliver messages, delay them, duplicate them or deliver them out of order, thus resulting in a best-effort broadcast communication primitive. The algorithm though, relies on weak synchrony to provide liveness, otherwise it could be used to implement consensus in an asynchronous system, which is not possible.

Tendermint makes strong assumptions about the P2P network connectivity and availability during crisis, since validators have to collect votes from each other, therefore they assume a reliable broadcast. Also, it makes assumptions about the participants capabilities to keep time.

SCP and XRP both assume a random graph as overlay network, since nodes in this algorithm have to communicate with the nodes they have in their quorum slice and UNL respectively. In particular, in SCP quorum slices are like network reachability and quorums analogous to transitive reachability. Also, it make weak-synchrony assumption for the internal nomination protocol. XRP, on the other side, assumes byzantine ability, which states that all nodes - even Byzantine ones - cannot send different messages to different nodes.

2.6 Conclusion

Numerous blockchain-related concepts have been covered in this chapter with the intention of clarifying what are the components of a blockchain protocol and which assumptions they take about the underlying components used. From the previous section it can be seen that consensus algorithms assume fully connected networks or P2P network to support them with different broadcast primitives built upon. Nevertheless, none of them has been designed to support interleaving of processes and therefore dynamic settings of the environment. A phenomenon like churn may heavily impact how these components work together and may pose at risk the guarantees offered by the algorithms, like agreement or validity.

These matters will be covered in greater depth in the following chapter.

Chapter 3

Modular Approach for the Analysis of Blockchain Consensus Protocol under Churn

3.1 Introduction

This work is situated within the context of an investigation into the efficiency of blockchain-based solutions. This technology is gaining in popularity on a continuous basis and is receiving a great deal of attention as a result. There have been a lot of different solutions studied and defined, but the majority of them were for rather static settings. The real world, on the other hand, is dynamic; it changes over the course of time and the set of processes that contribute to the longevity of a system is always evolving. The phenomenon known as churn, which represents the continuous and interleaved process of arrival and departure of nodes, is what accurately depicts the dynamics of the system as a whole.

This research is aimed at defining a framework that can be used to analyze different blockchain solutions in a more severe and dynamic setting that includes churn and to compare them in order to select the one that is the best fit for the environment that is being taken into consideration.

The proposed framework that enables a modular approach to the analysis of blockchain consensus protocols under churn is going to be described in this chapter.

3.2 Problem statement

It becomes clear that the primitives described in chapter 2 that make up a blockchain protocol are not independent of each other, but they rather work in synergy. Following a bottom-up approach, starting from the overlay component, it can be observed that its characteristics heavily impacts the immediate layer above it, that is the communication layer. In a case in which the overlay network gets partitioned temporarily, due to factors like churn, the communication layers suffers this issue, and therefore some properties of the reliable broadcast may not be guaranteed anymore. Therefore, it is fundamental to have a resilient overlay network when partitions happen. In this way, the properties of the layer above it can remain satisfied.

The communication layer, that is in charge of sending and receiving messages to/from neighbors, can be highly impacted by network partitions of participants leaving it. In fact, the delivery of messages may not be guaranteed anymore. The messages can be both consensus-related messages and application messages, i.e, transactions that have to be stored on the blockchain. It comes natural to say that this block influences the performance of the third and last block, that is the agreement primitive. Since messages cannot reach all their destinations, the consensus protocol running above the communication layer may suffer, resulting into a halt or even worse, a blockchain fork. Since these blocks work in synergy and are highly interconnected, decisions taken for each one will impact the others and the entire blockchain protocol. Therefore, it is important to understand how these blocks must be built, the choices that have to be taken, the assumptions about the system model, and the characteristics that the application must accomplish.

From the preliminary studies of this work, it emerged that the phenomenon known as churn was not properly analyzed and managed when blockchain protocols have been designed. They consider rather static environments where the system does not change or changes very slowly. In this work, more severe settings were considered in order to evaluate how churn impacts these components, since in real world networks (especially those underlining public permissionless blockchain) are not static and are subject to a progressive refreshment of the peers participating in the system. At the best of the author's knowledge there are not studies related to this kind of issue.

In the following sections will be depicted the proposed framework that has the aim to facilitate the analysis of blockchain protocols under dynamic churn events, and therefore understand how the blockchain components are affected by these external factors.

3.3 Proposed Framework

The framework is developed around a distributed building blocks composition model that allows to abstract the relevant elements composing the system. Then, a churn model is used to capture the dynamics of the system and to represent how nodes are arriving and departing from the system. Subsequently, a transactions load model is introduced, that allows to characterize the way in which transactions are submitted to the system, translated into blocks that are attached to the blockchain. Finally, the framework identifies several metrics that can be used to assess the behavior of the blockchain from the performance and dependability point of view.

This section will first describe the distributed building blocks composition model, that are at the center of the framework, as illustrated in the figure 3.1, the churn and load models, along with the metrics used to evaluate the system behavior in the dynamic settings.

3.4 Composition Model

Reviewing and analyzing the consensus protocols in section 2.4 it comes out that every blockchain protocol can be split into three main distributed building blocks:

- *OMP*: responsible for connecting replicas into a logical overlay network and preserve the connectivity of the overlay network graph.
- *Communication Layer:* implementing one-to-one, one-to-many and many-to-many communication primitives that allow the dissemination of transactions and blocks to all interested

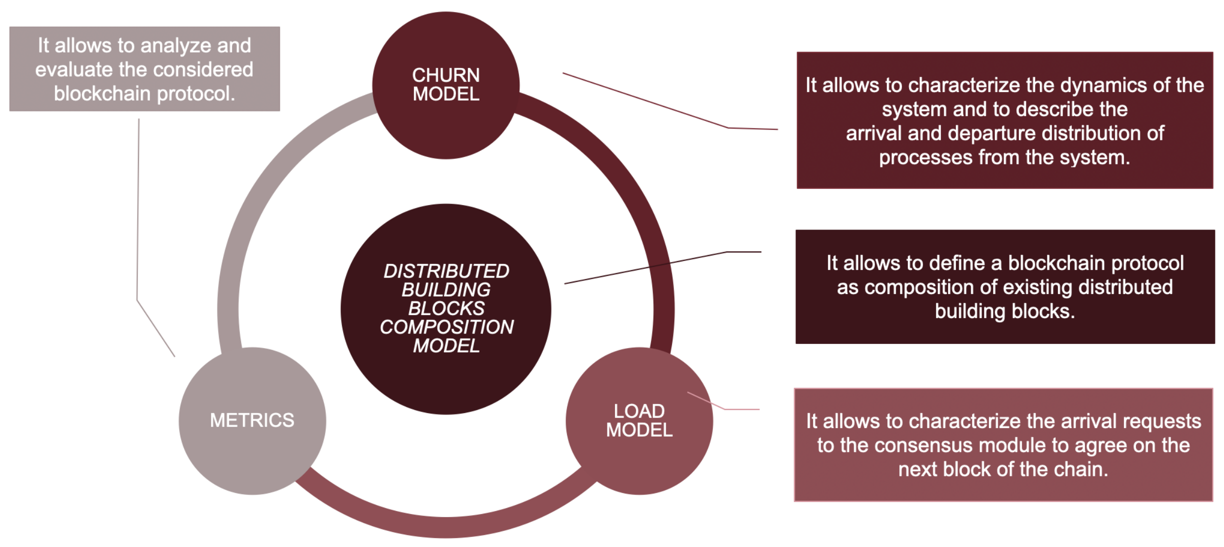

It allows to analyze and evaluate the considered blockchain protocol.
CHURN MODEL
It allows to characterize the dynamics of the system and to describe the arrival and departure distribution of processes from the system.
DISTRIBUTED BUILDING BLOCKS COMPOSITION MODEL
It allows to define a blockchain protocol as composition of existing distributed building blocks.
METRICS
LOAD MODEL
It allows to characterize the arrival requests to the consensus module to agree on the next block of the chain.

Figure 3.1: Framework components.

replicas.

- *Agreement primitive:* (e.g., a consensus, a leader election, a committee-based voting) that is used to select, validate and attach blocks to the blockchain consistently with other replicas in the system.

Any blockchain protocol has at its core an agreement primitive that takes basic assumptions in terms of the communication layer subsystem that is in charge to exchange information. Downstream of this fact, assumptions are also made about the underlying system, that is the overlay supporting the communication.

During this work, this conceptual framework has been decomposed into a modular framework where one can take any consensus algorithm or agreement protocol, make the necessary assumptions regarding the communication primitives, use specific instances of the protocols that are going to support these assumptions and test them over a specific overlay management protocol that is connecting the processes in the network.

In the parts that follow, these blocks will be explained, and for each one, various options for constructing a blockchain consensus mechanism will be examined.

3.4.1 Overlay Management Protocol

In this section, the first distributed block that was identified will be discussed, along with some associated ideas concerning overlay networks and the many forms those networks can take. The first distributed block identified covers the overlay network that is built on top of the underlying physical computer network. Therefore, in a blockchain protocol an OMP plays a crucial role, since it has to ensure that there are no network partitions and at the same time must handle joining and leaving operations of the participants from the network. The main goal of any OMP is properly arranging the overlay to keep as much as possible some desired global properties over the time, despite the continuous and interleaved process of arrival/departure of nodes, i.e., churn. A continuous and concurrent replacement of nodes may permanently determine the partition of the overlay, and an OMP is set up in order to evade this.

What follows is a discussion of the overlay network that can be derived and handled by an OMP, along with advantages and disadvantages.

Overlay Network

Over the course of the past several years, peer-to-peer Internet applications for the purpose of data sharing have grown in popularity to the point where they are now one of the primary producers of traffic on the Internet. Their peer-to-peer approach has been proposed as the underlying model for a wide variety of applications, ranging from storage systems and cooperative content distribution to web caching and communication infrastructures. This is due to the fact that these applications all share a common goal of improving efficiency and reducing costs. The storage and retrieval of objects, as well as the routing of messages, are both handled by existing peer-to-peer systems through the use of overlay network protocols.

The peers (nodes) of a peer-to-peer system collaborate to create an overlay network, which is built on top of the underlying physical computer network but operates largely independently from it. Any overlay should demonstrate a topology that is able to support a P2P application in a way that is both efficient and scalable while retaining a level of dependability that is adequate.

These overlay protocols can be broadly categorized as either structured or unstructured according to the limits that are put on how peers are ordered and where stored objects are kept. This distinction can be made depending on whether or not the constraints are applied.

Unstructured overlay networks have emerged as a viable solution to settle such issues in order to effectively support large-scale dissemination and flooding-based content searching. This is because unstructured overlay networks are able to combine the advantages of both structured and unstructured networks. Unstructured overlays begin by constructing a random graph, and then they either flood or take random walks across that graph in order to identify the data that is stored by overlay nodes. Without relying on a deterministic topology, an unstructured overlay demonstrates good global qualities such as connection (for dependability), low-diameter and constant-degree (for scalability), and so on. These properties are important for scaling.

Peers in such unstructured systems establish overlay connections to a (largely arbitrary) collection of other peers that they know, and shared objects can be placed at any node in the system. There are minimal limits placed on the building of the overlay and the placement of data in such systems. The random overlay structures and data distributions that are the result may provide high resilience to the degrees of transiency (also known as churn) that are found in peer populations. However, this results in clients being limited to nearly blind searches, which require either flooding or random walks to cover a large number of peers.

Structured protocols, also known as DHT-based protocols, on the other hand, reduce the cost of searches by constraining both the overlay structure and the placement of data. Data objects and nodes are each given a unique identifier or key, and queries are routed to the node responsible for keeping the object based on the searched object key (or a pointer to it). Even though the resulting overlay offers efficient support for exact-match queries (usually in $O(log(n))$ operations, this may come at a significant cost in terms of churn resilience, as well as the systems' capacity to exploit node heterogeneity and efficiently support complicated queries.

When compared to unstructured graphs, structured graphs are more expensive to maintain, and due to the limits that are imposed by the structure, it is more difficult to take advantage of heterogeneity in order to improve scalability.

Several examples of organized and unstructured overlay networks are presented here. Structured overlays can be distinguished from one another based on a number of different dimensions, including the maximum number of hops that can be taken to fulfill a routing request, the routing algorithm, the node degree in relation to the size of the overlay, the overlay geometry, and the lookup type. Some of the peer-to-peer systems that are built on structured overlays are Chord, Pastry, and Kademlia.

There are many solutions out there for an accurate choice of the OMP necessary in order to build the blockchain protocol. The maintenance approach regarding the actions needed to rearrange the overlay is often distinguishing the different OMPs like SCAMP or CYCLON, with the latter having a proactive approach instead of the reactive one implemented by SCAMP. Also, they may create different random graphs with different diameters and features that in a way or another impact the communication layer built upon it.

3.4.2 Communication layer

The second distributed block identified concerns the communication layer.

For both application messages and consensus-related messages, all the participants must be sure that their messages reach any other participant in the network. This is important for spreading the transactions received from the clients to all the other participants for validation reasons, as well for the agreement primitive that is built upon this block. In fact, if consensus-related messages are not delivered to the peers in charge of the consensus, they might not reach consensus at all, resulting in a halt of the blockchain and at a later date of the application using it.

An easy to implement protocol may be a gossip-based protocol, in which a peer sends the self-generated messages to all its neighbors, that are defined by the OMP. Also, the peer must handle the retransmission of the messages sent to it by the other peers. The are other options like Best-effort, Reliable, Probabilistic or Atomic Broadcast. It depends always on what the system must achieve and what are the its requirements.

3.4.3 Agreement Primitive

This block is where consensus takes place. The participants to the network will have to exchange messages in order to reach consensus, no matter what type of consensus they use, i.e. leader-election, committee-based voting. When discussing about consensus, there are many more types of messages that are exchanged, like checkpoint messages in case of PBFT or votes collection when the algorithm used is Tendermint or recovery mechanisms that may took place. Also regarding this block, there are many solutions that can be employed, raging from BFT based algorithms to FBA, each with its own characteristics. The decision is guided by the type of application that the blockchain must support, and this choice reflects on the properties that must hold underneath this layer.

A set of defined metrics will be described in section 3.7.

3.5 Churn Model

It allows to characterize the dynamics of the system and to describe the arrival and departure distribution of the processes from the system. Its importance is due to the fact that the intensity of the churn applied to the system may heavily impact the system's behavior.

