

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/325053899>

# Assessing Security and Performances of Consensus algorithms for Permissioned Blockchains

Preprint · May 2018

CITATIONS

0

READS

184

1 author:



[Stefano De Angelis](#)

University of Southampton

4 PUBLICATIONS 9 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Blockchain Security, Performance & Scalability [View project](#)



SAPIENZA  
UNIVERSITÀ DI ROMA

FACULTY OF INFORMATION ENGINEERING, COMPUTER SCIENCE AND STATISTICS

DEPARTMENT OF COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

M.SC. IN ENGINEERING OF COMPUTER SCIENCE

# Assessing Security and Performances of Consensus algorithms for Permissioned Blockchains

AUTHOR: Stefano De Angelis

ADVISOR: Roberto Baldoni

CO-ADVISOR: Leonardo Aniello

ASSISTANT ADVISOR: Federico Lombardi

ACADEMIC YEAR

2016-2017

---

**Assessing Security and Performances of Consensus algorithms for Permissioned Blockchains.**

M.Sc. Thesis

© Stefano De Angelis. All rights reserved.

Department of Computer, Control, and Management Engineering “Antonio Ruberti”  
Sapienza University of Rome  
Via Ariosto 25, I-00185 Rome, Italy  
`stefano.deangelis@me.com`

To my parents, who encouraged me  
to fly toward my dreams.

To my Family, who always supported me.

## Acknowledgements

For the fulfilment of this work and for the achievement of this important goal, I want to thank all the people who supported me in this period. First of all I would like to thank the professor Roberto Baldoni, for having interested me to the cyber security and for giving me the opportunity to carry out this thesis project abroad. Secondly, I would like to mention the professor Leonardo Aniello, for helping and supporting me during the entire work.

Then I would also like to thank the professor Vladimiro Sassone, for giving me a warm welcome to his research group at the University of Southampton, for his precious advices and for having believed in me. Moreover I would mention also the doctor Andrea Margheri, for his kindness and helpfulness in monitoring my job.

A special thank goes to Federico, for being a friend more than a colleague. For having encouraged me, also in difficult times. For helping me in the realisation of this work, as if it was his, addressing me step by step. For being with me during important decisions and for being always ready to give me a sincere and precious advise. Thank you.

Finally I would thank to all my family, because without them this goal would not be possible to achieve and, for their unconditional love. Then to all my friends to be always with me, even when I am far away from home.

*"The main advantage of blockchain technology is supposed to be that it's more secure, but new technologies are generally hard for people to trust, and this paradox can't really be avoided."*

Vitalik Buterin

## ABSTRACT

*Blockchain* is a novel technology that is rising a lot of interest in the industrial and research sectors because its properties of decentralisation, immutability and data integrity. Initially, the underlying consensus mechanism has been designed for *permissionless blockchain* on trustless network model through the *proof-of-work*, i.e. a mathematical challenge which requires high computational power. This solution suffers of poor performances, hence alternative consensus algorithms as the *proof-of-stake* have been proposed.

Conversely, for *permissioned blockchain*, where participants are known and authenticated, variants of distributed consensus algorithms have been employed. However, most of them comes out without formal expression of security analysis and trust assumptions because the absence of an established knowledge. Therefore the lack of adequate analysis on these algorithms hinders any cautious evaluation of their effectiveness in a real-world setting where systems are deployed over trustless networks, i.e. Internet.

In this thesis we analyse security and performances of permissioned blockchain. Thus we design a general model for such a scenario in a way to propose a general benchmark for the experimental evaluations. This work brings two main contributions.

The first contribution concern the analysis of *Proof-of-Authority*, a Byzantine Fault-Tolerant consensus protocol. We compare two of the main algorithms, named *Aura* and *Clique*, with respect the well-established *Practical Byzantine Fault-Tolerant*, in terms of security and performances. We refer the CAP theorem for the consistency, availability and partition tolerance guarantees and we describe a possible attack scenario in which one of the algorithms loses consistency. The analysis advocates that Proof-of-Authority for permissioned blockchains deployed over WANs experimenting Byzantine nodes, do not provide adequate consistency guarantees for scenarios where data integrity is essential. We claim that actually the Practical Byzantine Fault-Tolerant can fit better for permissioned blockchain, despite a limited loss in terms of performance.

The second contribution is the realisation of a benchmark for practical evaluations. We design a general model for permissioned blockchain under which benchmarking performances and security guarantees. However, because no experiment can verify all the possible security issues permitted by the model, we prototype an adversarial model which simulate three attacks, feasible for a blockchain system. We then integrate this attacker model in a real blockchain client to evaluate the resiliency of the system and how much the attacks impact performances and security guarantees.

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Background and Context</b>	<b>4</b>
1.1 Blockchain Technology . . . . .	4
1.1.1 Public versus Private Blockchain . . . . .	5
1.2 Bitcoin: The Genesis of Blockchain . . . . .	6
1.2.1 A probabilistic consensus: Proof-of-Work . . . . .	7
1.3 Ethereum . . . . .	9
1.3.1 Toward alternative consensus schemes: Proof-of-Stake . . . . .	11
1.3.2 Ethereum private networks . . . . .	12
1.4 Hyperledger . . . . .	13
<b>2 Distributed Consensus</b>	<b>14</b>
2.1 Network models . . . . .	15
2.2 Failure Classification . . . . .	16
2.3 Atomic Broadcast . . . . .	17
2.4 Consensus in Permissioned Blockchain . . . . .	18
<b>3 Assessing Proof-of-Authority Consensus for Permissioned Blockchain through a CAP Theorem-based Analysis</b>	<b>21</b>
3.1 Related Work . . . . .	22
3.2 Proof-of-Authority Consensus . . . . .	24
3.2.1 Aura . . . . .	24
3.2.2 Clique . . . . .	26
3.3 Comparison of Aura, Clique and PBFT . . . . .	28
3.3.1 Consistency and Availability Analysis based on CAP Theorem . . . . .	28
3.3.2 Performance Analysis . . . . .	31



---

<b>4</b>	<b>Benchmark for permissioned blockchain</b>	<b>33</b>
4.1	System and Threat Model . . . . .	34
4.1.1	Blockchain Properites . . . . .	37
4.1.2	Threat Model . . . . .	37
4.2	Benchmark . . . . .	38
4.3	Byzantine client for blockchain . . . . .	39
<b>5</b>	<b>Conclusions and Future Directions</b>	<b>42</b>
	<b>Appendices</b>	<b>45</b>
A	Parity Byzantine client . . . . .	46
B	Consensus Finality . . . . .	51
	<b>List of Figures</b>	<b>52</b>
	<b>Bibliography</b>	<b>53</b>

# Introduction

Blockchain is one of the most disruptive technologies of the recent years. Firstly appeared as a decentralised public ledger for the Bitcoin cryptocurrency [44], blockchain is nowadays widely exploited to a broad area of application domains. Its distinguishing properties of data immutability, integrity and full decentralisation are key drivers for general purpose exploitations, ranging from Cloud computing to business-to-business applications.

Essentially, blockchain is a linked data structure replicated over a peer-to-peer network, where transactions are issued to form new blocks. Peers achieve distributed consensus on transaction ordering by placing them into new blocks; each block is linked to the previous by means of its hash. Such block creation process is carried out by distinguished nodes of the network, named *miners*, according to a distributed consensus algorithm. Besides cryptocurrencies à la Bitcoin, miners can also run *smart contracts*, immutable programs deployed and executed in a decentralised fashion upon a blockchain. Ethereum [60] is the first smart contract framework of its denomination, whose transactions represent non repudiable executions of smart contracts.

Blockchain systems like Bitcoin and Ethereum are called *permissionless*, i.e. any node on the Internet can join and become a miner. Distributed consensus is here achieved via so-called *Proof of Work* (PoW), a computational intensive hashing-based mathematical challenge. PoW enjoys strong integrity guarantees and tolerates a sheer number of attacks [27], but this comes at a huge cost: lack of performance. Moreover the PoW-based blockchain achieve eventual consensus where conflicts between blocks are possible and eventual solved later by some architectural rules, i.e. eventual consistency. This has led, together with the absence of privacy and security controls on data, to so-called *permissioned* blockchain, where an additional authentication and authorisation layer on miners is in place. Examples of permissioned blockchains are Multichain [43] and R3 Corda [16], (Hyperledger) Fabric [11] and permissioned-oriented Ethereum clients. Such systems have prompted federation of companies thus to facilitate their interactions without giving out guarantees on control and computation of data. By way of example, the Cloud Federation-as-a-Service solution [51] is exploiting a permissioned blockchain to underpin

the governance of a federation of private Clouds [26] connected via the Internet.

Being the operating environment more trusted, permissioned blockchains rely on message-based consensus schema, rather than on hashing procedures. In such setting, dominant candidates are Byzantine Fault-Tolerant (BFT) algorithms such as the Practical BFT (PBFT) [13]. Indeed, BFT-like algorithms have been widely investigated for permissioned blockchains [56] with the aim of outperforming PoW while ensuring adequate fault tolerance. Furthermore BFT-based blockchain offers, by design of the underlying consensus schemas, strong consistency on transactions despite availability. However this shift to BFT consensus in blockchain comes with a set of challenges. Even if permissioned blockchain promise to offer better performances, there is a lot of confusion around this.

Actually countless proposal have appeared, each one based on different ideas and assumptions. Most of these implementations do not have exhaustive security analysis and detailed documentations because there is not an accepted standard for consensus in blockchain [12]. This poor informations generate ambiguity between current implementations and make difficult their evaluation and comparison.

## Contributions

The goal of this thesis is to clarify the context of permissioned blockchain. We focus on the underlying consensus protocols and, by referencing the most general problems in distributed computing, we propose a solution to evaluate security and performance guarantees in a general blockchain framework.

The first contribution of the thesis is the analysis of two algorithms for blockchain permissioned. The algorithms, namely *Aura* and *Clique*, belongs to a new interesting family of BFT protocols, i.e. *Proof-of-Authority* (PoA), and promise to offer better performances than the well known PBFT. Because the poor documentation, we understand the functioning of these algorithms through reversed engineering techniques. Therefore we analyse their security guarantees, by referencing the CAP Theorem, comparing the results with the well established PBFT protocol [2]. This work has been presented at the conference ITASEC18.