Different types of churn models were identified and may be applied to this framework:

- *Continuous:* nodes leave and join the network in a constant manner and the churn lasts for the entire execution of the system.
- *Intermittent:* in which the system execution alternates periods with churn and other periods without churn.
- *Quiescent:* where the churn terminates at some point in time.
- *Trace:* the churn is predetermined and follows some specific guidelines.

Moreover, there may be a number of other churn models that can be implemented, for example based on the type of agreement selected. If XRP was selected, one may think to model the churn in a way to hit nodes that mostly appear in the UNLs present in the system, or in the case of PBFT churn may always need the primary node.

Each model is characterized by a rate parameter, that stands for the number of processes that leave and join the network in that moment in time. For example, a rate that has value 4 means that, in a certain moment of the execution, 4 nodes leave the network and 4 new participants join it. This aspect can certainly be changed and allow a network to increase or decrease in size during the lifetime of the system, thus creating yet another churn model.

3.6 Load Model

The model characterizing the transactions load is straightforward, and is represented by the distribution of the inter-arrival requests. Like in the case of churn, there are many different transactions load models that can be implemented. Apart from the more or less large number of requests and their creation rate, one could decide to load only specific parts of the network or may alternate periods with a high number of requests to periods with no activity.

Just to add further layers of difficulty, transactions and churn loads may be combined and observe how the overlay network and the consensus behave.

3.7 Metrics

In order to understand how the blockchain protocol is affected when churn is added to the system, there is the need to define some metrics that give insights about it. The metrics that have been set are as follows:

- *Blockchain length:* measures the length of the blockchain.
- *Average connectivity:* the average percentage of nodes that can be reached by any node in the system, at any time during the execution of the system.

- *Largest strong component:* the size of the largest strong component in the system, at any time during the execution of the system.
- *Number of strong components:* number of strong components in the system, at any time during the execution of the system.
- *Requests latency:* the time elapsed from the creation of the block to its addition to the blockchain.
- *Blocks creation speed:* time indicating how rapidly blocks are created.
- *OMP messages:* number of messages necessary to maintain the overlay network.

Many more metrics may be extracted that are specific to the kind of application and composition model adopted.

3.8 Implemented instance

As stated in the previous sections, there are many different options when coming to the choices for the composition model. For this work, the stack instance of components is represented as follows:

- *PBFT:* the first practical solution for the implementation of a consensus protocol in byzantine settings is provided by the PBFT algorithm. Due to its clarity and powerful mechanisms, it has been implemented as the agreement primitive.
- *Flooding-based Broadcast:* in this particular case, since the agreement primitive chosen was PBFT, then it is needed that in ideal conditions, a message sent by a node reaches any other node in the network, meaning that anyone participating in the network is able to see and decide upon the messages received from all the other participants. This kind of broadcast primitive have been chosen because the PBFT algorithm assumes a fully-connected network.
- *Scalable Membership Protocol (SCAMP):* has been selected as the Overlay Management Protocol in charge of creating and maintaining the overlay network. Therefore, thanks to SCAMP mechanisms, a random graph will be utilized. In section 3.8.1 this protocol will be deeply explained. A random graph is a graph in which properties such as the number of graph edges and connections between vertices them are determined in some random way. In a blockchain context, vertices are the nodes participating to the protocol, and edges represent the direct connections a node has, i.e., to whom they can send messages directly.

3.8.1 Scalable Membership Protocol (SCAMP)

SCAMP is a gossip-based protocol whose main distinguishing feature is that the view size is adaptive in relation to the a priori unknown size of the entire system. More specifically, view size in SCAMP is logarithmic of system size.

The goal of this protocol is to eliminate the need of keeping some centralized or global information like the full membership. This protocol provides each node with a partial view of the membership, and includes the following design requirements:

- *Scalability:* the size of the partial view maintained at each node should grow slowly with the group size.
- *Reliability:* the partial views at each node should be large enough to support gossip with reliability comparable to that of traditional schemes relying on full knowledge of group membership.
- *Decentralized operation:* the partial views should be updated as members join or leave the network while maintaining the above properties. The updates should only use local data. Even if no node knows the system size, the partial view sizes should scale to the correct value automatically.
- *Isolation recovery:* if nodes choose their gossip targets based on a partial view that remains unchanged for long periods of time, a mechanism for recovering isolation is required.

SCAMP Mechanisms In order for SCAMP to achieve its goals, the mechanisms used are listed below:

- *Data Structures:* each node keeps two lists: a PartialView of nodes to which it sends messages and an InView of nodes from which it receives messages, namely nodes with its node-id in their PartialViews.
- *Join algorithm:* new nodes join the overlay by sending a join request to any member, referred to as a contact. They begin with a PartialView that only includes their contact. When a node receives a new join request, it sends the new node-id to all PartialView members. It also makes c additional copies of the new join request (c is a design parameter that determines the proportion of failures tolerated) and sends them to nodes in its PartialView at random. When a node receives a forwarded join request, it integrates the new node with probability $p = 1/(1 + sizeofPartialView_n)$ if the subscription is not already present in its PartialView. If it decides not to keep the new node, it forwards the join request to a node randomly chosen from its PartialView. If a node i decides to keep the join request of node j, it places the id of node j in its PartialView. It also sends a message to node j telling it to keep the node-id of i in its InView.
- *Leave algorithm:* the leaving node orders the id's in its PartialView as $i(1), i(2), ..., i(l)$ and the id's in InView as $j(1), j(2), ..., j(l^{'})$. The leaving node will inform nodes $j(1), j(2), ..., j(l^{'}-c-1)$ to replace its id with $i(1), i(2), ..., i(l^{'}-c-1)$ respectively (wrapping around if $(l^{'}-c-1) > l$). It will inform nodes $j(l^{'}-c), ..., j(l^{'})$ to remove it from their lists without replacing it by any id.
- *Recovery from isolation:* when all nodes with the same identifier in their PartialViews fail or leave, the node becomes isolated. A heartbeat mechanism is used to connect such nodes. Each node sends heartbeat messages to the nodes in its PartialView on a regular basis. A node in its PartialView that has not received a heartbeat message in a long time re-joins via an arbitrary node.

3.9 Conclusion

Throughout this chapter the modular approach used to assess blockchain protocol performance has been described. There were described the three main distributed building blocks that are in common to each blockchain protocol, and how they are working in synergy, explaining how choices for one of them end up impacting the others. The identified problem, represented by churn, that can affect how blockchain protocols behave has been brought into the picture, and subsequently, other components like churn, transactions load and metrics composing the framework were detailed. Then, the implemented and tested composition model has been described.

As already stated, the one presented in this work is just one of the many possible combinations one can have regarding the composition model, therefore any type of Overlay Management Protocol can be selected, and consequentially any type of communication layer and agreement protocol can be built upon it. For example, one could choose to implement the CYCLON OMP, a reliable broadcast and a FBA algorithm for the consensus layer.

In the following chapter it will be described the technical details of the implementation developed in Objective Modular Network Testbed in C++ (OMNeT++) that allows to take a generic blockchain protocol, map it to the three distributed building blocks and using the interfaces available to evaluate how churn impacts the protocol's performance.

Chapter 4

Simulation Framework

4.1 Introduction

As seen in the previous chapter, a blockchain protocol has at its core different components that work together in order to achieve the purpose of the application built upon the blockchain. In this chapter it will be explained how the proposed framework was implemented in a simulation environment created with OMNeT++, and therefore how one can decompose the blockchain protocol and map its components to the objects implemented in the framework. In this way, the most disparate simulations can be carried out, since the components used can be transparently interchanged. For example, with very meager effort one could replace an unstructured overlay component with a structured one, by simply changing the environment parameters.

Before diving into the implementation details, a brief description will be done about the simulation modeling topics, and in particular the OMNeT++ and Overlay Network Simulation Framework (OverSim) characteristics will be illustrated.

4.2 Simulation modeling and analysis

This section provides an overview of the Discrete-Event Simulation model, as well as an open-source overlay and peer-to-peer network simulation framework. The simulation environment is based on the C++ programming language.

Analytical models might not always be applicable when trying to analyze real-world systems because of certain circumstances. This is due to the fact that systems in the real world can end up being too complex to study directly, so simulation is required instead. Within the context of a simulation, the model is subjected to a numerical analysis, and data are amassed in order to arrive at an estimate of the desired true characteristics of the model [9]. A collection of entities, such as people or machines, that act and interact together towards the accomplishment of some logical end is referred to as a system. One definition of a system is as follows: A collection of variables that are required to adequately describe a system at a specific point in time in relation to the goals of a study is what is meant when one refers to the *state of the system*.

There are two types of systems, discrete and continuous. An example of a continuous system is an airplane that is traveling through the air. On the other hand, a bank is an example of a discrete system due to the fact that the variables change at different instants at different points in time.

There are numerous approaches to research that can be used on a system. Altering the system physically and allowing it to function under the new conditions, after which one could observe the system's behavior, is an option if doing so is feasible and economical. Because running such an experiment would be too disruptive to the system, it is not always possible to accomplish this. Because of these factors, it is typically necessary to construct a physical or mathematical model as a representation of the system and study this model instead. This can be done for a number of different reasons. A mathematical model represents a system in terms of logical and quantitative relationships that can be manipulated and changed to see how the model reacts, and thus how the system would react — if the mathematical model is a valid one. A physical model represents a physical construct whose characteristics resemble the physical characteristics of the modeled system. A mathematical model represents a system in terms of logical and quantitative relationships that resemble the physical characteristics of the modeled system. After a mathematical model has been constructed, an analytical solution may be utilized in order to work with the model's relationships and quantities in order to obtain an exact behavior of the model and, consequently, answers to the questions that are of interest regarding the system. Because so many systems are so complicated, it is not always possible to find an analytical solution to the problem. In this scenario, the model needs to be investigated by means of simulation, which means that it needs to be exercised numerically using the relevant inputs to determine how those inputs influence the model's performance measures as an output.

Simulation models can be classified along three different dimensions:

- *Static vs. Dynamic.* A static simulation model is a representation of a system at a particular time. On the other hand, a dynamic simulation model represents a system as it evolves over time.
- *Deterministic vs Stochastic.* If the model does not contain random components, it is called deterministic. However, many systems must be modeled as having at least some random input components, and these give rise to stochastic simulation models.
- *Continuous vs. Discrete.* Loosely speaking, these models can be defined analogously to the way discrete and continuous systems are defined.

4.2.1 Discrete-Event Simulation

Discrete-event simulation is concerned with the modeling of a system as it evolves over time using a representation in which the state variables change instantly at different points in time. These are the points in time where an event happens, where an event is defined as an instantaneous occurrence that can change the state of the system.

Because of the dynamic nature of discrete-event simulation models, the value of the simulated time must be tracked as the simulation progresses. This value is called *simulation clock*. In general, there is no relationship between simulated time and the time required to run a computer simulation. There are two methods for advancing the simulation clock: *next-event time advance* and *fixed-increment time advance*, with the latter being a special case of the first. The simulation clock is reset to zero and the times of future events are determined using the next-event time-advance method. The simulation clock is then advanced to the time of occurrence of the most recent of these future events, and this process is repeated until either a pre-specified stopping condition is met or no more events are scheduled.

All discrete-event simulation models share a number of common components that are linked together logically. In particular:

- *System state:* The collection of state variables necessary to describe the system at a particular time.
- *Simulation clock:* The current value of the simulated time, stored as a variable.
- *Event list:* A timetable of forthcoming activities.
- *Statistical counters:* Statistics-related variables that are used to keep track on system performance.
- *Initialization routine:* A subprogram to initialize the simulation model at time 0
- *Timing routing:* A module that iteratively scans the event list, finds the next event, and sets the simulation clock to the time at which that event will happen.
- *Report generator:* A module that, once the simulation is complete, calculates the performance metrics of interest based on the statistics counters and writes a report detailing the results.
- *Main program:* A program that calls the timing routine to find out when the next event will be, and then calls the relevant event routine to make the necessary changes to the system. The main program may additionally keep track of simulation completion and call the report generator accordingly.

Regarding this body of work, the discrete-event simulation model was selected as the appropriate one to use. In specifically, a simulation library and framework built in C++ has been utilized, and it properly conforms to this kind of model.

4.2.2 OMNeT++

OMNeT++ is a modular object-oriented discrete-event simulation framework. It has a generic architecture, which allows it to be used in a variety of problem domains, such as protocol modeling, hardware architecture validation, modeling of wired/wireless or peer-to-peer communication networks, and evaluating performance aspects of complex software systems [5].

The OMNeT++ model is made up of simple modules that communicate via message passing. They can be combined into compound models with an unlimited number of hierarchy levels. Simple modules typically send messages through gates, but they can also be sent directly to their destination modules. Connection parameters such as propagation, delay, data rate, and bit error rate can all be set. A model also includes NED language topology descriptions, which characterize the module structure with parameters, gates, and so on, and message definitions, which allow you to define message types and add data fields to them. OMNeT++ simulations can be run through a variety of user interfaces. Graphical, animating user interfaces are ideal for demonstration and debugging, while command-line user interfaces are ideal for batch processing.

Countless simulation models and model frameworks have been written for it over the years by researchers from various fields, such as queuing, resource modeling, internet protocols, wireless networks, switched LANs, peer-to-peer networks, and mesh networks. OverSim stands out among them for the purposes of this work.

4.2.3 OverSim

OverSim is an open-source overlay and peer-to-peer network simulation framework built on the OMNeT++ simulation environment. OverSim includes models for both structured (such as Chord, Kademlia, and Pastry) and unstructured (such as GIA) peer-to-peer protocols [10].

OverSim simulates the exchange and processing of network messages by utilizing Discrete- Event Simulation (DES). The underlying network model is positioned at the lowest level of stack design. Three underlying network models are supported by OverSim: Simple, SingleHost, and INET. The most scalable underlay is the Simple. Using a global routing table, packets are sent directly from one overlay node to another in this model. Packets between overlay nodes are delayed by a constant period of time, or, in more realistic scenarios, by a delay calculated based on the distance between the nodes. As a result of the low overhead, all relevant influences of the underlying network can be simulated with a single simulation event, resulting in a high level of accuracy and the ability to simulate networks with a large number of nodes. The INET underlay model is derived from the OMNeT++ INET framework, which includes simulation models of all network layers beginning with the MAC layer. If necessary, it can be used to simulate entire backbone structures. To the overlay protocols, all of the underlying network models have a consistent UDP/IP interface. As a result, switching to a different underlying network model is completely transparent to the overlay network.