The second contribution of the thesis is a benchmark proposal for permissioned blockchains. For this scope, we firstly define a standard model for such systems. Thus we design an attacker model to outline the possible vulnerabilities and attack scenarios. Then we propose the implementation guidelines for a Byzantine client, based on the attacker model, which simulate in real systems the established attacks. In such a way we propose to measure how the attacks impact to security and performance properties in such a scenarios.

## **Thesis structure**

In Chapter 1 we outline the context, by introducing the blockchain technology and the most famous implementations. In Chapter 2 we introduce the problem of consensus in distributed systems and in permissioned blockchains. In Chapter 3 is presented the CAP Theorem evaluation of security and performances between two PoA algorithms and PBFT. In Chapter 4 we introduce the benchmark evaluation, the system model for permissioned blockchain and the threat model. Finally in Chapter 5 we summarise the obtained results and we outline the future works.

# Chapter 1

## Background and Context

### 1.1 Blockchain Technology

Blockchain is a disruptive technology emerged in the recent years, firstly employed within Bitcoin's cryptocurrency [44] as public ledger. It is essentially a decentralised and distributed data structure, replicated over a peer-to-peer (p2p) network. It mainly consists of consecutive chained blocks, each one linked with the hash of the previous, containing records. These records witness transactions occurred among participants (see Figure 1.1), such transactions represents an asset exchange between the nodes of the network, e.g. Bitcoin's cryptocurrency.

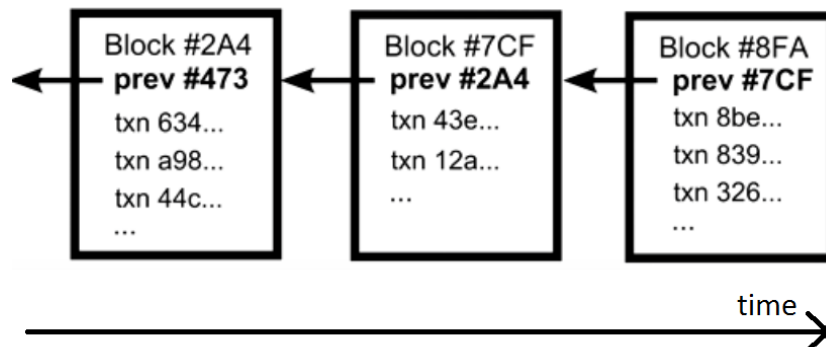


Figure 1.1: *Blockchain representation. Blocks reference each other by the hash of the previous*

The process of block creation is called *mining process* and is carried out by special nodes called *miners*, which periodically propose new blocks according to a distributed consensus algorithm. In such a way participants achieve transactions agreement, share the same state and guarantee consistency. Because decentralised and fully-replicated, blockchain also deals with the necessity of trust in a central authority as the most general modern sys-

tems do. There is not the necessity of a trusted third-party which controls all transactions and manage data, indeed every node can independently verify the consistency of the shared state. These important characteristic ensures data integrity and data immutability, since any improper alteration (e.g. tampering) would be immediately detected and rejected by the network.

Moreover, blockchains have the possibility to deploy and execute immutable programs, i.e. *smart contracts*, in a fully decentralised fashion. The first and most common framework for decentralised applications was proposed by Ethereum [60].

Blockchain systems like Bitcoin and Ethereum are so called *permissionless*, i.e. any node on the Internet can join the network and become a miner. In this case distributed consensus is achieved via the so-called *Proof of Work* (PoW), a computational intensive mathematical challenge. Conversely if there is an authentication and authorisation layer for miners, then the blockchain is *permissioned*.

### 1.1.1 Public versus Private Blockchain

Blockchain systems can be classified on the basis of the access types in different permission models. Participants of a blockchain network have rights to: (i) accessing data on the blockchain (*Read*), (ii) submitting transactions (*Write*) and (iii) updating the state with new blocks (*Commit*) [33].

The work proposed by BitFury and Garzik in [29] and [30], define blockchain systems on the basis of these rights. Specifically a blockchain can be essentially divided in two main classes:

- *public blockchain*: no restrictions applied on Read operations;
- *private blockchain*: only a predefined list of entities is allowed to run Read operations.

Hence for the Write and Commit operations are identified two classes of blockchain:

- *permissionless blockchain*: there are no restrictions on Write and Commit operations;
- *permissioned blockchain*: only a predefined list of entities is allowed to Write and Commit operations.

In permissionless blockchain anyone can join the network, execute Write operations, Commit operations and, participate to consensus. Contrarily in permissioned blockchain nodes are known thanks to an authentication mechanisms, and only these nodes can control the blockchain.

Table 1.1 shows a comparison between four blockchain models based on these classes.

	Read	Write	Commit
Public permissionless	Open to anyone	Anyone	Anyone
Public permissioned	Open to anyone	Authorised participants	All or subset of authorised participants
Consortium	Restricted to an authorised set of participants	Authorised participants	All or subset of authorised participants
Private permissioned	Fully private or restricted to a limited set of authorised nodes	Network operator only	Network operator only

Table 1.1: Main types of blockchain systems

Public permissionless blockchains, e.g. Bitcoin, operate in a hostile environment, requiring the use of crypto-techniques to incentivise participants to behave honestly. Contrarily, private permissioned blockchains operate in an environment where participants are already known. In this scenario nodes are incentivised to behave honestly because they are authenticated, so every misbehaviour can be detected and convicted. Consortium blockchains are a special class of permissioned blockchain where members of a consortium operate in a given context.

The private settings where participants are known and authenticated, make blockchain platforms step away from their original purpose and enter in the domain of database-replication protocols, notably, the classical *state-machine replication* (SMR) [52], where only a predetermined set of nodes participate to the network..

## 1.2 Bitcoin: The Genesis of Blockchain

Bitcoin [44] was the first popularised application of permissionless blockchain for a virtual currency. It is an electronic payment system based on cryptographic proof, which avoids the need of a centralised authority, e.g. a bank. Current payment systems require a trusted third-party to process transactions, however this mediation is often subject to transaction costs, limiting the minimum practical transaction size and cutting off the possibility for small casual transactions.

To guarantee machine-to-machine payments, the Bitcoin protocol uses a flood propa-

gated p2p network that processes transactions in a fully decentralised system. Transaction order is guaranteed by an hash based consensus mechanism. To maintain a consistent, immutable, and therefore trustworthy, ledger of transactions, nodes share the Bitcoin Public Ledger Figure 1.2, i.e. a blockchain with the list of all transactions ever made.

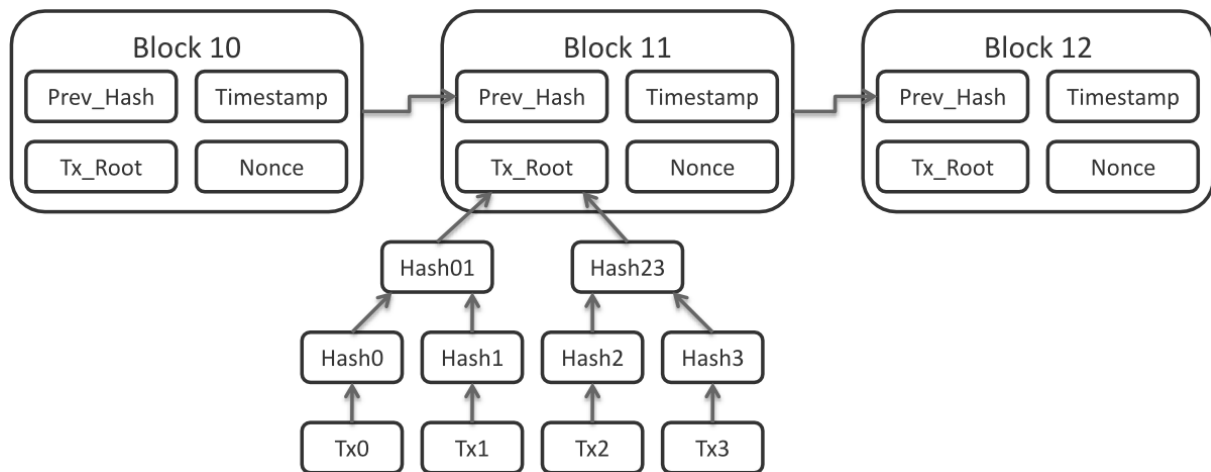


Figure 1.2: *Bitcoin Public Ledger.*

Transactions are managed by asymmetrical cryptograph as a chain of digital signatures. Each owner transfers the coin by digitally signing a hash of the previous transactions in input and the public key of the next owner and adding these to the end of the coin as output. In this way only users which have previously received coins, can process new transactions. In Bitcoin, every node has a balance of owned cryptocurrency coins identified by the history of previous transactions.

When a node receive a transaction verifies the crypto-currency balance of the sender by checking his previous unused transactions with its available balance. In such a way only valid transfers are accepted. Moreover, Bitcoin protocol prevents also the *double-spending* problem: in order to be validated, each input transaction is checked if used or unused. Only unused transactions are accepted.

### 1.2.1 A probabilistic consensus: Proof-of-Work

To achieve consensus on transactions order over the blockchain and to prevent duple-spending, Bitcoin's network is based on the *Proof-of-Work* (PoW) algorithm. It consists in a computational intensive hashing task. Specifically, to solve a block miners need to find a random number (also called *guess*) which acts as a proof of work (see Figure 1.3); indeed, such a number concatenated to transactions collected inside the block has to provide the overall hash of the block less than the target number. The target number is regulated



according to the so-called *blockchain difficulty* that regulates the average time spent by miners to accomplish such a task and create a new block.

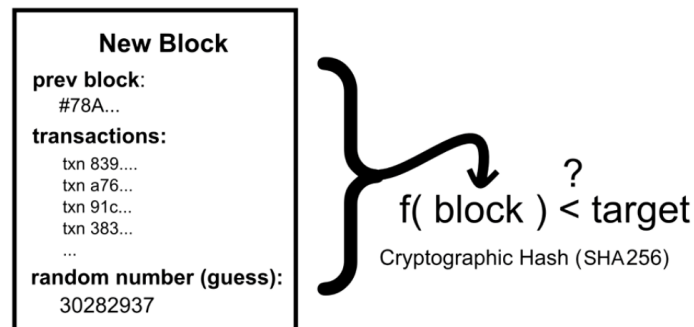


Figure 1.3: *Proof-of-Work as a computational puzzle to solve a block*

Once a miner solves a block, it broadcasts that block over the network. They consider such a block as the latest of the chain and start mining new blocks to be appended on it. For the sake of simplicity, we can say that once a miner has created a new block, it becomes part of the chain. However if multiple miners concurrently add a block, a transient fork is created which is usually resolved over time by concatenating other blocks as by design miners always consider the longest chain. Miners are incentivised to correctly support the network due to rewards obtained for successfully mining blocks and for fees that each transaction can optionally acknowledge to them [44, 28].