The overlay protocols represent the second level of the stack. Peer-to-peer networks are available in both structured and unstructured forms. Several functions have been integrated into the simulation framework to aid in the implementation of new overlay protocols:

- *Overlay message handling (RPC and statistical data).* It makes dealing with timeouts and packet retransmissions due to packet losses easier.
- *Generic lookup function.* As a result, for each overlay protocol, only a method to query the local routing table, which returns the closest node in the overlay topology, must be implemented.
- *Support for visualization of the overlay topology.* OMNeT++'s graphical user interface aids in the debugging of new overlay protocols by displaying transferred messages in detail. During run-time, it is also possible to monitor and even change overlay-specific routing tables and other internal states.
- *Bootstrapping support.* In every simulation, a generic module called Global Observer has a global view of the overlay network. It can perform a variety of user-defined functions, but for the time being, it is primarily used as a Bootstrap Oracle, providing the address of a random node already in the overlay network to nodes wishing to join the overlay. The module can also be used to collect global statistics.

The Common API is used for communication between the overlay and the application. Every overlay protocol that wants to use this API must provide the application with at least a Key-based Routing (KBR). The applications represent the third and final level. A wide range of different applications that rely on key-based routing can be evaluated with exchangeable overlays using the Common API design. In this work, OMNeT++ and OverSim were both used to implement the SCAMP as the second level of the stack, while the PBFT algorithm was implemented as the third element of the same stack.

4.3 Implementation

As described in the previous sections, OMNeT++ is highly modular, allowing the users to write simple modules that can be grouped together in order to create more complex modules. These compound modules will serve different purposes throughout the framework.

In this section will be described several of these compound models, that allow to compose the framework described in chapter 3. Moreover, it will be described how these modules can be used together, the way they need to interact and what constraints they must meet. Towards the end of the section, it will be outlined how a simulation object is created in this framework and what are its main characteristics.

4.3.1 Underlay

The OverSim framework provides three underlying network models, but for this framework the *Simple* one has been chosen, mainly because it is the most scalable. This component is totally transparent to the overlay, and some of its characteristics can be seen in figure 4.1.

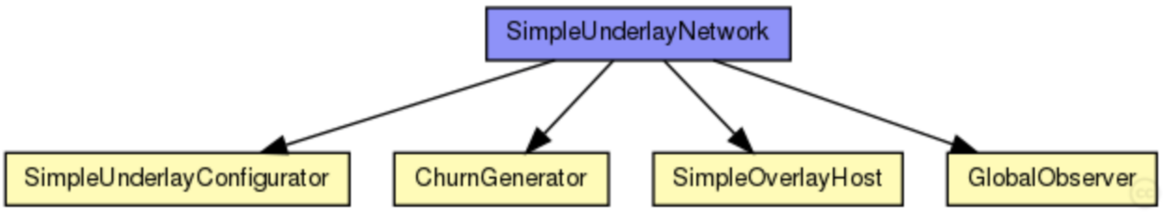


Figure 4.1: Simple Underlay Network modules.

Beside the *Configurator* and *Observer* modules, that are necessary for the configuration and observation of the underlay network, the other two modules abstract the churn model to be applied to the network and the type of hosts that represent the nodes belonging to the network. Given that the churn module will be explained in a subsequent section, in figure 4.2 are the details about the *SimpleOverlayHost* module.

From the figure 4.2 it can be observed that these compound modules abstract most of the network features, like TCP and UDP interfaces to the overlay protocol. In addition, it includes a bootstrap list that may help the node in its initialization phase and also three tiers, each one representing a simple or compound module offering additional functionalities to the host. Most of these modules make available methods that can be called in order to allow the interaction between them, that is through message passing. For example, one could decide to implement a module that represents the behavior of the overlay. This module is to be intended distributed, in the sense that

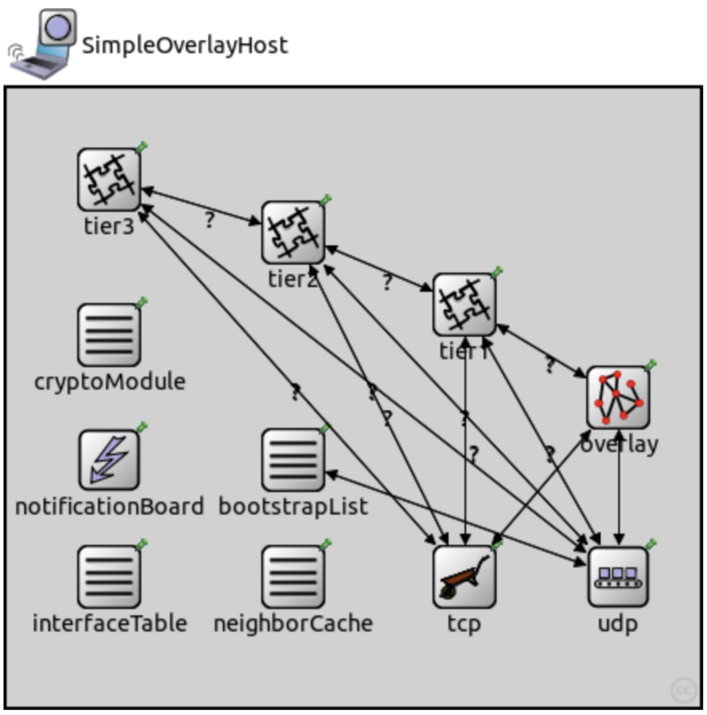


Figure 4.2: Simple Overlay Host modules.

it will run locally at each host in the network, but all of them with the same logic. Who designs it, may decide that the communication with other hosts has to use TCP connections instead of UDP, and it can be implemented accordingly. All the messages generated by the overlay logic will then travel towards the TCP or UDP module, that will handle the sending and delivery of the messages to the correct recipient, along with all the network-related parameters, like delay, error rate, etc.

Regarding the configuration of the underlay network, there are many parameters to set, the most important being:

- the list of churn generators to use (if necessary).
- eventual global functions that allow to take measurements of the system progress.
- the list of host types (that can be others than SimpleOverlayHost).
- the type of the overlay to be used by the hosts belonging to the network.
- the specification of any additional module that will map to tiers 1, 2 or 3.
- the probability for a node to leave gracefully.
- time between prekill and removal from the overlay.

This configuration will be completed once the implementation of all the other necessary modules will be terminated.

4.3.2 Overlay

Until this point of the framework description, no code had to be written. As explained in chapter 3, the first building block to take care of is the overlay network, in particular the OMP that is in charge of creating and maintaining it, guaranteeing the necessary properties to the blockchain protocol. In order to have more context, in this section the focus will be on the overlay square of the figure 4.2. This object is an overlay interface, that must be implemented by the class or classes defining the behavior of the OMP. The interface makes available 6 gates, 2 for the TCP module, 2 for the UDP module and 2 for tier 1 module (figure 4.3). In this way, by calling the correct methods, the overlay module can interact with all these 3 modules. It will be able to handle both application messages (originating from tier 1, 2 or 3), as well as OMP messages necessary for the creation and maintenance of the overlay network.

```
moduleinterface IOverlay
{
    parameters:
        @display("i=block/network2");

    gates:
        input udpIn;     // gate from the UDP layer
        input appIn;     // gate from the application
        input tcpIn;     // gate from the TCP layer
        output tcpOut;     // gate to the TCP layer
        output udpOut;     // gate to the UDP layer
        output appOut;     // gate to the application
}
```

Figure 4.3: IOverlay gates.

The first step in implementing the overlay block is to create a file (with .ned extension) written in NED language, typical of OMNeT++. This file will be provided as the value for the parameter *overlayType* when defining the underlay network configuration.

Inside this file will have to be defined the following components:

- an overlay modules section that extends the IOverlay interface mentioned above. Inside it, will have to be listed all the gates and submodules that will be used by it. Also, there must be done the connections mapping. In here, parameters functional to the overlay can be specified.
- a simple module description for each submodule specified in the overlay modules file. These modules can extend BaseOverlay class or can represent plain C++ classes. also in this case, the necessary parameters for the proper functioning of the overlay can be specified. For each module there must exist a .cc and a .h C++ implementation.
- imports of the IOverlay and BaseOverlay classes.
- package definition.

As an example, since in this work the overlay network has been created relying on the SCAMP OMP, the .ned file included the definitions seen in figures 4.4, 4.5 and 4.6.

From the description of the SCAMP mechanisms, here can be observed that the overlay module is making use of the views necessary to the algorithm implementation. As listed in the above code, there is defined the module that extends the IOverlay interface, inside which are defined the gates, the submodules necessary for its functioning, and the mapping of the connections with the ones provided from the interface. It can be noticed that the *inView* and *partialView* submodules refer to the same implementation, they differ just by the name and the default value for the parameter

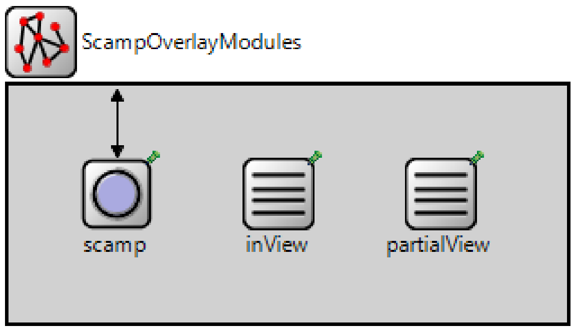


Figure 4.4: SCAMP components.

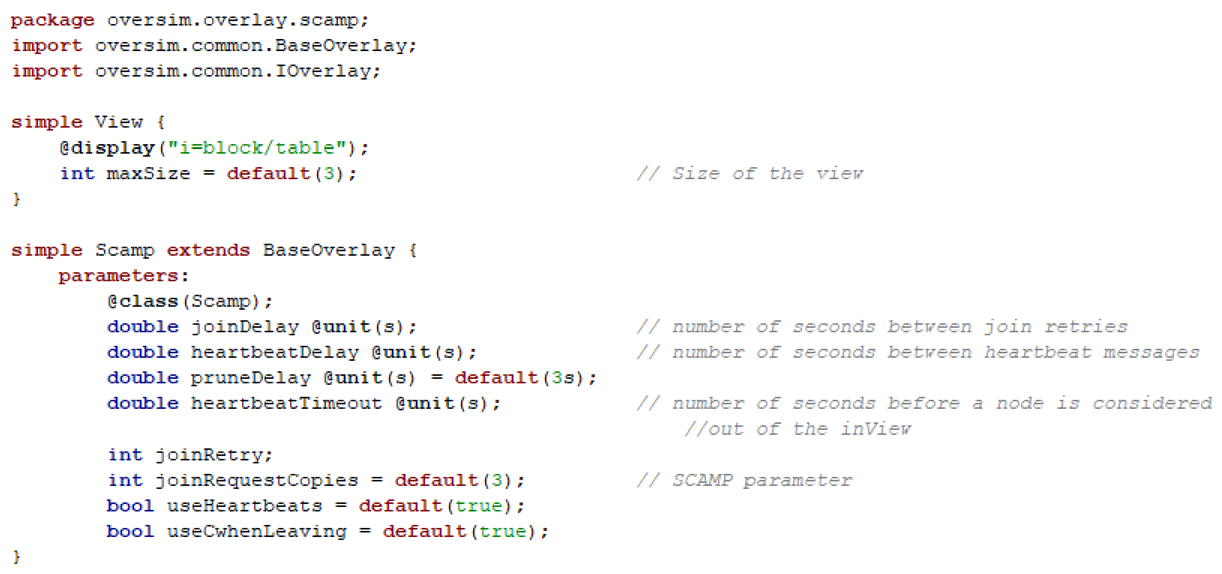
```
module ScampOverlayModules like IOverlay {
    parameters:
        @display("i=block/network2");

    gates:
        input udpIn;        // gate from the UDP layer
        output udpOut;      // gate to the UDP layer
        input tcpIn;        // gate from the TCP layer
        output tcpOut;      // gate to the TCP layer
        input appIn;        // gate from the application
        output appOut;      // gate to the application

    submodules:
        scamp: Scamp {
            parameters:
                @display("p=60,60");
        }
        inView: View {
            parameters:
                @display("p=150,60");
                maxSize = 0;
        }
        partialView: View {
            parameters:
                @display("p=240,60");
        }


    connections allowunconnected:
        udpIn --> scamp.udpIn;
        udpOut <-- scamp.udpOut;
        appIn --> scamp.appIn;
        appOut <-- scamp.appOut;
}
```

Figure 4.5: SCAMP definition.

maxSize. The simple Scamp module extends BaseOverlay class, and adds to it the parameters used in the Scamp.cc and Scamp.h C++ classes. The parameters can assume default values or the values can be provided through a .ini configuration file that will be explained later on.

Thanks to the extension of the BaseOverlay class, the Scamp.cc class will inherit a series of both defined and virtual classes to implement. Some of these are:

- *bindToPort:* Tells UDP we want to get all packets arriving on the given port.
- *route:* Routes message through overlay.



```
package oversim.overlay.scamp;
import oversim.common.BaseOverlay;
import oversim.common.IOverlay;

simple View {
    @display("i=block/table");
    int maxSize = default(3);                 // Size of the view
}

simple Scamp extends BaseOverlay {
    parameters:
        @class(Scamp);
        double joinDelay @unit(s);            // number of seconds between join retries
        double heartbeatDelay @unit(s);       // number of seconds between heartbeat messages
        double pruneDelay @unit(s) = default(3s);
        double heartbeatTimeout @unit(s);     // number of seconds before a node is considered
                                                 //out of the inView

        int joinRetry;
        int joinRequestCopies = default(3);   // SCAMP parameter
        bool useHeartbeats = default(true);
        bool useCwhenLeaving = default(true);
}
```

Figure 4.6: SCAMP simple module.

- *callDeliver:* Calls deliver function in application.
- *callUpdate:* Informs application about state changes of nodes or newly joined nodes.
- *join:* Join the overlay with a given nodeID.
- *virtual joinOverlay:* Join another overlay partition with the given node as bootstrap node.
- *sendMessageToUDP:* Sends message to underlay.
- *virtual isSiblingFor:* Query if a node is among the siblings for a given key.
- *virtual neighborSet:* Returns the set of neighbors.
- *virtual handleUDPMessage:* Processes messages from underlay.
- *virtual handleAppMessage:* Processes messages sent by the application layer.
- *virtual handleNodeGracefulLeaveNotification:* Method called when the node must exit from the overlay gracefully. Inside this method can be executed some actions prior to the exit, like sending a notification to the nodes in the neighbor set.
- *virtual recordOverlaySentStats:* Can be added some more overlay stats based on the type of overlay is being implemented.