When a block is part of the chain, it means that all miners have agreed on its content, hence it is practically non-repudiable and persistent unless an attacker has the majority of miners' hash power. In such case the adversary is able to create a chain fork or reverse a transaction. Assuming a majority of hash power controlled by honest miners, the probability of fork, with depth  $n$ , is  $\mathcal{O}(2^{-n})$  [6]. Since the probability of forks decreases exponentially, the users simply waiting for a small number of blocks to be appended (e.g. 6 blocks in Bitcoin), can be sure that their transactions are permanently included with high confidence (avoiding the double spending attack). However, in [22] the authors showed as "*majority is not enough: Bitcoin mining is vulnerable*" if only 25% of the computing power is controlled by an adversary.

Although such systems provide strong integrity properties, PoW-based blockchains have a main drawback: *performance*. This lack of performance is mainly due to the broadcasting latency of blocks on the network and to the time-intensive task of PoW. Indeed thousands of miners spread world-wide are required to render tampering with transactions computational infeasible and broadcast blocks over this kind of network topology takes very long. Thus as a matter of fact, each transaction stored on a blockchain has a

high confirmation time, which causes an extremely low transaction throughput. In Bitcoin, the average latency is 10 minutes, and the throughput is about 7 transactions per second, while for Ethereum the latency is on average 10 seconds with 15 transactions per second as throughput [6]. Moreover, PoW is energetically inefficient leading a huge waste of money and resources to make the hashing computations.

## 1.3 Ethereum

Ethereum [60] is the second main project (open source) of blockchain, following Bitcoin. It builds on the idea of *smart contracts*, immutable programs deployed and executed autonomously on blockchain as transactions. It is featured by a Turing-complete programming language which introduce to the distributed computation system that allows anyone to write smart contract and therefore decentralised applications. This widens the application domains of blockchain making it a computational infrastructure with any of the participant in control of it.

Like Bitcoin, the shared state is managed by enrolled accounts which are featured by a balance of the Ethereum's cryptocurrency, i.e. *ether*. The ether is the main internal crypto-fuel of Ethereum, and is used to pay transaction's fees. Like Bitcoin, nodes run a proof-of-work consensus to achieve agreement on transactions order and to maintain the shared state consistent. Every time a transaction is accepted, the state is updated by the Ethereum State Transition Function.

There are two types of accounts: *externally owned accounts*, controlled by private keys, and *contract accounts*, controlled by their contract code. The first are similar to the classic accounts of Bitcoin, whereas the others are allowed to read/write internal storage interacting with smart contracts.

### Messages and Transactions

Transactions, which in Ethereum are sent from external accounts, are quite similar to the Bitcoin's one. Two more fields have been introduced:

1. **Gas Price:** a scalar value representing the number of Ether to be paid for each computational step of the transaction's execution;
2. **Gas Limit:** a scalar value representing the maximum amount of gas that should be used in executing the transaction.

In order to prevent accidental or hostile infinite loops or other computational wastage in code, each transaction must have a computation limit on code execution. The unit

*gas* setup a fee system in such a way is required by attackers to pay, proportionately to every resource that they consume, including computation, bandwidth and storage; hence, any transaction that leads to the network consuming a greater amount of any of these resources must have a gas fee roughly proportional to the increment.

Moreover, Ethereum introduce the concept of *message*. Messages are produced by contracts and essentially represent the call to a smart contract function. These are virtual objects that are never serialised, they exist only in the Ethereum execution environment and are sent by contract accounts. A message contain:

- the sender of the message;
- the recipient of the message;
- the amount of ether to transfer alongside the message;
- the Gas Price.

## GHOST Protocol

Blockchains with fast confirmation times currently suffer of low security due to a high stale rate. Blocks take a certain time to propagate through the network, e.g. if miner A mines a block and then miner B happens to mine another block before miner A's block propagates to B, miner B's block will end up wasted and will not contribute to network security by violating consistency: forks [59]. Moreover blockchains which quickly produce blocks are very likely to lead to one mining pool, i.e. batch of miners working together to get higher computational power, having a large enough percentage of the network hashpower to have de facto control over the mining process.

In Bitcoin forks are solved by converging to the longest chain of blocks, indeed Ethereum applies the "Greedy Heaviest Observed Subtree" (GHOST) protocol [53]. Here to calculate the main chain, also the stale descendants of the block's ancestor (in Ethereum jargon, i.e. *uncles*) are added to the calculation on which block has the largest total proof of work backing it.

Figure 1.4 illustrates a chain selected by the GHOST protocol. Consider the block 1B, it is supported by blocks 2B, 2C, and 2D that extend it directly, and include it in their chain. Similarly, blocks 3C, 3D, and 3E support both 1B and 2C as part of their chain. At each fork is selected the block leading to the heaviest subtree as part of the main chain. In such a way every addition of new forks to a subtree, makes harder the omission of the root from the main chain. Moreover to address the security issue of centralisation,

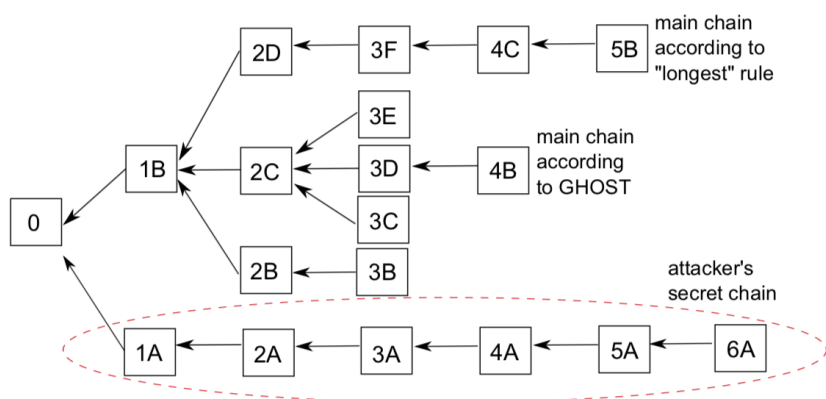


Figure 1.4: Block tree where the longest chain is not the one selected by the GHOST protocol.

this protocol also provide block rewards to stales: a stale block receives 87.5% of its base reward, and the nephew that includes the stale block receives the remaining 12.5%.

Ethereum implements a simplified version of GHOST which only goes down seven levels. Specifically, a stale block can only be included as an uncle by the 2nd to 7th generation child of its parent, and not any block with a more distant relation. Calculations show that five-level GHOST with incentivisation is over 95% efficient even with a 12s block time, that is the time between the creation of two blocks in Ethereum compared to 10 minutes in Bitcoin.

### 1.3.1 Toward alternative consensus schemes: Proof-of-Stake

Ethash is the PoW implementation for Ethereum. Like in Bitcoin it suffers of the main drawbacks of probabilistic algorithms. Thus, has been proposed by Ethereum the *Proof-of-Stake* (PoS), with the Nxt cryptocurrency [15], as alternative consensus algorithm for public blockchain. PoS employs a deterministic (pseudo-random) solution to define the creator of the next block and the chance that an account is chosen depends on its wealth (i.e. the stake). In PoS the blocks are said to be *minted*, rather than mined. Moreover, generally all coins are created at the beginning, and thus the total number of coins never change afterwards, but there exist some PoS versions where new coins can be created. Therefore, in the original version of the PoS there are not any block rewards, hence the minter take only the transaction fees [49].

In the Ethereum's PoS implementation, called Casper [46], a set of validators take turns proposing and voting on the next block. The weight of each validator's vote depends on the size of its ether deposit (i.e. the stake). The process of creating and agreeing to new blocks is then done through a consensus algorithm that all current validators can

participate in. Two major types of consensus can be identified for PoS protocol:

1. **Chain-based PoS:** the algorithm pseudo-randomly selects a validator during each time slot (eg. every period of 10 seconds might be a time slot) to propose a block. The block is then appended to the blockchain
2. **BFT-style PoS:** validators are randomly assigned the right to propose blocks, but agreeing on which block is canonical is done through a multi-round process where each validator sends a "vote" for some specific block during each round, and at the end of the process all (honest and online) validators permanently agree on whether or not any given block is part of the chain. Note that blocks may still be chained together; the key difference is that consensus on a block can come within one block, and does not depend on the length or size of the chain after it.

Differently from PoW, the PoS algorithms do not require waste of energy because they do not need long computational periods. Hence the minting process is performed by coin owners, putting money (i.e. a deposit) at stake. Here honest validators receive small rewards for making blocks but, security against malicious behaviours should be granted by penalties. Indeed in this case, the costs of reverting transactions are hundreds or thousands of times larger than the rewards that they got in the meantime. As the Ethereum's creator, Vitalik Buterin, wrote in a blogpost [9] : "In Proof-of-Stake, security comes not from burning energy, but rather security comes from putting up economic value-at-loss".

However many chain-based PoS algorithms are rising with a reward policy for producing blocks without penalties [48] and this lead to the *nothing at stake* problem. Thus this issue, in case of multiple chains, incentivise validators to sign blocks on every chain since it does not cost anything to them. In this scenario a validator is encouraged to act badly and creates as many blocks as possible leading to multiple forks and consequently to double spend attacks.

### 1.3.2 Ethereum private networks

There exist different implementations of the Ethereum protocol, most of them offer the possibility to be used in consortium chains for private settings. Two of the most interesting Ethereum's client are Geth [21], the Ethereum implementation in Golang language, and Parity [55], a Rust-based implementation. Both of them offer, through special configurations, the possibility of build permissioned private blockchain environments with lightest consensus protocols.

These type of chains are used in development testnets where distributed applications are tested and debugged before the deployment on the main Ethereum blockchain. More-

over private networks, optimised for permissioned settings, can be also used for enterprise use cases where performance and privacy are important.

Beyond the currently PoW consensus, BFT-based algorithms can be employed in such a scenario to guarantee highest performances. In the next chapter we will see which are the most prominent algorithms and some of their use cases.

## 1.4 Hyperledger

The Hyperledger consortium is an ombrella project under the Linux Foundation that incubate a large number of blockchain projects. Fabric [23] is the first released smart contract, here called *chaincode*, blockchain system for permissioned settings. It offers a customisable platform in terms of pluggable consensus mechanism, regulation on member enrolment and visibility of data. Avoiding the use of proof-of-work prevents Fabric to be used in trust-less settings, but its performance is remarkable and enabled a wide range of industrial applications.

Firstly released under the most conform characteristics of a state-machine replication system, i.e. release *v0.6* [11], actually a more elaborate design has been adopted. The last release *Fabric v1.0* [25] separates the execution of chaincode transactions from the ordering to ensure agreement and consistency. This new architecture promises to bring better scalability. The fundamental key features of this architecture are:

- Channels for sharing confidential informations;
- An Ordering Service to guarantee transactions agreement and consistency, over the network;
- An Endorsement policy for transactions;
- CouchDB world state which supports queries;
- A customisable Membership Service Provider (MSP).

At the beginning, the first release of Fabric had come with a native implementation of PBFT [13] as consensus protocol. With the new version, the Ordering Service can be provided by the Apache Kafka protocol [34], i.e. a crash tolerant pub/sub consensus and, by a noops protocol, called SOLO, which essentially delegate the ordering to one peer. Furthermore a new implementation of PBFT is under development.

# Chapter 2

## Distributed Consensus

Modern systems, evolved from centralised to distributed architectures where interconnected computers, i.e. *replicas*, communicate and collaborate for a common goal combining their resources, i.e. *Distributed Computing*. Replication in computing helps to improve availability, reliability and fault-tolerance. There exists two types of replication: (i) *data replication*, used in Distributed Database systems where data are stored across replicas, (ii) *computational replication* where replicas execute the same computing task many times. Furthermore replicated systems can be identified as:

- ***active replication***: replicas process the same request;
- ***passive replication***: each single request is processed on a single replica and the result is dispatched to the other replicas.

The simplest active replication scheme is the *primary/backup* (or master-slave in the database field) where a primary replica is in charge to order all the requests and dispatch them to all the backups. On the other side, if any replica processes a request and then distributes a new state, then this is a *multi-primary* scheme (or multi-master).

A fundamental problem in distributed computing is coordinating replicas operations also in unreliable settings, i.e. *Distributed Consensus*. The most common paradigm for implementing this type of services is the State Machine Replication (SMR) [52], wherein a deterministic state machine is replicated across a set of nodes, such that it functions as a single state machine despite failures. Replicas start in the same initial state and apply operations in the same order. Considering the client/server paradigm, SMR systems ensure resiliency against single-point-of-failure, contrarily to services using a single centralised server. Clients submit operations to the replicated server system, i.e. *transactions*, each one causing a state transition and return a deterministic result.

Blockchain is an implementation of a distributed database based on active replication with a multi-primary scheme.

To be feasible for an SMR system, distributed consensus algorithms must guarantee two main properties:

***Safety.** This property states that the algorithm should do anything wrong during its normal execution. This property prevent unwanted executions and ensure different properties like consistency, validity agreement, and so on. If it is violated at some point in time, safety property will be never satisfied again.*

***Liveness.** This property ensures that eventually something good happens. It is a property of a distributed system which grant that the algorithm works properly in time and that sooner or later every execution completes correctly.*

According to these properties different consensus algorithms have been proposed in the last years. All of this protocols rely on different network models and guarantee correctness despite some fault model. The fault model and the synchrony assumptions play a fundamental role for safety and liveness guarantees in SMR systems. Fixing a particular type of fault and the assumptions on network synchrony, one can characterise the model by its resiliency, i.e. the maximum number of faults which can be tolerated in any protocol in the given model.

## 2.1 Network models

Typically replication protocols, to guarantee resiliency, require an upper bound on the time for messages to be delivered and on processor clocks synchronisation. In *synchronous* environments these upper bound are fixed and defined a priori, therefore achieving consensus results easy. However synchronous solutions are unfeasible in real environments where networks can be partitioned, messages may be delayed or lost, nodes may crash, may be subverted by an attacker and other unpredictable events.

*Asynchronous* systems avoid the necessity of any assumption about synchronised clocks and timely network behaviours. Therefore protocols designed in this scenario could provide the best resiliency to any type of fault. However due to a fundamental discovery by Fischer et al., i.e. the celebrated “FLP impossibility result” [24], which shows that the existence of timing bounds is necessary to achieve any resiliency, is proved that in this model, if even a single processes can crash, no consensus can be achieved.

The most accepted network model relies on *eventual synchrony* [20]. This simulates an asynchronous network which eventually, after undefined bound in time, behave synchronously and delivers all the messages. Protocols in this model never violate safety as



long as the trust assumptions on the kind and the number of faulty nodes are met. During the synchronous period, the protocol terminate correctly, guaranteeing liveness.

## 2.2 Failure Classification

Distributed systems can be subject to several issues caused by replication like random communication delays and systems failures. These brings to the systems problems in distributed consensus, and may cause the violation of safety and liveness properties.

In [17], Cristian et al. propose a classification of failures in several nested classes, so that protocol's complexity can be measured with the size of the class of failures it tolerates. Failures are classified with respect to system components: a failure occurs when a component does not behave in the correct manner.

An *omission* failure occurs when a component never gives the expected output to a specified input. A *timing* failure occurs when the component responds too early, too late or never to a request. A *Byzantine* failure [40] occurs when the component does not behave correctly. More precisely a Byzantine fault can be cause by an omission fault, a timing fault or if the component gives different output from the one specified. An important subclass of this failure is the so called *authentication-detectable* Byzantine failure, where any corruption of messages is detectable by using a message authentication protocol, e.g. public-key cryptosystems based on digital signatures.

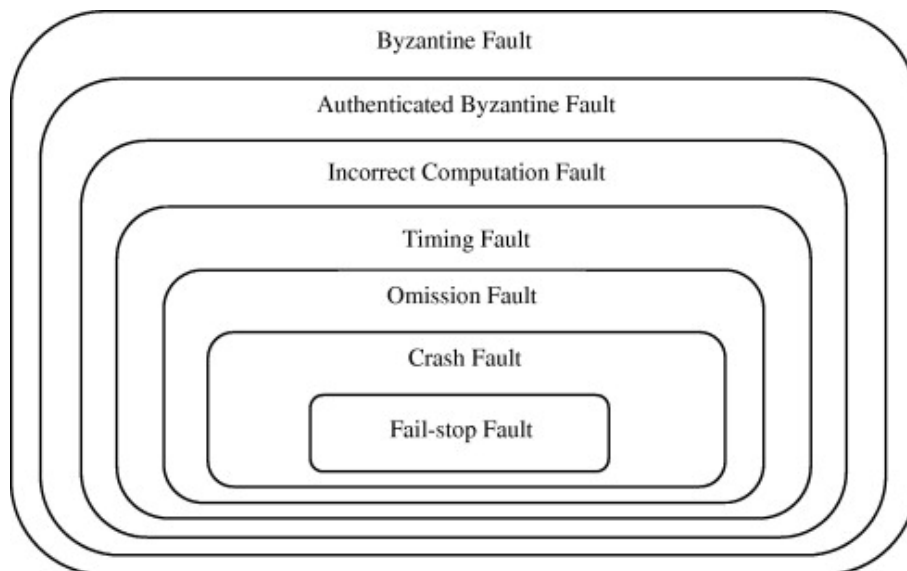


Figure 2.1: *Byzantine failures model.*

Figure 2.1 shows the Byzantine Failures model diagram where is explained the clas-

sification of failures. Crash failures, i.e. when a component stops to work, are a subclass of omission failures (*Fail-stop* failure is a special class of crash, where after a first omission a component omits to respond to all subsequent input events). Omission failures are a subclass of timing failures, timing failures are a subclass of authenticated-detectable Byzantine failures which are a subclass of the class of all possible failures, the Byzantine failures.

The nested nature of the failure classes makes it easy to compare fault-tolerant protocols.

## 2.3 Atomic Broadcast

The most relevant algorithm for SMR systems, for guaranteeing agreement and order on requests, is the *atomic broadcast* (or *total order broadcast*) consensus.

Atomic broadcast is a *reliable broadcast* which ensures also that correct nodes output or *deliver* the same sequence of messages, i.e. total order. Specifically it ensures the following safety properties [31]:

**Validity.** *If a correct node  $p$  broadcasts a message  $m$ , then  $p$  eventually delivers  $m$ .*

**Agreement.** *If a message  $m$  is delivered by some correct node, then  $m$  is eventually delivered by every correct node.*

**Integrity.** *No correct node delivers the same message more than once; moreover, if a correct node delivers a message  $m$  and the sender  $p$  of  $m$  is correct, then  $m$  was previously broadcast by  $p$ .*

**Total Order.** *For messages  $m_1$  and  $m_2$ , suppose  $p$  and  $q$  are two correct nodes that deliver  $m_1$  and  $m_2$ . Then  $p$  delivers  $m_1$  before  $m_2$  if and only if  $q$  delivers  $m_1$  before  $m_2$ .*

There exists several implementations of the atomic broadcast consensus, however a fair comparison between each algorithm is still considered challenging [12] as some of them require a *synchronous* network, while others can work in a *eventually synchronous* network which may strongly impact performance and scalability. Therefore safety and liveness properties in each model are guaranteed under different trust assumptions. Typically the generic trust assumption for a system with  $n$  nodes says that no more than  $f < n/k$  nodes became faulty, thus the other  $n - f$  nodes are correct. For instance, in SMR systems considering the simplest replication method, i.e. the primary/backup, and the special fail-stop model, only  $f + 1$  replicas are required by tolerating up to  $f$  faulty nodes.

Failure Mode	Synchronous	Asynchronous	Partially Synchronous
Fail-Stop	$f$	$\infty$	$2f + 1$
Omission	$f$	$\infty$	$2f + 1$
Byzantine Authentication	$f$	$\infty$	$3f + 1$
Byzantine	$3f + 1$	$\infty$	$3f + 1$

Table 2.1: Smallest number of nodes for which a  $f$ -resilient consensus protocol exists.

In Table 2.1 is shown the comparison of trust assumptions required for consensus, considering all the failure models introduced before in different synchrony models.

All those algorithms can be categorised in two family: (i) *Crash Tolerant* and (ii) *Byzantine Fault Tolerant* (BFT).

The main references for those families are (i) Paxos [38, 39], proposed by Lamport in its original version only tolerant to a crash of  $t < n/2$  nodes, and (ii) Practical Byzantine Fault Tolerance (PBFT), proposed by Castro and Liskov [13], which is able to resist to  $f < n/3$  Byzantine nodes.