There are many more methods provided by the BaseOverlay class, offering a wide range of functionalities, from constructors/destructors, initialization and finishing, Key-based Routing (KBR), message handling, icons and UI support, routing, to statistics helpers.

Additionally, the BaseOverlay class will include a wide range of variables that are used to simulate the behavior of the overlay:

- *overlayID:* identifies the overlay this node belongs to.
- *hopCountMax:* maximum number of overlay hops.

- *localPort:* UDP port for overlay messages.
- *isSiblingAttack:* if node is malicious, it tries a isSibling attack.

Furthermore, the class includes a wide collection of statistics about the number of messages and bytes sent or received, some of the most important being:

- *numDropped:* number of dropped messages by the overlay.
- *joinRetries:* number of join retries to the overlay.
- *bytesAppDataSent:* number of application data bytes sent.
- *bytesAppDataReceived:* number of application data bytes received.

Practically speaking, also the communication distributed building block is included in this implementation. Due to the fact that the overlay is handling the communication, the communication will happen based on how the overlay manages the interaction with its neighbors, or how the lookup is done in case of a structured overlay. When talking about SCAMP, an application message that has to be sent, will be sent to all the neighbors of the node in that moment, in particular to all the nodes in the partialView list that the node is storing. This mechanism is transparent to the application built above.

In conclusion, this class allows to extend its functionalities to the overlay chosen, to use defined methods and parameters in order to exploit its implementation and at the same time to extract useful information from the statistics fields saved on behalf of the class. Also, implementing the virtual methods provided, or new ones, further functionalities can be added.

4.3.3 Application

The third building block identified in chapter 3 refers to the agreement primitive, which can be referred as to the application component in the OMNeT++ framework. At this level, one can implement any application along with its features, that will interact through the overlay with all the other nodes in the system. For example, in this work the PBFT algorithm was implemented at the tier 1 block (figure 4.2). The entire algorithm logic takes place in this block, and the communication it needs is in charge of the overlay component described previously.

With regards to the application components that have to be implemented in the OMNeT++ framework, the approach is similar to the one took for the overlay. There must be defined a .ned file that will include the same kind of modules used for the overlay, with some subtle differences.

For the PBFT algorithm implemented here, figure 4.7 illustrates the composition of the modules that are essential for proper operation. In this case, additional simple modules were used for the correct PBFT implementation, like Blockchain and ReplicaState.

In figure 4.8 there were defined the submodules indicated above. These submodules main task is to store information about the local replicas' blockchain and its state.

The submodules employed by the algorithm are defined in the same file, and can have all the needed parameters also initialized by default. In figure 4.9 the f faulty replicas number allowed by the PBFT is initialized to 1 for the replicaState submodule, as well as other parameters like *blockCapacity* of *checkpointPeriod* required by the algorithm are present.

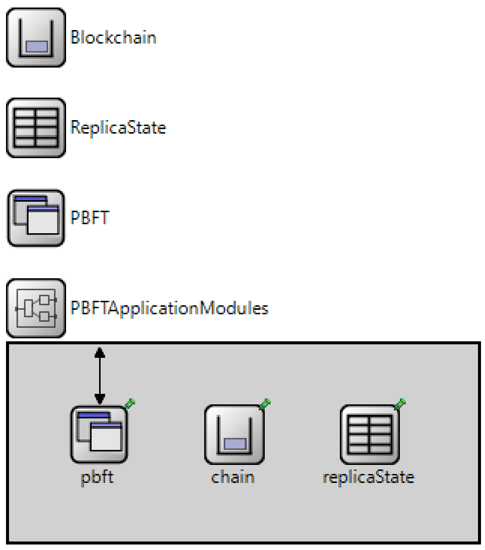


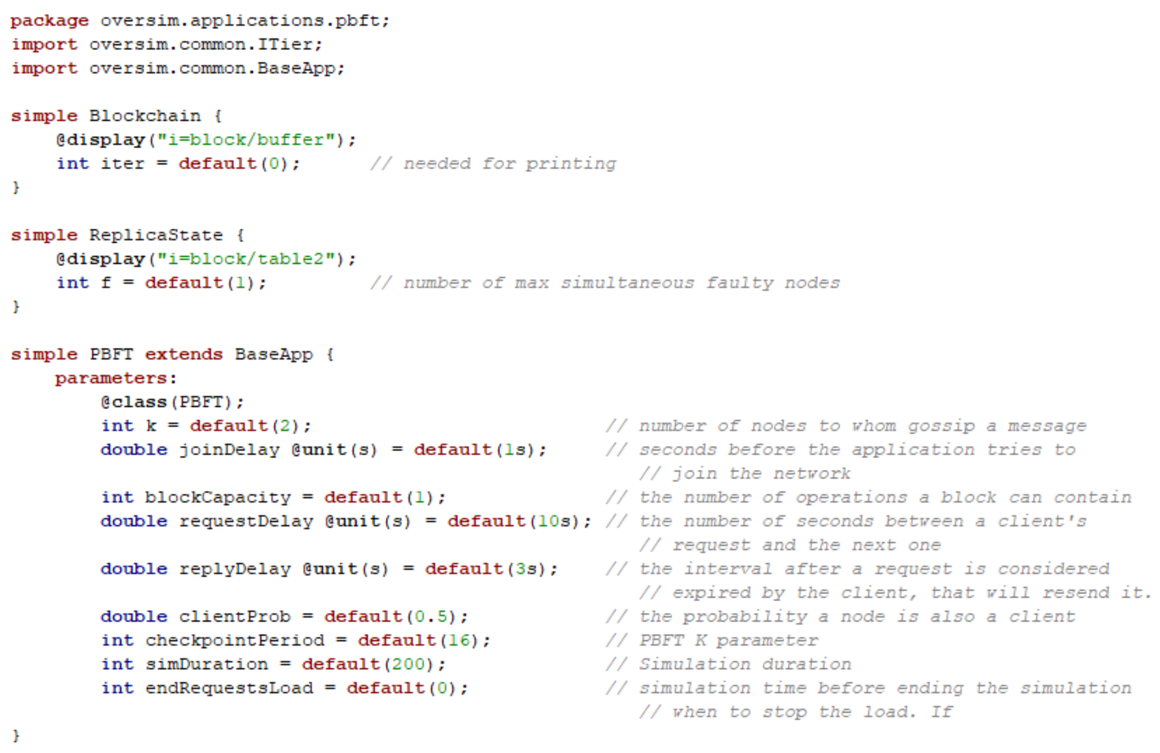
Figure 4.7: PBFT implementation components.

```
module PBFTApplicationModules like ITier {
    gates:
        input udpIn;              // gate from the UDP layer
        output udpOut;            // gate to the UDP layer
        input from_lowerTier;     // gate from the lower tier
        input from_upperTier;     // gate from the upper tier
        output to_lowerTier;      // gate to the lower tier
        output to_upperTier;      // gate to the upper tier
        input trace_in;           // gate for trace file commands
        input tcpIn;              // gate from the TCP layer
        output tcpOut;            // gate to the TCP layer

    submodules:
        pbft: PBFT{
            parameters:
                @display("p=60,60");
        }
        chain: Blockchain{
            parameters:
                @display("p=150,60");
        }
        replicaState: ReplicaState{
            parameters:
                @display("p=240,60");
        }

    connections allowunconnected:
        from_lowerTier --> pbft.from_lowerTier;
        to_lowerTier <-- pbft.to_lowerTier;
        udpIn --> pbft.udpIn;
        udpOut <-- pbft.udpOut;
        tcpIn --> pbft.tcpIn;
        tcpOut <-- pbft.tcpOut;
        trace_in --> pbft.trace_in;
}
```

Figure 4.8: PBFT modules definition.

Unlike the overlay implementation, where many different interface methods have to be implemented in order to run the code correctly, in this case there is much more freedom about how the functionalities can be implemented. The concepts are the same as before, meaning that the application one has to write will have to extend the BaseApp class.

The BaseApp class is the base for applications (Tier 1-3) that use overlay functionality. This



```
package oversim.applications.pbft;
import oversim.common.ITier;
import oversim.common.BaseApp;

simple Blockchain {
    @display("i=block/buffer");
    int iter = default(0);      // needed for printing
}

simple ReplicaState {
    @display("i=block/table2");
    int f = default(1);         // number of max simultaneous faulty nodes
}

simple PBFT extends BaseApp {
    parameters:
        @class(PBFT);
        int k = default(2);                          // number of nodes to whom gossip a message
        double joinDelay @unit(s) = default(1s);     // seconds before the application tries to
                                                        // join the network
        int blockCapacity = default(1);              // the number of operations a block can contain
        double requestDelay @unit(s) = default(10s); // the number of seconds between a client's
                                                        // request and the next one
        double replyDelay @unit(s) = default(3s);    // the interval after a request is considered
                                                        // expired by the client, that will resend it.
        double clientProb = default(0.5);            // the probability a node is also a client
        int checkpointPeriod = default(16);          // PBFT K parameter
        int simDuration = default(200);              // Simulation duration
        int endRequestsLoad = default(0);            // simulation time before ending the simulation
                                                        // when to stop the load. If
}
```

Figure 4.9: PBFT simple-modules definition.

class provides a set of methods and statistics useful for measuring the application performance. Among the most important methods, there are:

- *virtual initializeApp:* one of the first methods called when the application starts. In here will have to be initialized all the variables needed, and also eventual friend modules necessary to the correct behavior.
- *virtual handleUDPMessage:* method to handle messages that come directly from the UDP gate.
- *sendMessageToUDP:* sends a packet over UDP.
- *virtual handleLowerMessage:* method to handle messages coming from the overlay.
- *virtual finishApp:* collects statistical data derived from the application.
- *virtual update:* informs application about new or lost neighbors and own nodeID.

Also, the class provides some out-of-the-box statistics about the application internal data, like number of packets/bytes sent/received by the overlay, and number of packets/bytes sent/received from the UDP.

4.3.4 Churn

In the real world networks are not static, but are subject to a progressive refreshment of the peers participating in the system. Such phenomenon is known as churn, and has great importance when talking about peer-to-peer networks and blockchain protocols. Churn must be properly analyzed when designing overlay network and agreement primitives built upon them. In relation to this work, one of the objective is to deeply understand how churn impacts the performance of a blockchain protocol, and how the latter reacts because of consequences of churn on the communication layer and the overlay network.

The OverSim framework allows, by design, to have churn implementations throughout the stack used. In fact, when configuring the UnderlayNetwork, one can specify the type of churn to apply to the system. OverSim incorporates different classes with this purpose, like:

- *NoChurn:* No churn generating class (only bootstraps a networks)
- *RandomChurn:* Class that creates random churn, based on input parameters like creationProbability, removalProbability, churnTimer, targetMean.
- *ParetoChurn:* Churn based on shifted Pareto distribution.
- *TraceChurn:* Parse a trace file and schedule node joins/leaves according to trace data.

For the churn models listed in section 3.5, there were implemented two different classes:

- *Continuous:* A type of churn that each time is triggered it kills *rate* nodes and spawns as many. The rate parameter is an input to the class. This kind of churn is called continuous because the triggers happen at constant time, that is also an input to the class. From the implementation point of view this is a simple module that has access to the GlobalNodesList, and periodically selects rate processes to leave the network, but also creating as many new nodes that will join it.
- *Intermittent:* The intermittent churn is similar to the continuous one, with the difference that the interval between periods of churn are different. In fact, the continuous one employs 100% of the time to churn nodes, instead the intermittent type can employ different loads of churn, like 5%, 10%, 25%, 50% and 75%. This was done in order to allow the system to have less churn load.

As stated before, these classes can be used interchangeably by an UnderlayNetwork configuration, just by switching them from the .ini file.

4.3.5 Measurements

In order to measure the output of interest deriving from the system, the classes described have incorporated different mechanisms. As described earlier, the classes extended by the implementations already provide useful statistics that store many different values, like number of bytes sent or received, number of packets, etc., and also the churn class traces the number of killed and joined processes during the lifetime of the system. Beside these measurements, there is the need for some more complex values to pull from the system, like for example the average connectivity of the network, or number of leaves in the network, as well as the number of strong components belonging to the network. Since these parameters cannot be computed inside the nodes, classes may be created separately that have access to the required structures like GlobalNodesList, or even to some methods provided by the overlay or application classes. Therefore, these classes may access system-wide data useful for more complex computations like the ones listed above. The output will be saved in *.vec* and *.sca* files at the and of the simulations.

In order to better understand the blockchain protocol's performance when continuous or intermittent churn is applied, there was implemented a class that allows to measure all metrics listed in section 3.7. This class is represented by a simple module that has access to GlobalNodesList and GlobalStatistics. In this way, each time it is triggered, it is able to reconstruct the network topology based on the inView and partialView data structures of the nodes implementing the SCAMP OMP. Consequently, it is able to compute data useful to measure the system's performance, or even write to external files for post-processing purposes.

This class, in order to be effectively used by the simulation framework, has to be an input to the globalFunctions parameter of the GlobalObserver component of the UnderlayNetwork module.

4.3.6 Simulation composition

The OMNeT++ framework allows a very high configurable way of running the simulations, both from the CLI and GUI. Using the GUI has the advantage of visualizing the interaction between components, consult the body of the messages being passed between them and even modify parameters at runtime. Instead, the CLI allows to automate the simulations through a configuration file, and this implies that simulations complete in a faster way, since the GUI elements are not loaded at runtime. This is the preferred option when large simulations must be run.

In order to run the simulations a configuration file with extension .ini has to be created. Inside the file will be placed all the configurations necessary to the simulation to run, like the type of UnderlayNetwork, the overlay and the application associated. Beside these, there must be configured a number of other parameters:

- *result-dir:* The directory where output files like .sca and .vec must be saved.
- *underlayConfigurator.churnGeneratorTypes:* The types of churn generators.
- *targetOverlayTerminalNum:* Number of nodes to be created inside the system.
- *globalObserver.globalFunctions[0].functionType:* The module that is in charge of the measurements.
- *globalObserver.globalFunctions[*].function.connectivityProbeInterval:* The interval of time between triggers of the measurement module.
- *seed-set:* A seed useful for the initialization of the modules that adds some randomness in the system. In this way, simulations are different between them, even having the same input parameters.