The PoW algorithm, even though is not related to neither families, can be categorised as a BFT algorithm on a SMR model. Rigorously analysed in the technical report [1], it is modelled in a network with a near-infinite number of nodes, with each node representing a very small unit of computing power and having a very small probability of being able to create a block in a given period. In this model, the protocol has 50% fault-tolerance assuming zero network latency (synchronous network),  $\sim 46\%$  (Ethereum) and  $\sim 49.5\%$  (Bitcoin) fault-tolerance under actually observed conditions, but goes down to 33% if network latency is equal to the block time, and reduces to zero as network latency approaches infinity (asynchronous network) [46].

## 2.4 Consensus in Permissioned Blockchain

Blockchain is essentially a distributed database holding a list of records controlled by a trustless p2p network. Actually the modern distributed database implementations, e.g. DynamoDB [18], Cassandra [37], Facebook TAO [8], referencing the CAP Theorem, favour availability and partition tolerance over consistency. Furthermore these systems offer the concept of *eventual consistency* where replicas in a distributed database eventually converge to identical copies. Such scenarios address the scalability and availability issues associated with strong consistency semantics, i.e. linearizability [32, 58].

**CAP Theorem** The CAP Theorem [7] states that in a distributed data store only two out of the three following properties can be ensured: *Consistency* (C), *Availability* (A) and *Partition Tolerance* (P). Thus any distributed data store can be characterised on the basis of the (at most) two properties it can guarantee, either CA, CP or AP. However, since replicated systems may be subject to failures that can be thought as partition, partition-tolerance must be guaranteed, thus only CP or AP systems are feasible.

Even if the blockchain system is a distributed database, it should enforce strong consistency, i.e. atomic broadcast, among transactions, to prevent issues such as the double-spending attack. However PoW-based permissionless blockchain cannot guarantee strong consistency because they run a sort of *eventual consensus* where forks are possible and eventually resolved, i.e. such as the longest branch rule of Bitcoin or the GHOST protocol in Ethereum. These systems do not guarantee atomic broadcast and cannot be identified as SMR [58]. The eventual consensus in PoW (and PoS) leads to favour availability over consistency guaranteeing only eventual consistency [46].

Conversely to permissionless blockchains, permissioned blockchains usually do not employ cryptocurrencies and their purpose is more oriented to exchange of asset and transactions. They can be faster by design, implementing a classical SMR schema. Here the authentication of nodes changes the trust level, allowing thus to employ a consensus schema lighter than the PoW. A report of Swanson compares the two models [54].

Hence blockchain are turning back to classical, strongly-consistent BFT distributed consensus, based on atomic broadcast. These systems avoid the possibility of forks and, in blockchain jargon this property is identified as *consensus finality* introduced in [57] (in Appendix B is detailed the proof of strong consistency in finality).

**Finality.** *Given two correct nodes  $n_1, n_2$ , if  $n_1$  appends a block  $b$  to its copy of the blockchain before appending block  $b'$ , with  $b \neq b'$  then no correct node  $n_2$  appends block  $b'$  before  $b$  to its copy of the blockchain.*

Classical BFT-based consensus algorithms are strongly consistent, e.g. PBFT, and guarantee finality to blockchains based on it. The most feasible algorithms for permissioned blockchain belongs essentially to two implementations:

- *Mining Rotation algorithm* [29]. Distinguished nodes are in charge to seal all of the blocks of the blockchain electing a leader periodically.
- *PBFT (Practical Byzantine Fault Tolerant)* [13]. Traditional Byzantine Fault Tolerant consensus or similar.

This trend of moving on strongly-consistent BFT-based consensus in blockchain is exemplified by all the emerging practical systems which are rising. The most important blockchain technologies that offer BFT-based distributed ledger systems are: (i) Hyperledger Fabric with Apache Kafka [34] (i.e. a crash tolerant pub/sub) and with PBFT, (ii) Tendermint [35] a BFT consensus protocol for blockchain similar to PBFT, (iii) R3 Corda with Raft [47] (i.e. a popular variant of Paxos, thus crash tolerant) and BFT-SMaRT [5] (i.e. the most advanced and tested implementation of a BFT consensus), (iv) Multichain [43], (v) Sawtooth Lake [36] with the Proof-of-Elapsed-Time (a.k.a. PoET, i.e. a novel protocol based on the insight that PoW essentially imposes a mandatory but random waiting time for leader election) and many others. An exhaustive comparison between different algorithms employed by existing technologies among those properties is available in [12]. The work compares these technologies and their requirements in the sense of safety and liveness properties. However from this study emerge a set of challenges.

BFT-based protocols proposed for permissioned blockchain guarantee strong consistency but have to give up availability or partition-tolerance because the CAP Theorem. Moreover a fair comparison between each algorithm is still considered challenging [12] as some of them requires a *synchronous* network, while others can work in a *eventually synchronous* network which may strongly impact on performance and scalability. Indeed, these protocols are based on developers' ideas, not relying on established knowledge on blockchain. Most of them come without formal expression of their trust assumptions and security model. There is not an agreed consensus in the industry and this generates a lot of confusion in the public opinion [12].

In the next section we propose an analysis of two Mining Rotation algorithms against the PBFT one, through the help of the CAP Theorem, to advocate liveness and safety guarantees.

## Chapter 3

# Assessing Proof-of-Authority Consensus for Permissioned Blockchain through a CAP Theorem-based Analysis

Recently, a new line of research on *distributed consensus* is arising, as it plays a key role in the blockchain ecosystem. In permissioned settings, dominant candidates are *Byzantine fault tolerant* (BFT) algorithms such as the Practical BFT (PBFT). Indeed, BFT-like algorithms have been widely investigated for permissioned blockchains with the aim of outperforming PoW while ensuring adequate fault tolerance.

Proof-of-Authority (PoA) [45] is a new family of BFT algorithms which has recently drawn attention due to the offered performance and toleration to faults. It is currently used by Parity [55] and Geth [21], two well-recognised clients for permissioned setting of Ethereum. Intuitively, the algorithms operate in rounds during which an elected party acts as *mining leader* [29] and is in charge of proposing new blocks on which distributed consensus is achieved. Differently from PBFT, PoA requires less message exchanges hence provides better performance [19]. However, the actual consequences of such performance improvement is quite blurry, especially in terms of availability and consistency guarantees in a realistic *eventually synchronous* network model such as the Internet [10]).

To this aim, we take into account two of the main PoA implementations, named Aura [4] and Clique [14], which are used by Ethereum clients for permissioned-oriented deployments. In particular, the lack of appropriate documentation and analysis prevents from a cautious choice of PoA implementations with respect to provided guarantees, fault tolerance and network models.

In this chapter, we first derive the actual functioning of the two PoA algorithms, both from the scarce documentation and directly from the source code. Then, we conduct

a comparison with PBFT in terms of security and performances. Specifically, we conduct a qualitative analysis in terms of the CAP theorem, i.e., by assessing which “two out of three” properties between *consistency*, *availability* and *partition tolerance* can be assured at the same time. Finally, we assess performances in terms of message exchange. The analysis assumes an *eventually synchronous network* and the presence of Byzantine nodes. The conducted analysis results that PoA algorithms favour availability over consistency, oppositely to what PBFT guarantees. In terms of latency, measured as the number of message rounds required to commit a block, PBFT lies in between Aura and Clique, outperforming the former and being worse than the latter. These results suggest that PoA algorithms are not actually suitable for permissioned blockchains deployed over the Internet, because they do not ensure consistency, and strong data integrity guarantees are usually the reason why blockchain-based solutions are employed. We advocate that PBFT is a better choice in this case, although its performance can be worse than some PoA implementations, thus their application may highly depend from the scenario.

*Contributions.* The content of this chapter has been published at the conference ITASEC18, the contributions can be summarised as follow:

- we detailed two of the most prominence version of PoA algorithms by reversing source code and online documentation;
- we proposed an approach to evaluate consensus algorithms through the CAP;
- we conducted such an analysis on the two PoA algorithms as well as on PBFT;
- we analysed performances of PoA and PBFT.

*Chapter Structure.* Section 3.1 introduces background concepts and comments the closest related work. Section 3.2 introduces the PoA consensus schema. Section 3.3 analyses PoA algorithms with respect to ensured guarantees and performance. Finally, Section ?? concludes the work.

## 3.1 Related Work

The most prominent consensus schema for consensus in distributed computing is so-called Byzantine Fault Tolerant (BFT). Protocols of such type are able to tolerate arbitrarily subverted nodes trying to hinder the achievement of an consistent agreement as we detailed in Chapter 2.

The *Practical Byzantine Fault Tolerance* (PBFT) [13] is one of the most well-established BFT algorithms. However many other BFT algorithms have been proposed, mainly for

improving PBFT performance; among others we can cite *Q/U*, *HQ*, *Zyzyva*, *Aardvark*, see the survey in [56] for further details of each solution.

The wide interest on blockchain has prompted substantial research efforts on distributed consensus schema, specifically towards new ad-hoc BFT ones. In [41] the author reviews well-known families of consensus algorithms for both permissionless and permissioned blockchains. This includes Proof-of-Work (PoW), Proof-of-Stake (PoS), Delegated Proof-of-Stake (DPoS), Proof-of-Activity (PoW/PoS-hybrid), Proof-of-Burn (PoB), Proof-of-Validation (PoV), Proof-of-Capacity (PoC or Proof-of-Storage), Proof-of-Importance (PoI), Proof-of-Existence (PoE), Proof-of Elapsed Time (PoET), Ripple Consensus Protocol and Stellar Consensus Protocol (SCP). Although each algorithm is briefly described, they lack of any form of analysis in presence of Byzantine nodes under an eventual synchronous model.

Understanding the most appropriate consensus algorithm among the plethora before is a challenging task that a few works have tried to tackle. For instance, Sankar et al. investigates in [50] the main differences between SCP and consensus algorithms employed in R3 Corda and Hyperledger Fabric. Similarly, Mingxiao et al. extensively compare in [42] performances and security of PoW, PoS, DPoS, PBFT and Raft. While, Tuan et al. propose in [19] a practical benchmark for blockchain, named Blockbench, to systematically compare performances, scalability and security of multiple blockchain systems.

From a more formal perspective, Vukolić compares in [57] PoW with BFT-like approaches introducing the distinguishing property of *consensus finality*: the impossibility of reaching consensus without fully distributed agreement. In blockchain's jargon, it amounts to the impossibility of having forks. As expected, PoW does not enjoy consensus finality (as forks can happen), while all BFT-like approaches does (all parties reach agreement before consensus).

More related to permissioned blockchain, Cachin and Vukolić propose in [12] a thorough analysis of most-known permissioned systems and their underlying consensus algorithms in term of safety and liveness guarantees under eventual synchrony assumption. This work introduces for the first time a structured comparison among consensus algorithms, but it overlooks consistency and availability guarantees ensured by their usage; most of all it does not address PoA.

To sum up, none of the aforementioned works discuss the PoA consensus family. To the best of our knowledge, we believe this is the first work tackling the analysis of blockchain consensus algorithms from the perspective of the CAP theorem.



## 3.2 Proof-of-Authority Consensus

*Proof of Authority* (PoA) is a family of consensus algorithms for permissioned blockchain whose prominence is due to performance increases with respect to typical BFT algorithms; this results from lighter message exchanges. PoA was originally proposed as part of the Ethereum ecosystem for private networks and implemented into the clients *Aura* and *Clique*.

PoA algorithms rely on a set of  $N$  trusted nodes called the *authorities*. Each authority is identified by a unique *id* and a majority of them is assumed honest, namely at least  $N/2 + 1$ . Consensus in PoA algorithms relies on a *mining rotation* schema, a widely used approach to fairly distribute the responsibility of block creation among authorities [29, 26, 3]. Time is divided into *steps*, each of which has an authority elected as mining leader<sup>1</sup>.

The two PoA implementations work quite differently: both have a first round where the new block is proposed by the current leader (*block proposal*); then *Aura* requires a further round (*block acceptance*), while *Clique* does not. Figure 3.1 depicts the message patterns of *Aura* and *Clique*, which will be detailed in next subsections.

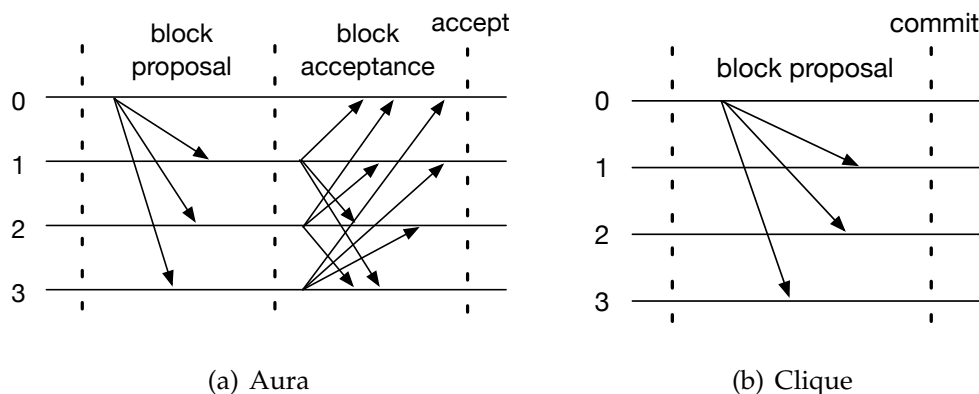


Figure 3.1: Message exchanges of *Aura* and *Clique* PoA for each step. In this example there are 4 authorities with *id* 0,1,2,3. The leader of the step is the authority 0.

### 3.2.1 Aura

*Aura* (Authority Round) [4] is the PoA algorithm implemented in Parity [55], the Rust-based Ethereum client. The network is assumed to be synchronous and all authorities to be synchronised within the same UNIX time  $t$ . The index  $s$  of each step is deterministically computed by each authority as  $s = t / \text{step\_duration}$ , where *step\_duration* is a constant

<sup>1</sup>For the cognitive, as there is no hash-based procedure like PoW, the consensus process is more appropriately called *minting*. However, for the sake of presentation, we will continue using the wording mining.

determining the duration of a step. The leader of a step  $s$  is the authority identified by the id  $l = s \bmod N$ .

Authorities maintain two queues locally, one for transactions  $Q_{txn}$  and one for pending blocks  $Q_b$ . Each issued transaction is collected by authorities in  $Q_{txn}$ . For each step, the leader  $l$  includes the transactions in  $Q_{txn}$  in a block  $b$ , and broadcasts it to the other authorities (*block proposal* round in Figure 3.1(a)). Then each authority sends the received block to the others (round *block acceptance*). If it turns out that all the authorities received the same block  $b$ , they *accept*  $b$  by enqueueing it in  $Q_b$ . Any received block sent by an authority not expected to be the current leader is rejected. The leader is always expected to send a block, if no transaction is available then an empty block has to be sent.

If authorities do not agree on the proposed block during the block acceptance, a voting is triggered to decide whether the current leader is malicious and then kick it out. An authority can vote the current leader malicious because (i) it has not proposed any block, (ii) it has proposed more blocks than expected, or (iii) it has proposed different blocks to different authorities. The voting mechanism is realised through a *smart contract*, and a majority of votes is required to actually remove the current leader  $l$  from the set of legitimate authorities. When this happens, all the blocks in  $Q_b$  proposed by  $l$  are discarded. Note that leader misbehaviours can be caused by benign faults (e.g., network asynchrony, software crash) or Byzantine faults (e.g., the leader has been subverted and behaves maliciously on purpose).

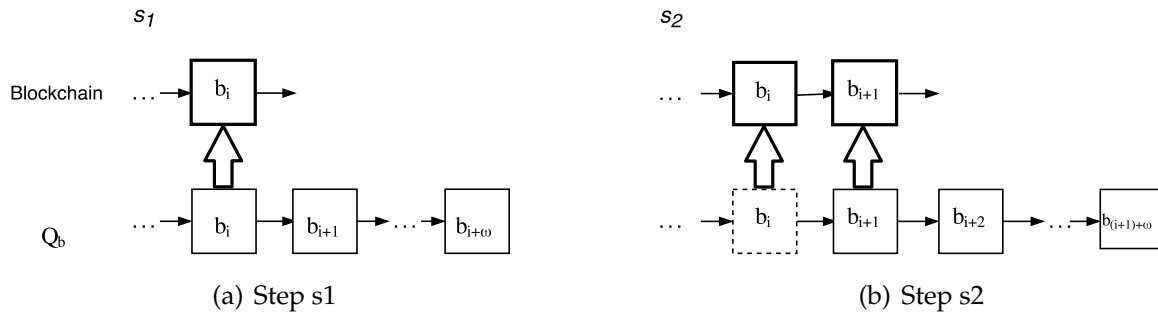


Figure 3.2: Aura finality

A block  $b$  remains in  $Q_b$  until a majority of authorities propose their blocks, then  $b$  is committed to the blockchain. With a majority of honest authorities, this mechanism should prevent any minority of (even consecutive steps) Byzantine leaders to commit a block they have proposed, thus it guarantee consensus finality (Figure 3.2). Indeed any suspicious behaviour (e.g., a leader proposes different blocks to different authorities) triggers a voting where the honest majority can kick the current leader out, and the blocks they have proposed can be discarded before being committed.

Figure 3.2 shows how Aura guarantees finality on two consecutive steps. At step  $s_1$  on the blockchain are committed blocks till  $b_i$ , whereas blocks  $b_{i+1} \dots b_{i+\omega}$  are pending. The block  $b_i$  can be committed, since  $\omega = \frac{k}{2} + 1$  further blocks have been proposed after  $b_i$ , and thus the block  $b_i$  achieved finality. Likewise, at step  $s_2$  the block  $b_{i+1}$  achieves finality as the queue contains  $\omega$  further blocks.

Summarising, the algorithm works as follows:

1. Nodes collect transactions in the transaction queue  $Q_{txn}$ ;
2. For each step, the primary node seals and broadcasts a block to the network;
3. Nodes who receive a new block, verify its correctness and append it in the queue  $Q_b$ , waiting for finalisation;
4. When an epoch expires, each node examines the subset of all unfinalised blocks from  $Q_b$ . If a chain of waiting blocks has been finalised, then it is committed to the blockchain. Each committed transaction is removed from the queue  $Q_{txn}$ ;

### 3.2.2 Clique

*Clique* [14] is the PoA algorithm implemented in Geth [21], the GoLang-based Ethereum client. The algorithm proceeds in *epochs* which are identified by a prefixed sequence of committed blocks. When a new epoch starts, a special *transition block* is broadcasted. It specifies the set of authorities (i.e., their ids) and can be used as snapshot of the current blockchain by new authorities needing to synchronise.

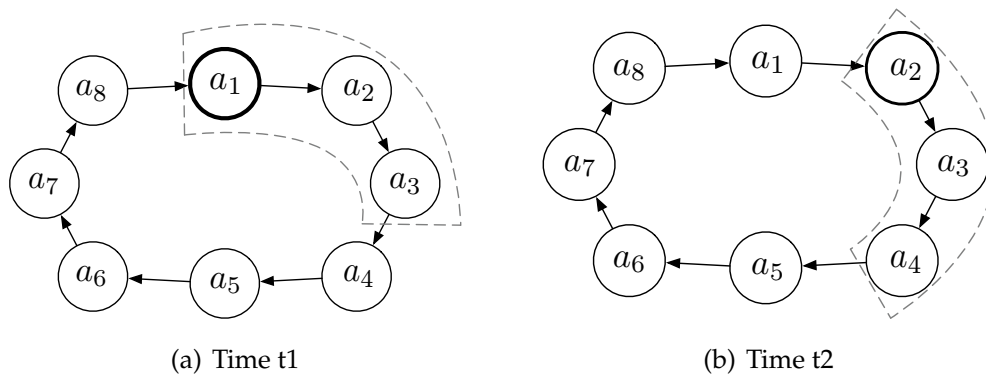


Figure 3.3: Selection of authorities allowed to propose blocks in Clique.