These listed parameters are used the most throughout any simulation, but there are many more parameters specific to the application and overlay modules. These parameters will be described in the following chapter.

The simulations carried out with OMNeT++ are divided into three parts.

- *Bootstrapping:* Representing the start-up of the simulation. In this period all the components part of the simulation are initialized using a bottom-up approach. In this particular case, the application layer of the nodes will be the last one to get initialized. The period ends when the targetOverlayTerminalNum number of nodes is hit. Also, the overlay network is created following the mechanisms of the overlay chosen at configuration time.

- *Churning:* Immediately after, the churning period starts. Churn load is applied based on the churn class chosen and the parameters it received. Since the application receives the notification from the overlay that the overlay is ready, they can start the normal application tasks, like issuing transactions throughout the network. Since this work is using an agreement primitive as the application layer, the nodes will start trying to reach consensus on the transactions that arrive from the nodes that are also clients. At the same time, it becomes clear that since churn is applied, the OMP is also working in order maintain the overlay connected, while the application above runs normally. This period duration is also specified in the configuration file.
- *Stability:* After the churning period ends, nodes will still try to run their tasks, with the difference that the churn load is not applied anymore. For example, the client nodes will still issue transactions to the network.

4.4 Conclusion

In this chapter the simulation framework used has been illustrated, together with the main components that must be configured and implemented in order to carry on the desired simulations.

Chapter 5

Analysis

5.1 Introduction

Leveraging the simulation framework described in the previous chapter, a set of simulations were run in order to understand how different levels and types of churn may affect the blockchain protocol. In this particular case, as already mentioned, the SCAMP OMP has been chosen in order to create and maintain a random graph overlay network supporting the blockchain. Thanks to its mechanisms, this kind of protocol tries to adjust the network when nodes leave the network, gracefully or not. It remains to understand how well the protocol reacts to the churn.

Regarding the other two components, PBFT and a flooding-based broadcast primitive have been chosen. For the communication layer, this choice has been dictated because the consensus algorithm assumes a fully connected network. Therefore, each node receiving a message will send it to all its other neighbors.

Throughout this chapter it will be illustrated how these simulations were run, the inputs they got and finally the results emerged.

5.2 Simulation

As defined earlier, each simulation configuration takes place into a .ini file. In this case, all the simulations have an own configuration section, that is unique inside the file. Each simulation has been run from the command line, providing in input the name of the configuration section related to the simulation.

Each simulation has been run 5 times, and each one of them has a different input seed. This is specified with the parameter seed-id, as in the following snippet:

```
**.chain.iter = ${I=0..4 step 1}
seed-set = ${I}
```

Here, since the iter field of the chain object has to assume the integer values from 0 to 4, also does the seed-set, which assumes the value of I. In this way, each simulation has been executed 5 times.

After the execution of the simulations, several Python scripts were developed in order to aggregate, post-process and plot the data written to the external files, like .sca and .vec already mentioned.

This was done since some more complex computations were needed, and for performance reasons it was better to offload/postpone them after the execution was finished.

5.2.1 Input parameters

In this section will be illustrated the input parameters contained in the .ini configuration file provided to the simulations, divided by components.

Underlay

An example configuration starts with the definition of the configuration name in square brackets. Then, a description of the simulation can be added. Following, the overlayType must be specified, the tier1 application, the target number of overlay nodes that the overlay has to reach, the probability for a node to exit the overlay in a graceful way along with the delay between the exit decision and the actual actions took in order to exit, and finally the churn generator type.

```
[Config propbaseline-50-intchurn25-4]
description = Simulation with network size=50, 25% of churn and churn rate=4.
**.overlayType = "oversim.overlay.scamp.ScampOverlayModules"
**.tier1Type = "oversim.applications.pbft.PBFTApplicationModules"
**.targetOverlayTerminalNum = 50
*.underlayConfigurator.gracefulLeaveDelay = 0s
*.underlayConfigurator.gracefulLeaveProbability = 1
*.underlayConfigurator.churnGeneratorTypes="oversim.common.IntermittentChurn"
```

Regarding the underlay, the churn and targetOverlayTerminalNum parameters may change, but the other parameters are the same for each simulation instance.

In the above example, 50 is the number of nodes that will compose the overlay network, the churn type is the intermittent one described in section 3.5, nodes always exit immediately from the overlay in always gracefully (i.e. notifying the others about they are leaving) and SCAMP and PBFT have been chosen for the overlay and application layers.

Metrics

Regarding the way statistics are gathered during the simulations, it was defined a global function class that was in charge of the task. It is specified to the simulation object GlobalObserver, with the following parameters, where the connectivityProbeInterval is the trigger interval, i.e., each 2 seconds the methods inside the class are triggered for taking measurements. These parameters are the same for each simulation instance.

```
**.numGlobalFunctions = 1
**.globalFunctions[0].functionType="oversim.overlay.scamp.ScampConnectivityModules"
**.globalFunctions[*].function.connectivityProbeInterval = 2s
```

Overlay

Then, the parameters regarding the overlay are specified, like the joinDelay, heartbeatDelay and joinRetry among these. They are all parameters necessary for SCAMP to work. An instance may specify to the nodes that the joinDelay is 1s, and that after 10s no heartbeat messages are heard from a neighbor, that neighbor will be considered offline, and therefore pulled out from the inView of the node. Also, the max partialView size is 6. The last parameter, dag_id is used as the name of the file created by the overlay to store data about inView and partialView structures.

```
**.overlay*.scamp.joinDelay = 1s
**.overlay*.scamp.heartbeatDelay = 5s
**.overlay*.scamp.heartbeatTimeout = 10s
**.overlay*.scamp.joinRetry = 3
**.overlay*.scamp.joinRequestCopies = 3
**.partialView.maxSize = 6
**.dag_id = ${I}
```

In this case, all the parameters except maxSize are system-wide. maxSize may change based on the targetOverlayTerminalNum specified in 4.3.6.

PBFT

After that, there are few more factors to consider regarding the PBFT tier1 application. These are parameters used by the blockchain layer to ensure it works properly. Following are some important parameters:

```
**.pbft.blockCapacity = 10
**.pbft.k = 6
**.pbft.clientProb = 1
**.pbft.checkpointPeriod = 8s
**.pbft.requestDelay = 5s
**.pbft.replyDelay = 10s
```

For these simulations a block capacity of 10 transactions was used, the frequency of issuing requests to the network was set to 5s, and the max delay to have an answer was set to 10. Also, each 8s the checkpoint PBFT procedures are run. Note that parameters like k (number of nodes to whom gossip a message) may change, based on the number of nodes belonging to the overlay. Finally, the clientProb indicates the probability a node has to be also a client (beside being a replica). Being a client allows a node also to issue transaction, with the delay set to 5s in this case. Parameters like k and clientProb may differ from one simulation to another, while the others are system-wide.

Churn Generator

Through the following parameters one can model the churn that will be applied to the simulations. For example, there can be set the number of leavers and joiners to the network, in this case set to 4, but also one can simulate that throughout the simulation, the overlay may decrease or increase in size. The churnInterval stays for the delay between one churn moment and another. When churn must happen, a churnRate number of processes must leave the network, and the same quantity of nodes are spawned and must join it. One could also set the percentage of nodes that must be permanent into the overlay, in this way those nodes will never be asked to leave the network.

The last parameter stands for the number of seconds between churn periods, used mostly for the intermittent churn types. This parameter increases when the proportion of applied churn to the simulation decreases. If the proportion has to be 100%, then this value must be set to 10s, reducing the churn to the continuous one.

```
*.churnGenerator*.leavers = 40
*.churnGenerator*.joiners = 40
*.churnGenerator*.churnInterval = 10s
*.churnGenerator*.churnRate = 4
*.churnGenerator*.permanentNodes = 10
**.intervalBetweenChurnPeriods = 87s
```

Since the main work of this thesis was to understand the churn impact against a blockchain protocol, these parameters differ for each simulation, thus applying different churn loads to different overlay settings.

Below are some examples of applied churn. In figure 5.1 is shown how the number of churn processes increases over time when a continuous type of churn is applied, with different rates, in this case 2, 4 and 8. As already stated, from the parameters specified in input, it comes out that the bootstrapping period of this simulation lasts 50 seconds, because there are 50 nodes to be added to the overlay initially, each one with a delay of 1s. Therefore, the churn load starts immediately after the initialization periods ends, that is around 50s. During the subsequent period, nodes join and leave the network, until the churn period ends, after 400s since it's start. From that moment on, no more churn is applied, end the simulation ends after other 50s.

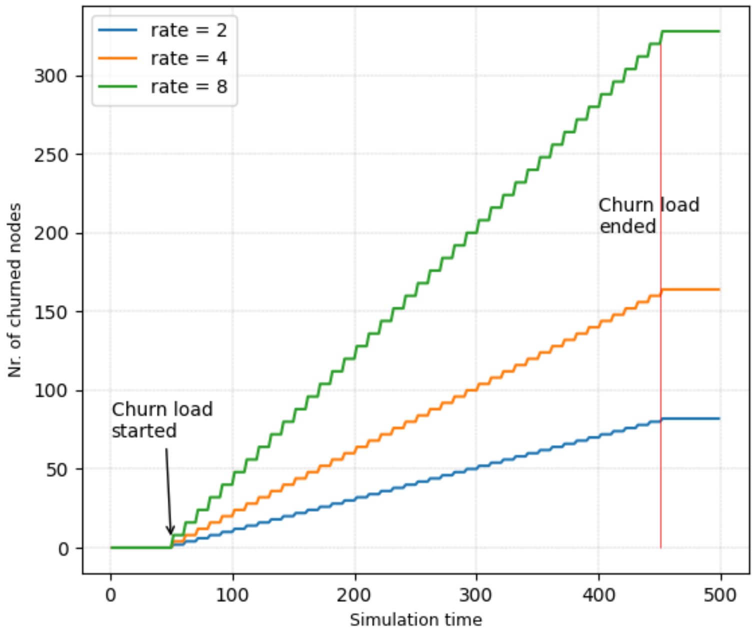


Figure 5.1: Continuous churn.

Network size is 50.

In figure 5.2 the other type of applied churn is shown. In this case, the intermittent churn, compared to the continuous one, is lighter in terms of load. In this particular example, periods of churn are alternate with periods of stability, therefore the seconds between these periods may vary. It can be observed that when the load is under 25%, the entire churn does not replaces the nodes that were present when the initial overlay was created, i.e., is less then 50.

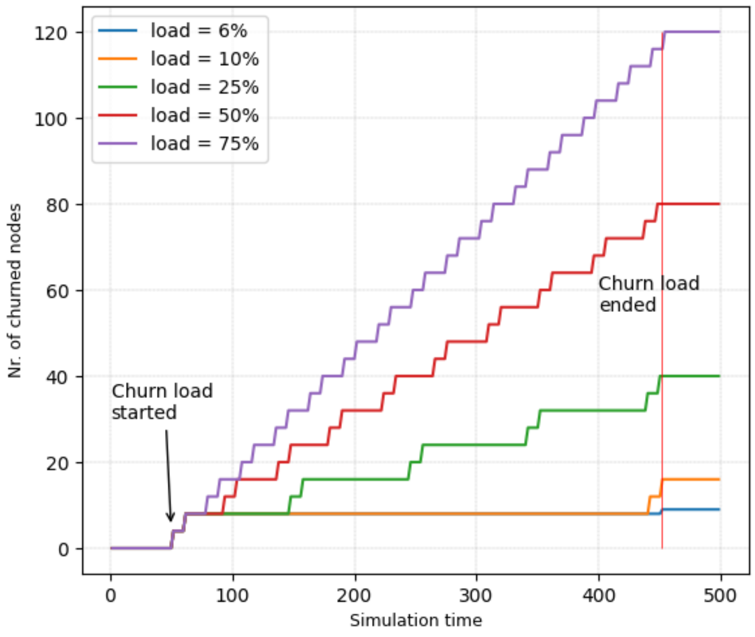


Figure 5.2: Intermittent churn.

Specified churn rate = 4.

The same loads of churn are applied also to simulations with other overlay dimensions, like 25, 75 or 100, and the number of interleaved processes is the same. For example, in a simulation with 100 nodes, the churn period will start after 100s, it will be stopped at $t = 500$ and the simulation will end after 550s. Basically, the churn period is shifted backward or forward.

In order to have a behavior with the system being stable to compare with churn-enabled simulations, also simulations without periods of churn, i.e., with a long period of stability in which churn was not applied, were run. These no-churn simulations have the same structure and share the same amount of time.

5.3 Results

In the following section there will be illustrated and discussed the results deriving from the simulations. The connecting theme regards the application of the above described churn to the different components stack of the simulations, therefore the results will be analyzed in order to understand the influence that churn has on those components from a variety of perspectives.

5.3.1 Connectivity

In figure 5.3 is shown the percentage value of average connectivity progress during the simulation. It can be observed that as soon as churn starts to be applied, the connectivity of the overlay decreases. In fact, when a node leaves the network, according to the SCAMP algorithm, it must notify his departure to its neighbors. Nevertheless, this does not prevent the overlay to decrease its average connectivity, since there are many nodes that leave the network in a small period of time. Regarding the churn load, one can see without any difficulty that when the churn rate is lower, for example 2, the overlay network is initially more resistant, whereas a churn rate 8 highly degrades the overlay connectivity. Because the connectivity measurements are taken exactly when the leaving nodes are kicked out of the system, the bottom spikes are caused by this. Immediately after the new nodes join, the overlay network is slightly reconnecting itself due to the fact that they are making join requests and adding nodes in their partialView and inView structures.

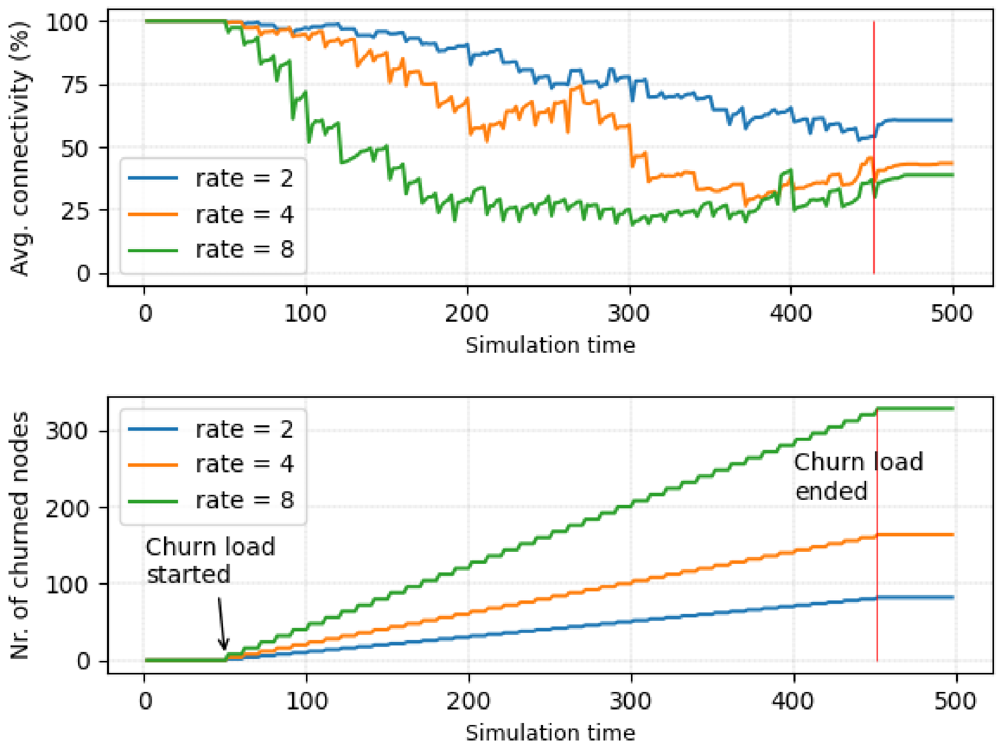


Figure 5.3: Connectivity with continuous churn.

Average connectivity trend for a 50-nodes network when applying continuous churn.

In figure 5.4 the impact of intermittent churn is compared against the continuous one. It becomes clear that decreasing the churn load, the overlay network stays more connected, and when the swapped nodes by the churn are less than the network size, the average connectivity stays above the 75%. Moreover, since the churn periods are rarer, the SCAMP lease mechanism has more time to recovery the overlay.

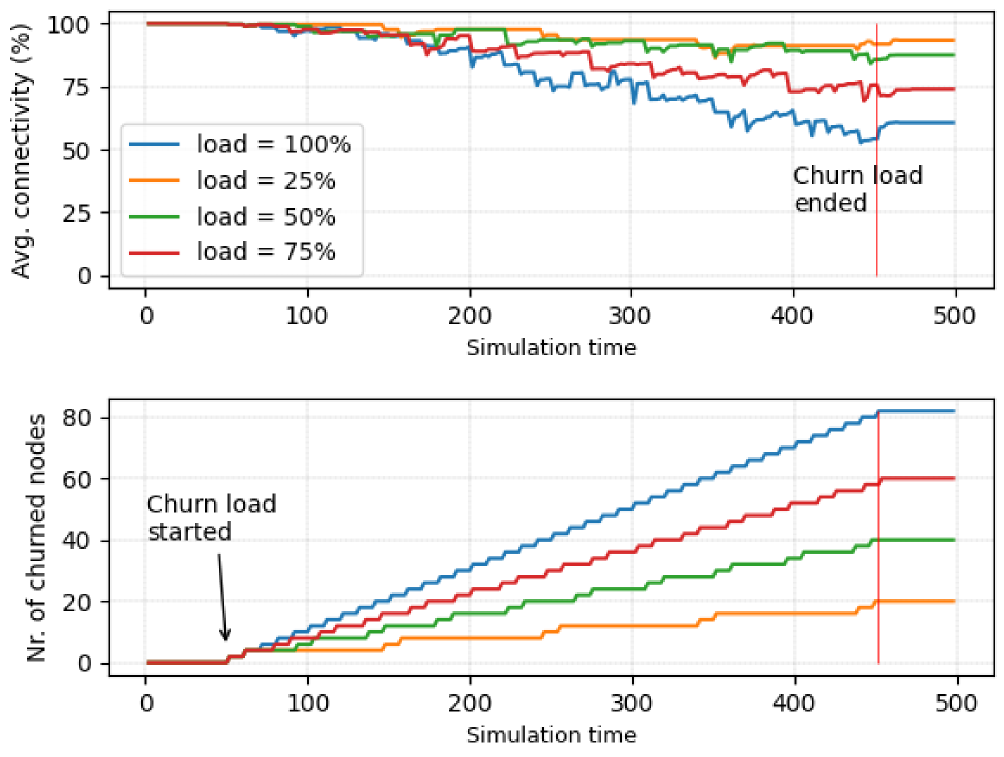


Figure 5.4: Intermittent churn connectivity vs. continuous churn.

Average connectivity trend for a 50-nodes network when applying both continuous and intermittent churn with rate = 2.

From the comparison done in figure 5.5, it emerges that also the network size plays a crucial role in how the overlay behaves when churn is applied. In fact, a bigger network is more resistant to churn than a smaller network, in this case 25. The same continuous churn seems to impact both overlays in the same way in the end, but when the churn load decreases, impacts are lighter on the larger network.

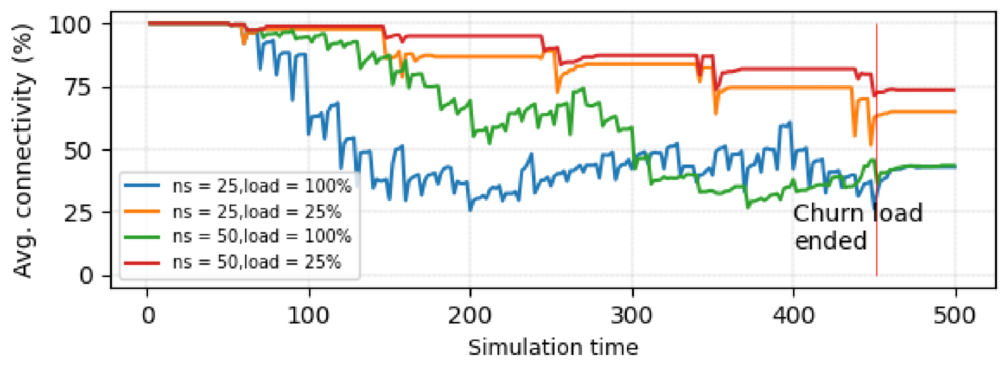


Figure 5.5: Connectivity 25-nodes network vs 50-nodes.

Average connectivity trend for a 50-nodes network when applying both continuous and intermittent churn with rate = 2.

From these preliminary results it emerges that both type and intensity of churn have major impacts on the overlay and the OMP selected. Furthermore, the network size is also important, since the network dimension can help the OMP lease mechanism.

The following section will delve deeper into the overlay-related measurements.

5.3.2 Strong components

SCAMP allows to create a random graph for the overlay network supporting the blockchain protocol. In the mathematical theory of directed graphs, a graph is said to be strongly connected if every vertex is reachable from every other vertex, in this case vertexes are nodes of the network, and their interconnections are modeled by the inView and partialView structures maintained by the SCAMP protocol. It is interesting to analyze therefore the number of the strong components belonging to the network while churn is applied, because their number and size have a considerable impact on the components stack considered in this instance.

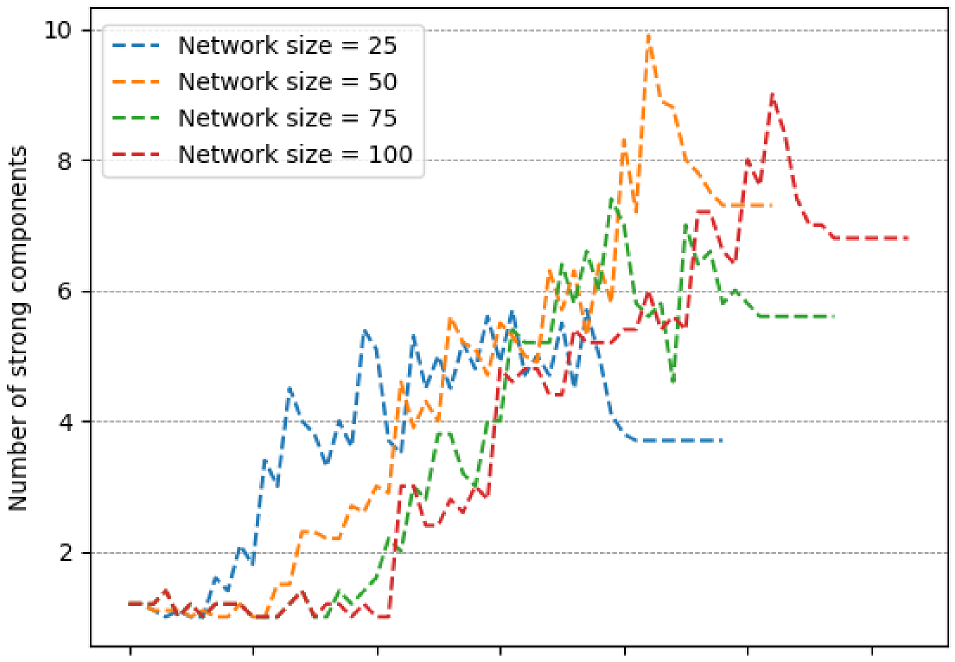


Figure 5.6: Strong components trend over time.

The progress of the strong components number when churn = 4.

In this particular instance that includes PBFT, it is crucial to understand the behavior of the network from the strong components point of view. The agreement algorithm assumes that when a node broadcasts a message, all the correct nodes receive it. Therefore, it is assumed that each message sent by a participant to the consensus gets received by all the other participants, and a BFT-based solution like PBFT requires that at least $3f + 1$ nodes agree on the next block to attach to the local instance of the blockchain. It is mandatory the overlay capability of sending and receiving messages sent by the nodes during the consensus, otherwise consensus cannot be reached. The primary node, since is the one that has to collect transactions, create the blocks that will be candidate as the next ones to append, broadcast this new block to the other participants and marking the 5 different phases planned by PBFT, plays a crucial role for the entire blockchain protocol. Also, it is important to understand how the strong component in which it belongs evolves over time, in size terms.

In figure 5.6 it can be observed that larger networks tend to have many strong components then smaller networks. In this case, the number of strong components was plotted since the simulation has started. In figure 5.7 it is shown how the size of the primary's largest strong component varies over time. It is interesting to notice that when the size of the networks is less than 50, at the end of the simulations the strong components that includes the primary is less than the required quorum size necessary for the PBFT algorithm to continue. This implies that the blockchain's progression has come to an end, since the primary is not able anymore to gossip new blocks and new consensus-related messages to all the required participants. PBFT, in fact, prefers safety over liveness. The same type of churn applied to the larger networks also affects the strong component of the primary, but in a lighter way that does not compromise the good output of the consensus built on top of the overlay.

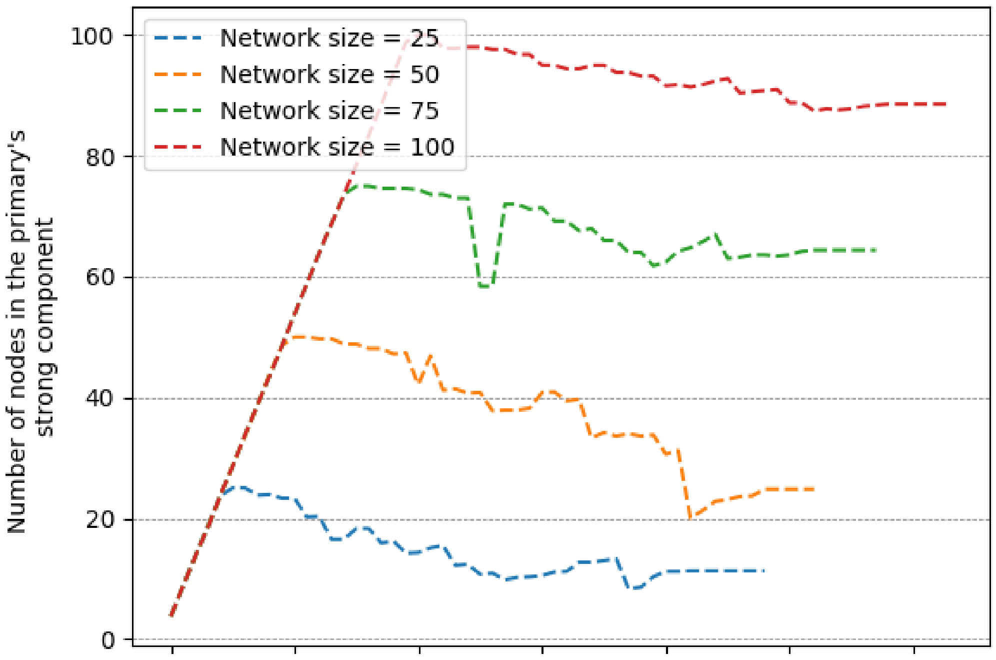


Figure 5.7: Primary's strong component size.

Size of the primary's strong component when churn = 4 is applied to the system.

Another interesting aspect to dig is about the number of times a node joins and leaves the main strong component. Here, when a node leaves the main strong component, it means that as a consequence of other churned nodes the node gets disconnected from that component. Therefore, for some time it will not be able anymore to send and receive messages to and from the same participants as before the disconnection. A join, on the other site, is when thanks to other nodes, freshly added to the system, contribute to the reconnection of the overlay network. Thus, counting how many times nodes leave and join the main strong component can provide useful insights of how churn impacts the blockchain protocol. In figure 5.8 it emerges that when a higher churn impact is applied, the nodes are part of the largest strong component for less time. Also, the time they employ into the system is not much. These two aspect are not helping the SCAMP protocol to keep connected the network, and in fact it degrades quickly, and the largest strong component size will constantly decrease, like it can be observed in figure 5.9 for the two most intensive churn loads of the example. This is also reflected in the figure 5.10, where a more aggressive churn forces the nodes to join and leave the main strong component many times, due to the fact that the interleaving of processes significantly influences the overlay network, thus partitioning it. On the other side, a lightweight churn allows nodes to stay connected for a grater period of time, the overlay algorithm has not additional work to do, and so the nodes can participate into consensus in order to progress the blockchain.