While Aura is based on UNIX time, Clique computes the current step and related leader using a formula that combines the block number and the number of authorities. Most of all, in addition to the current leader, other authorities are allowed to propose

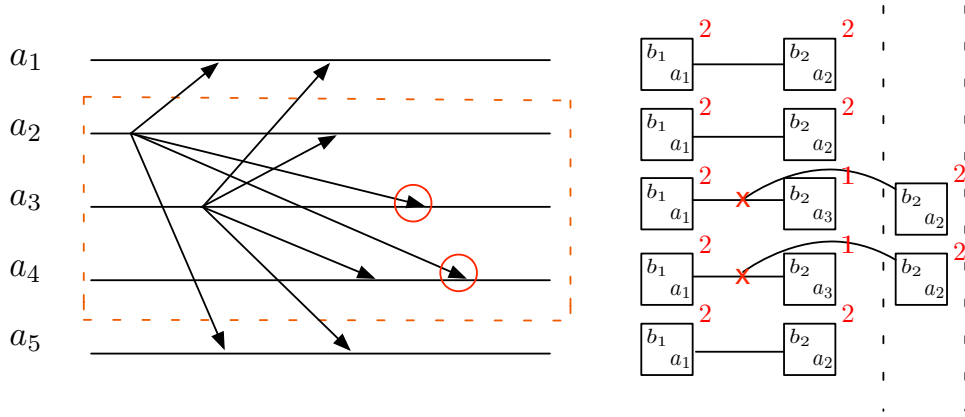


Figure 3.4: A fork occurring in Clique. Authority  $a_4$  has the block proposed by  $a_3$  as second block, while  $a_5$  has the block proposed by  $a_2$ . Eventually,  $a_4$  replaces the block proposed by  $a_3$  with that proposed by  $a_2$  because the latter has a higher score.

blocks in each step. To avoid that a single Byzantine authority could wreak havoc the network by imposing a sheer number of blocks, each authority is only allowed to propose a block every  $N/2 + 1$  blocks. Thus, at any point in time there are at most  $N - (N/2 + 1)$  authorities allowed to propose a block. Similarly to before, if authorities act maliciously (e.g., by proposing a block when they are not allowed) they can be voted out. Specifically, a vote against an authority can be casted at each step and if a majority is reached the authority is removed from the list of legitimate authorities.

As more authorities can propose a block during each step, forks can occur. However, fork likelihood is limited by the fact that each non-leader authority proposing a block delays its block by a random time, hence the leader block is likely to be the first received by all the authorities. If forks happen, the GHOST protocol [60] is used, which is based on a block scoring approach: leaders' blocks have higher scores, thus ensuring that forks will be eventually solved.

Figure 3.3 shows two consecutive steps and how current leader and authorities allowed to propose blocks change. There are  $N = 8$  authorities, hence  $N - (N/2 + 1) = 3$  authorities allowed to propose a block at each step, with one of them acting as leader (the bold node in Figure 3.3). In Figure 3.3(a), the  $a_1$  is the leader while  $a_2$  and  $a_3$  are allowed to propose blocks. In Figure 3.3(b),  $a_1$  is not allowed anymore to propose a block (it was in the previous step, so it has to wait  $N/2 + 1$  steps), while  $a_4$  is now authorised to propose and  $a_2$  is the current leader.

Regarding the message exchange (see Figure 3.1(b)), at each step the leader broadcasts a block and all the authorities directly commit it to the chain. By way of example, Figure 3.4 depicts a step where the leader authority  $a_2$  proposes a new block, as well as the other allowed non-leader authority  $a_3$ . The former block precedes the latter in the views

of  $a_1$  and  $a_5$ , while the opposite occurs for  $a_4$  and  $a_3$ . The resulting fork (see right-hand side of Figure 3.4) is easily detected by each authority when the next block is received, as it references a previous block not at disposal of the authority. By relying on the scoring mechanisms (i.e. blocks proposed by leaders win), the GHOST protocol resolves the forks.

### 3.3 Comparison of Aura, Clique and PBFT

Previous PoA algorithms are here compared with PBFT, first in terms of consistency and availability properties via the CAP Theorem (Section 3.3.1), then of performance (Section 3.3.2).

#### 3.3.1 Consistency and Availability Analysis based on CAP Theorem

Referencing the CAP Theorem introduced in Section 2.4, we refine the definitions of CAP properties in the context of permissioned blockchains in order to delving into the CAP-based analysis of the considered algorithms. Moreover we consider a system deployed over the Internet, hence subjected to unforeseeable network delays of variable durations. In the following, we then assume an eventually synchronous network model where messages can be delayed among correct nodes, but eventually the network starts behaving synchronously and messages will be delivered (within a fixed but unknown time bound, see Section 2.1). This model is considered appropriate when designing real resilient distributed systems [12].

*Consistency.* A blockchain achieves consistency when forks are avoided. This property, as reported in Section 2.4, is referred to as consensus finality which, in the standard distributed system jargon, corresponds to achieving the *total order* and *agreement* properties of atomic broadcast. The latter is the communication primitive considered as the relevant type of consensus for blockchains [12]. When consistency cannot be obtained, we have to distinguish whether forks are resolved sooner or later (*eventual consistency*) or they are not (*no consistency*).

*Availability.* A blockchain is available if transactions submitted by clients are served and eventually committed, i.e. permanently added to the chain.

*Partition Tolerance.* When a network partition occurs, authorities are divided into disjoint groups in such a way that nodes in different groups cannot communicate each other.

Therefore, an Internet-deployed permissioned blockchain has to tolerate these adverse situations: (i) periods where the network behaves asynchronously; (ii) a (bounded) num-

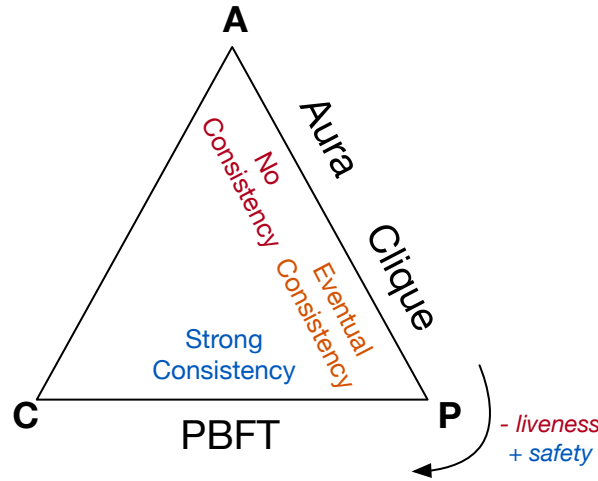


Figure 3.5: Classification of Aura, Clique and PBFT according to the CAP Theorem

ber of Byzantine authorities aiming at hampering availability and consistency. The maximum number of tolerated Byzantine nodes depends on the consensus algorithm:  $N/2$  for PoA,  $N/3$  for PBFT. Since a blockchain must tolerate partitions, hence CA option is not considered, we analyse the algorithms with respect to CP and AP options. The results of this analysis are reported in Figure 3.5 and commented in the following.

### Aura Analysis

Being based on UNIX synch time, authorities' clocks can drift and become out-of-synch. When authorities are distributed geographically over a wide area, resynchronization procedures cannot be effective due to network eventual synchrony. Hence, there can be periods where authorities do not agree on what is the current step and consequently on the current authority in force. Clocks' skews can be reasonably assumed strictly lower than the step duration, which is in the order of seconds, thus we can have short time windows where two distinct authorities are both considered as leaders by two disjoint sets of authorities, say  $\mathcal{A}_1$  and  $\mathcal{A}_2$ . This can critically affect the consistency of the whole system.

Let  $N_1 = |\mathcal{A}_1|$  and  $N_2 = |\mathcal{A}_2|$  (where  $N_1 + N_2 = N$  and  $N$  is an odd number) be the number of authorities in the two sets, respectively. We have a majority of authorities, say  $\mathcal{A}_1$ , agreeing on who is the current leader. This leads to a situation as depicted in Figure 3.6: authorities in  $\mathcal{A}_1 = \{a_1, a_3, a_5\}$  see steps slightly out of phase with respect to the authorities in  $\mathcal{A}_2 = \{a_2, a_4\}$ . Indeed, the time windows coloured in grey are those where  $\mathcal{A}_1$  disagrees with  $\mathcal{A}_2$  on who is the current leader. During time window  $W_1$ ,  $a_2$  considers itself the leader and sends a block to the other authorities.  $a_2$  is believed to be the leader by the authorities in  $\mathcal{A}_2$  but not by those in  $\mathcal{A}_1$ , hence the former authorities

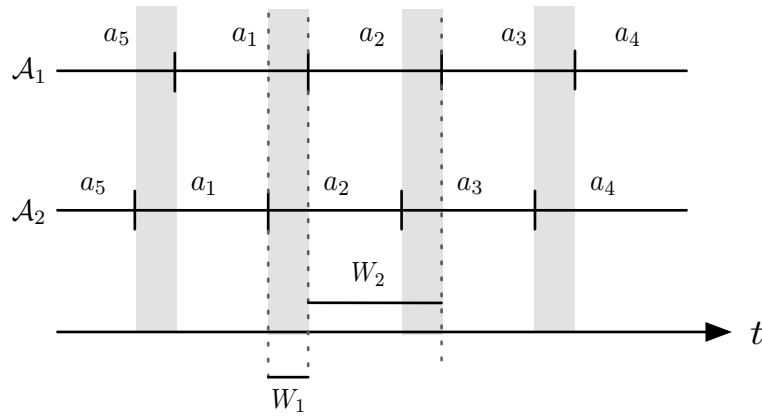


Figure 3.6: Example of out-of-synch authorities in Aura (each step for a set of authorities is labelled by the expected leader)

accept its block while the latter ones reject it. During the time window  $W_2$ , authorities in  $\mathcal{A}_1$  expect  $a_2$  to send a block but this does not occur because it has already sent its block for the current step: at the end of  $W_2$  authorities in  $\mathcal{A}_1$  will vote  $a_2$  as malicious and being a majority they will force  $a_2$  to be removed. Therefore, all the remaining authorities in  $\mathcal{A}_2$  will be eventually voted out one by one analogously. As  $\mathcal{A}_2$  is a minority, its authorities are voted out in a number of steps lower than that required to commit enqueued blocks (see Section 3.2.1), hence there will be no different views of the chain and the consistency is preserved. It is to note that in this case all the authorities are honest.

The same situation can affect consistency when authorities are Byzantine. Let us consider a scenario where there are  $B$  malicious authorities, all in  $\mathcal{A}_1$  and, differently from before, they do not vote against authorities in  $\mathcal{A}_2$ . If  $B \geq N_1 - N/2$ , then a majority is not reached to vote out authorities in  $\mathcal{A}_2$ , hence the blocks they have proposed achieve finality and are committed to their local chains. This causes a fork that is never resolved: if authorities are not voted out, their blocks are considered as valid and part of the chain. Therefore, a minority of Byzantine authorities is sufficient to realise this attack and causes *no consistency* in the system.