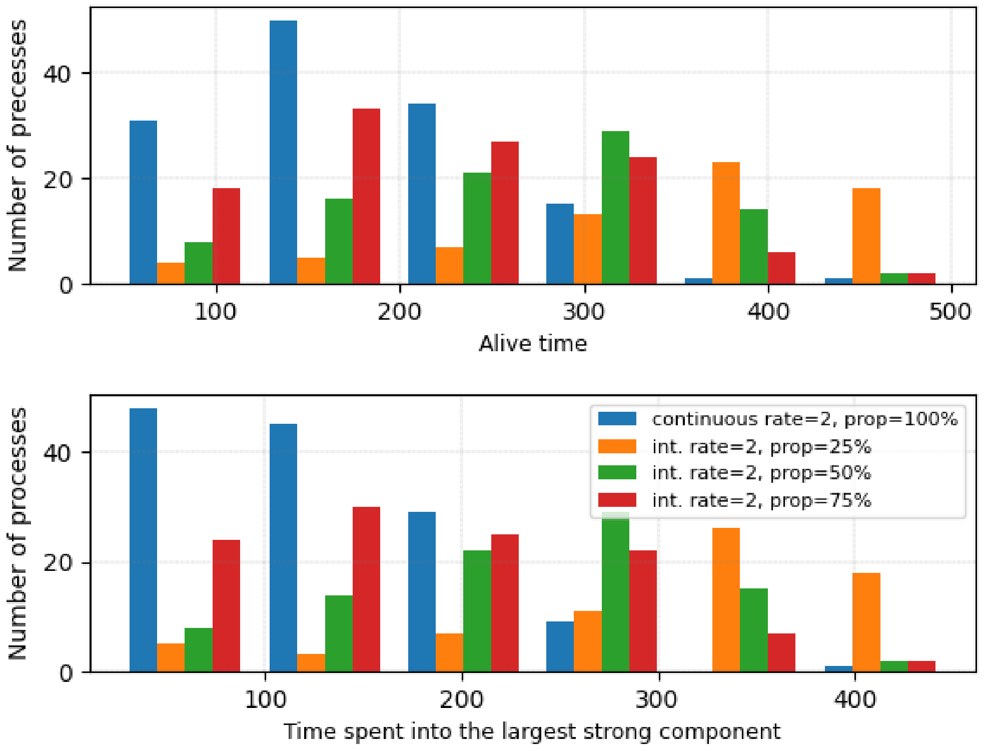


Figure 5.8: Time spent into the strong component.

(top) Histogram indicating the time nodes are ’online, (bottom) histogram showing the time spent into the largest strong component when different churn loads are applied.

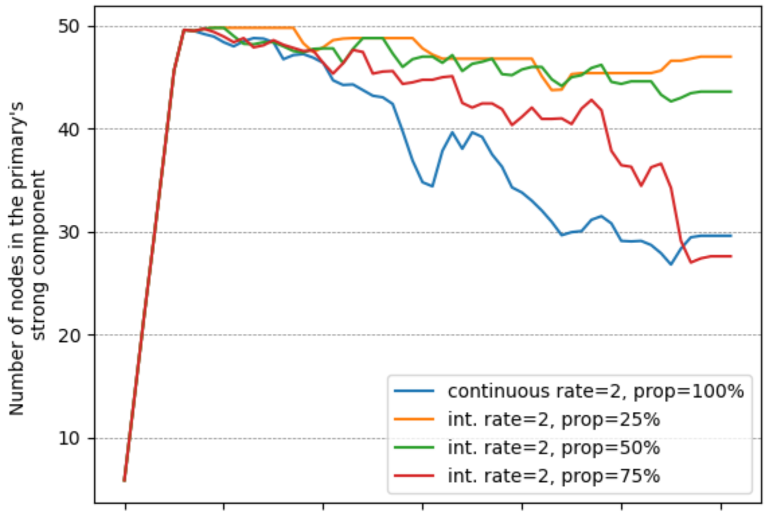


Figure 5.9: Primary’s strong component size trend.

Size of the primary’s strong component when continuous and intermittent churn rate = 2 is applied to the system.

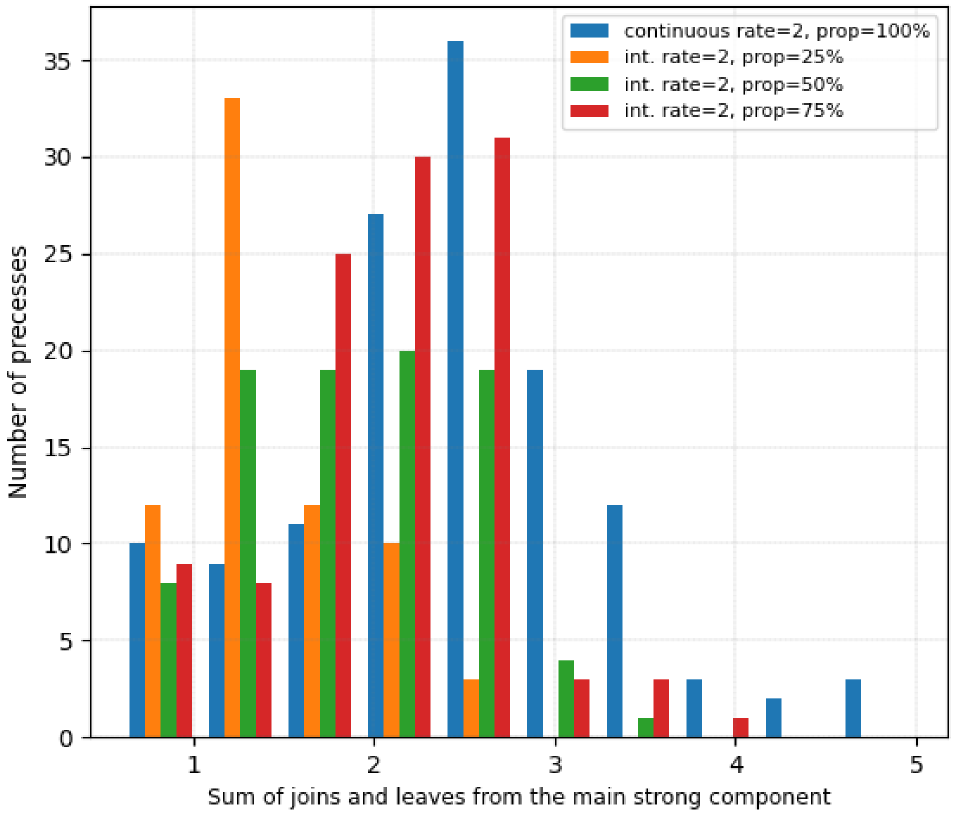


Figure 5.10: Joins and Leaves from the largest strong component.

Size of the primary's strong component when continuous and intermittent churn rate = 2 is applied to the system.

5.3.3 Blockchain

In this section an additional step is done analyzing the simulations outcomes. Until this moment, the results involved mainly the overlay network related aspects, like connectivity and strong components. Now, the application layer built on top of the overlay and communication layers will be addressed, that is represented by PBFT. First, it will be shown how the primary handles the new blocks creation and successively for how long the replicas reach consensus despite of the churn applied.

Figure 5.11 captures how the number of created blocks increases during the simulation having churn rate = 2. In PBFT, the primary is in charge of collecting transactions from the clients, pack them into a block and direct the consensus to be reached on the next block to be added to each replica local blockchain. It is possible to see the primary struggling as soon as the average connectivity decreases (figure 5.12). In fact, when the churn is continuous or at 75% load, it can be observed that the blocks stop to be added to the blockchain as soon as the average connectivity decreases around 65%, that is the minimum quorum size required by the PBFT algorithm. On the other hand, with churn loads less then 50%, the replicas continue reaching agreement and therefore adding block to the local structures. Moreover, the degradation of the overlay performance can be observed from the fact that the primary slows down the creation of new transaction blocks. This happens because of the overlay network being partitioned, and new transactions are collected by the primary with a slower rate, also because the messages are spread around the network with increased difficulty from churn consequences.

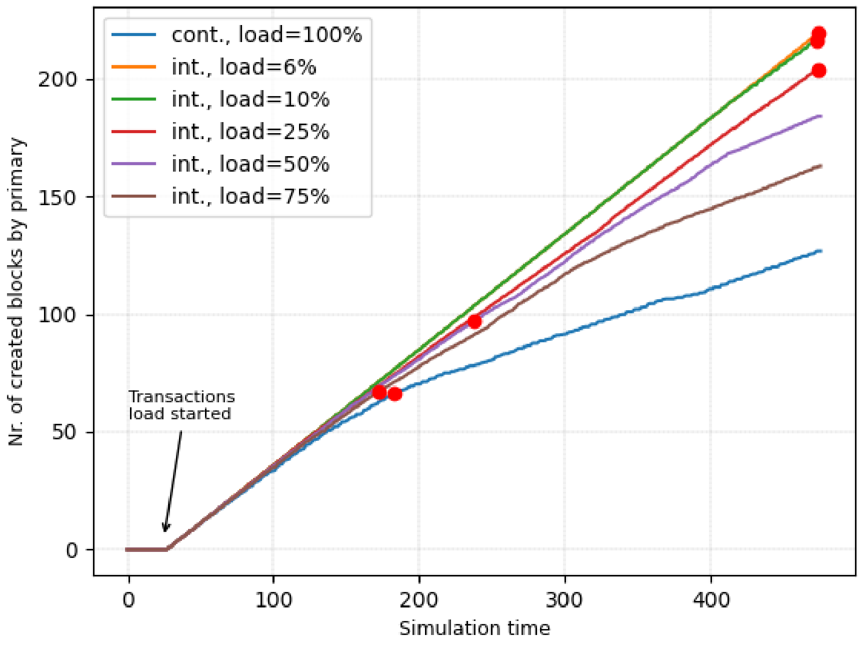


Figure 5.11: Primary blocks creation.

Number of blocks created by the primary when churn with different loads is applied and rate = 2 with an overlay network composed of 25 nodes. Red dots mark the timestamp of the last block the participants agreed upon.

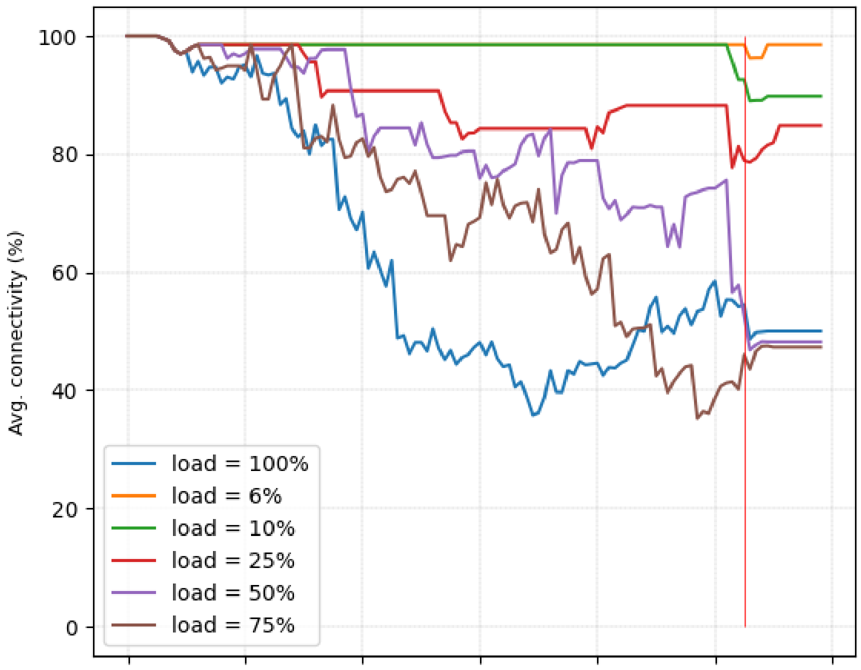


Figure 5.12: Average connectivity.

Connectivity related to figure 5.11 example.

Figure 5.13 illustrates a comparison between the primary nodes when different network sizes are involved, 25 and 50 in this case. The churn rate applied is 4, and it emerges that, when the network is composed of 25 nodes, in both cases the blockchain progress stops very soon, but when the load is lighter the primary still creates new blocks much more rapidly than the other case, with 75% load. Instead, the network with 50 nodes is more resilient in both cases, and with 25% of churn load the consensus still resists, with the replicas continuing to add the agreed blocks.

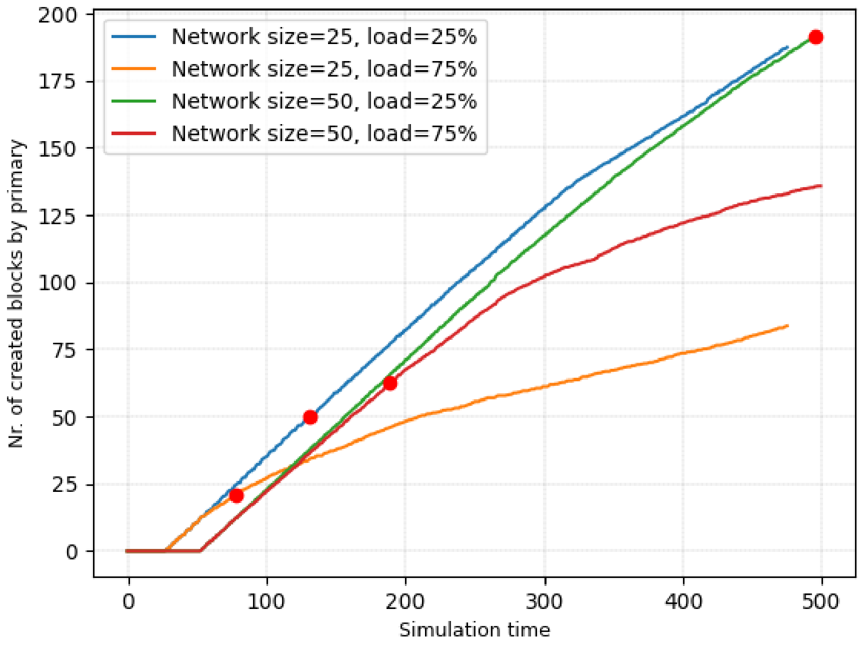


Figure 5.13: Primary blocks creation with churn rate = 4.

Comparison between how primary of networks with different size create new blocks to reach consensus on. Red dots mark the timestamp of the last block the participants agreed upon.

As an addition, in figure 5.14 the same data is shown from another perspective.

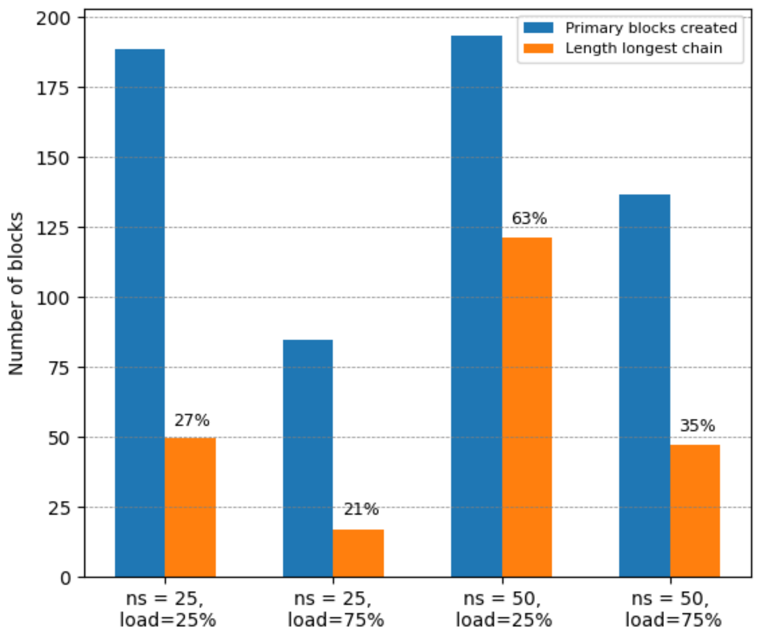


Figure 5.14: Percentage of added blocks with churn rate = 4.

Histograms showing the percentage of created blocks that were added to the blockchain replicated among the nodes.

For sure it is worth analyzing also how much the blockchain, replicated among the nodes participating to the system, progresses during the simulation in different and dynamic settings. PBFT algorithm states that the primary is in charge of creating new blocks to be gossiped, as explained previously, so the overlay network capabilities to efficiently broadcast the new blocks and consensus-related messages to all the other participants to the network depends on the connectivity parameter. In the following figures these aspects will be addressed.

In figure 5.15 there is an example of an overlay network composed of 50 nodes, to which is applied a continuous churn with different rates. The transactions load starts immediately after the bootstrap period ends, that is after 50s, and also the churn. When the churn rate has value 2, the blockchain length is much greater then the correspondent blockchains with more intensive churn loads. It is interesting to observe the trend of the strong connectivity in relation to the blockchain growth, since as soon as the size of the strong component drops under the quorum, the consensus halts. This is the case for all three simulations shown here. This happens due to the fact that once the overlay does not deliver the messages it has to deliver, the agreement primitive built upon it stops working. Even if the strong component returns growing, the consensus will not resume. This is a consequence of how the PBFT algorithm works, because once the replicas that should participate to the consensus are unable to do it, they won't receive the messages they are supposed to receive, therefore cannot advance the consensus. Even if they get reconnected to the main strong component, since they have lost some blocks to append, won't be able to append them anymore.

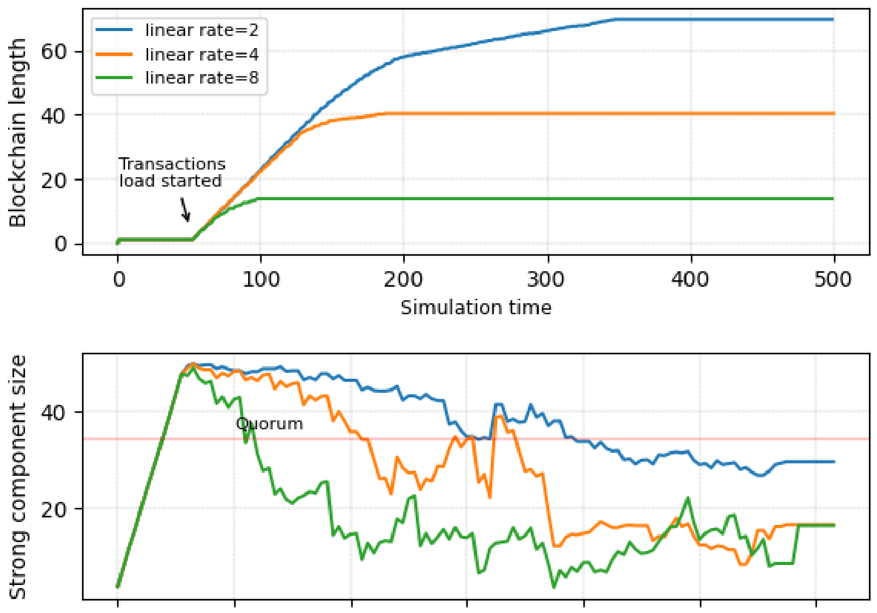


Figure 5.15: Blockchain trend with continuous churn.

(top) Blockchain trend, (bottom) strong component size.

In the next figure, 5.16, is shown that also with intermittent churn the trend is the same. As soon as the strong component drops under the necessary quorum size, the blockchain protocol halts.

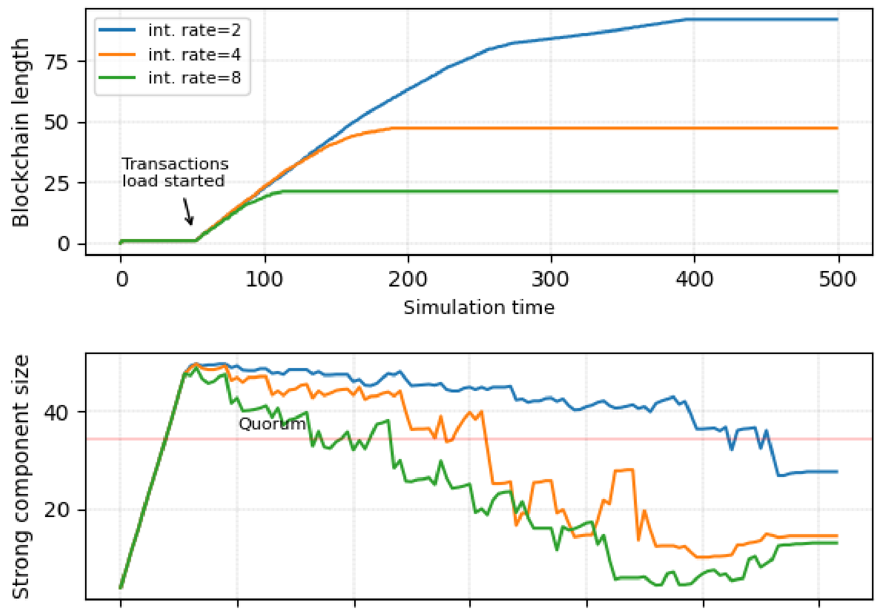


Figure 5.16: Blockchain trend with intermittent churn.

(top) Blockchain trend, (bottom) strong component size.

In figure 5.17 can be seen that the only blockchain that 'survives' is the one related to the network with size 50 and churn load of 25%, while the same load but for the network with 25 nodes slows down the blockchain progress, since the main strong component size is just above the quorum size needed for consensus. As in the previous charts, the growth of the blockchain related to the smaller network is because the bootstrapping period in that case finishes earlier, after 25s.

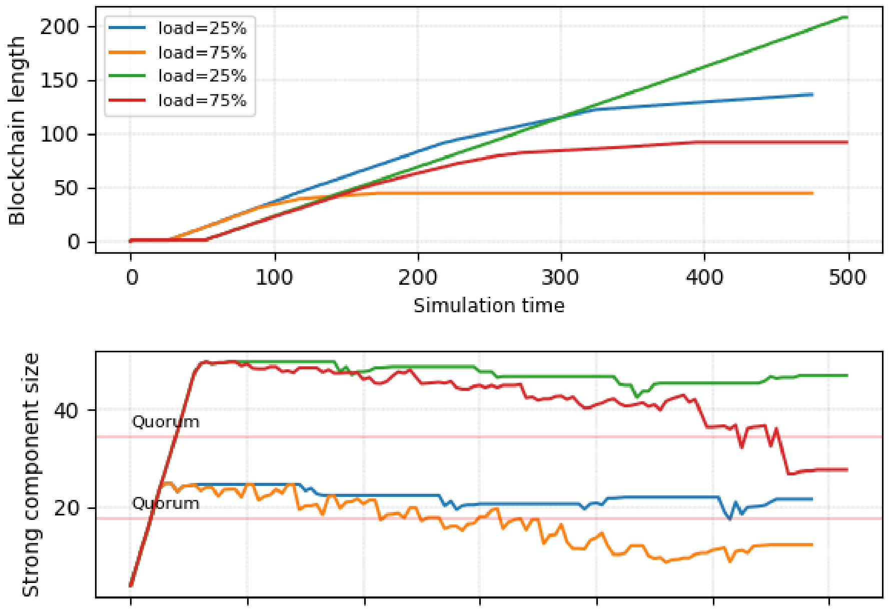


Figure 5.17: Blockchain comparison.

Comparison between the blockchain progress when networks are composed of 25 and 50 nodes respectively.

5.4 Conclusion

The main objective of this chapter was to deeply understand what practical impacts different levels of churn have on a blockchain protocol instance, in this case composed by the SCAMP overlay network, a flooding-based communication layer and the PBFT consensus algorithm. This was possible thanks to the modular approach explained in chapter 3 that inspired the simulation framework specifically created and described in chapter 4.

Throughout this chapter, it was described how the simulations in this work were structured and what input parameters they had. Following, a series of practical results deriving from the execution of several simulations were presented, with the goal of comprehending the various facets involved, from the overlay network, through the communication layer and finally the PBFT. The common thread of these simulations was the churn load and the way churn may affect the correct functioning of the entire stack.

Starting from the connectivity topic, it was observed that the average connectivity plays a crucial role, because the communication layer depends on it in order to satisfy the assumptions of the agreement primitive on top. In this case, PBFT assumes a fully connected network, therefore for the consensus to work the overlay must hold the capabilities to deliver the messages to every to all participants to the consensus. Larger networks have a better resilience, even with higher churn loads. This is because the random graph created and maintained by the SCAMP OMP has some lease mechanisms that help him to stay connected also when nodes leave the network. Unfortunately, this is not enough for keeping an adequate level of average connectivity. From this point of view, these work notices a lack of efficient lease mechanisms that are able to reconnect the network even in the case it gets partitioned.

Strong components are another valuable aspect when tackling the churn problem from an overlay point of view. In particular, since the PBFT algorithm has a special role for a node, that is the primary, it also requires this node to be able reach all the other participants to consensus, or at least a quorum of them, with both new blocks messages and consensus-related messages. An useful analysis that can be conducted regards the number and the size of the largest strong component that includes the primary. In this way, one can understand more in deep how the consensus proceeded. This aspect is linked to the previous one, since this parameter also depend heavily on the network size and the OMP's abilities to keep the strong component large. Alongside, some insights about the time spent by the nodes in the strong component and the number of times they left and joined it was shown. It is clear that higher the churn, more are the nodes that spend less time on average in the strong components, thus participating less to the consensus compared to systems in which the churn is lighter. At the OMP level, a high number of interleaved processes degrade the network connectivity, therefore a better lease mechanism is needed. Also, an obvious solution could be to increase the SCAMP's partialView and inView dimensions, in order to allow nodes to have many neighbors and have a higher probability to stay connected when churn happens.

When it comes to the blockchain-related aspects of this thesis, studies were focused on the way the primary had to create new blocks to propose, on the length of the blockchains that were created during the simulations, and the percentage of blocks created and also added to the blockchain. Following the previous conclusions,the interesting aspect is that once the consensus stops, it does not recover anymore. In other terms, when the average connectivity of the overlay network brushes or drops under the quorum size required by the PBFT algorithm, the consensus process stops. A reason for this is because the PBFT algorithm prefers safety over liveness. Even if the connectivity value comes back above the quorum size, the consensus is compromised, in the sense that several nodes may have lost important messages (blocks and consensus-related messages) that won't allow them to progress anymore, or to add blocks to their local instance. This is due also to a lack of consistent-recovery mechanisms in the case in which the network has more than f faults. In fact, PBFT includes a proactive recovery mechanism that enables the algorithm to tolerate any number of faults over the lifetime of the system, so long as fewer than one-third of the replicas become flawed within a small window of vulnerability. From the analysis run, it also emerges that when churn is intense, the primary starts to struggle even with the creation of the blocks to propose. This happens because of the decreased number of transactions it gets, received directly from the clients or broadcast by the neighbors.

To sum up, there is a wide variety of settings in which to take action, some of which will be covered in the following and final chapter, in order to improve the performance of the blockchain protocol.

Chapter 6

Future Work

The work presented in this thesis and the modular approach to analyzing blockchain consensus protocols in the face of churn, along with the proposed framework, is intended to serve as a helpful tool for future research into the various factors that may enhance the robustness and performance of blockchain protocols in the context of churn.

To this purpose, it comes from the previous chapter analysis that several aspects can be improved. At the overlay level it would be interesting to find out how the protocol behaves with different stack instances including also structured networks. Also, more sophisticated prevention and detection mechanisms may be investigated, in order to enhance the overlay capabilities. For the communication aspects, there may be implemented techniques like message retransmissions, as for the consensus algorithm, since few of them provide some type of recovery. Moreover, regarding the agreement layer, if the peers refreshment is fast, a state transfer solution should be in place, allowing nodes to quickly share the actual replica state in order to progress the blockchain. Furthermore, with the help of the framework presented here, other transaction and churn models may be implemented and tested, as well as deeper measurements about the performance of the entire composition model.

Bibliography

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

[2] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 2015.

[3] Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol. *CoRR*, abs/1802.07242, 2018.

[4] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[5] Andreas Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference*, 2001.

[6] Floris Ciprian Dinu and Silvia Bonomi. A modular approach for the analysis of blockchain consensus protocol under churn (poster). In Sara Tucci Piergiovanni and Natacha Crooks, editors, *5th International Symposium on Foundations and Applications of Blockchain 2022, FAB 2022, June 3, 2022, Berkeley, CA, USA*, volume 101 of *OASIcs*, pages 8:1–8:2. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[7] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985.

[8] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, jul 1982.

[9] Kelton Low. Simulation modeling and analysis. *McGraw-Hill*, 1991.

[10] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A flexible overlay network simulation framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.