Indeed, let us consider a set  $\mathcal{S}$  with an odd number of elements  $N = 2K + 1$ , and a partition of such set in two non-empty subsets  $\mathcal{S}_1$  and  $\mathcal{S}_2$  with cardinality  $N_1$  and  $N_2$ , respectively, such that  $\mathcal{S}_1$  is a majority and  $\mathcal{S}_2$  a minority, i.e.,

$$K + 1 \leq N_1 \leq 2K \tag{3.1}$$

$$1 \leq N_2 \leq K$$

$$N_1 + N_2 = N$$

We want to prove that it suffices to remove a minority<sup>2</sup> of  $B$  elements from  $\mathcal{S}_1$  to make it become a minority. Hence, we want to prove that

$$\exists B \mid N_1 - B \leq K \wedge B \leq K \quad (3.2)$$

*Proof.* Equation 3.2 can be proved by demonstrating that  $N_1 - K \leq K$ . This expression can be written as  $N_1 \leq 2K$ , which is always verified because of Equation 3.1.

Anyway, transactions keep being committed over time regardless of what a minority of Byzantine authorities do. Hence, Aura can be classified as an AP system, with no consistency guarantees.

### Clique Analysis

By design, Clique allows more than one authority to propose blocks with random delays. This permits coping with leaders that could not have sent any block due to either network asynchrony or benign/Byzantine faults. Resulting forks are anyway resolved by the Ethereum GHOST protocol, hence we have *eventual consistency*. On the other hand, as the mining frequency of authorities is bounded by  $\frac{1}{N/2+1}$ , a majority of Byzantine authorities is required to take over the blockchain. This PoA algorithm can thus be classified as AP, with eventual consistency guarantees.

### PBFT Analysis

As long as less than one third of nodes are Byzantine, PBFT has been proved to guarantee consistency, i.e. no fork can occur [13]. Because of the eventual synchrony of the network, the algorithm can stall and blocks cannot reach finality. In this case, consistency is preserved while availability is given up. PBFT can then be easily classified as a CP system according to the CAP theorem.

### 3.3.2 Performance Analysis

The analysis here reported is qualitative and only based on how the consensus algorithms work in terms of message exchanging. The performance metrics usually considered for consensus algorithms are transaction latency and throughput. In the specific case of permissioned blockchains, we measure the latency of a transaction  $t$  as the time between the

---

<sup>2</sup>A minority with respect to the set  $\mathcal{S}$ .



submission of  $t$  by a client and the commit of the block including  $t$ . Contrary to CPU intensive consensus algorithms such as PoW, here we can safely assume that latency is communication-bound rather than CPU-bound, as there is no relevant computation involved. Hence, we can compare the algorithms in terms of the number of message rounds required before a block is committed. Evaluating the throughput at a qualitative level is much more challenging, as it closely depends on the specific parallelisation strategy (e.g., pipelining) employed by each algorithm implementation. Thus, we deem more correct to compare throughput performance by the means of proper experimental evaluations that we plan to carry out as future work.

We assess how many message rounds are required for each algorithm in the normal case, i.e. when no condition occurs that makes any corner case to be executed. For example, for Aura we do not consider the situation when some authorities suspect the presence of subverted nodes and trigger a voting.

In Aura, each block proposal requires two message rounds: in the first round the leader sends the proposed block to all the other authorities, in the second round each authority sends the received block to all the other authorities. A block is committed after a majority of authorities have proposed their blocks, hence the latency in terms of message rounds in Aura is  $2(N/2 + 1)$ , where  $N$  is the number of authorities.

In Clique, a block proposal consists of a single round, where the leader sends the new block to all the other authorities. The block is committed straight away, hence the latency in terms of message rounds in Clique is 1. Such a huge difference between Aura and Clique is due to their different strategies to cope with malicious authorities aiming at creating forks: Aura waits that enough other blocks have been proposed before committing, Clique commits immediately and copes with possible forks after they occur. Clique seems to outperform PBFT too, which takes three message rounds to commit a block.

## Chapter 4

# Benchmark for permissioned blockchain

In the previous chapters we outline the difficulty of comparing the rising BFT-based consensus protocols for blockchain systems. These solutions should guarantee atomic broadcast for transactions order, to the detriment of availability or partition-tolerance. However the evaluation of these algorithms is actually challenging in adversarial environment. Indeed to be secure, a protocol should come out with clear security model and trust assumptions, under which the protocol proceeds as expected guaranteeing safety and liveness properties. Actually most of the modern BFT proposals for blockchain come out without these information, so is difficult to evaluate their employability.

In the information technology is well known that through the experimental evaluation can be only demonstrated the failure of a system [12]. Furthermore, by simulating possible attack scenarios can be measured the resiliency against adversarial environments. Thus, for the evaluation of blockchain systems (for the remainder of the chapter this term will be used interchangeably with 'blockchain') we highlight the need of a general purpose benchmark who measure performances under the assumptions above.

In this chapter we propose the implementation of such a benchmark. We move towards a formalisation of blockchain to define a framework for benchmarking and evaluating these algorithms in a more formal approach. Concerning this, we define the evaluation metrics under which measure performances and security. Specifically, to evaluate security, we identify from the general Byzantine model (Section 2.2) three most significant attacks for a blockchain system to be exploited in the experimental evaluation.

To analyse the fault tolerance behaviour and hence the resiliency of the consensus algorithms, we propose an experimental approach by taking a real blockchain client and making it Byzantine. By monitoring the behaviours at runtime of the systems, we can then evaluate the impact of Byzantine nodes in real blockchain systems.

*Chapter Structure.* Section 4.1 introduces the system model for a permissioned blockchain

and the threat model for the benchmark. Section 4.2 introduces the benchmark and the evaluation metrics for measurements. Finally Section 4.3 illustrate the practical implementation of a Byzantine node in a real blockchain client.

## 4.1 System and Threat Model

In this section is introduced the system model for permissioned blockchain. We assume a *partially synchronous* communication network (see Section 2.1). A peer-to-peer distributed system of authenticated nodes communicate over the network and keep a common state update. This state is essentially a replicated data structure shared in memory between replicas, i.e. *blockchain*. The blockchain is a collection of transactions organised in blocks, it can be referenced as a ledger of blocks where each one is cryptographically linked to the previous by means of its hash.

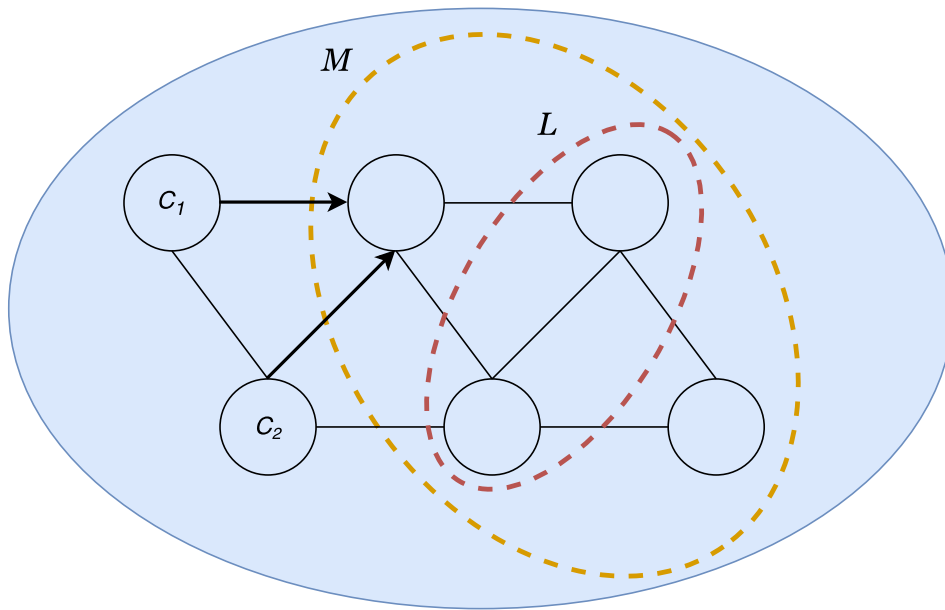


Figure 4.1: Example of blockchain network with six nodes which maintains the ledger. There are four miners of which two possible leaders. There are also two clients which submit transactions (bold arrows). Note that a node of the network can be a miner and a client simultaneously.

A transaction represent an asset exchange between two parties, secured by asymmetrical cryptography, while blocks are essentially collections of transactions with a determined size. In this scenario we identify two class of nodes, *miners* and *clients*. The clients

are nodes in charge to submit transactions, while miners are responsible for accepting correct transactions and collect them in blocks. We define the following property for the transaction validation:

**External Validity (*safety*)** Verifies the correctness of transactions broadcasted through the network from each client. Only valid transactions can be accepted by the protocol and committed to the blockchain. The property has been defined as *external validity* by Cachin et al.[?] and can be formalized as an external predicate  $V$  deterministically computable by every node locally. This property can be formalized as follows[12]:

**Definition 1. (Transaction Validation)** *Given a deterministic predicate  $V$  s.t.  $V \rightarrow \{True, False\}$ , and given two correct nodes of the blockchain  $n_1, n_2$  s.t in atomic broadcast they have committed the same sequence of transactions  $\mu$ . Then  $n_1$  obtain,  $\forall tx \in \mu, V(tx)=True$  iff  $\forall tx \in \mu, n_2$  obtain  $V(tx)=True$*

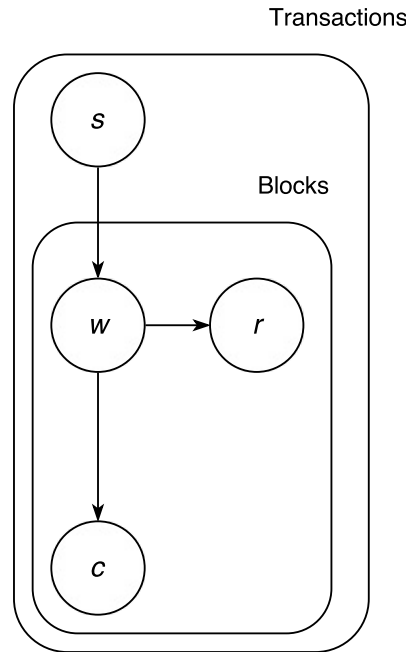


Figure 4.2: Transactions are Submitted by clients to the network. Then, once these transactions are collected in blocks, they change the state with the relative block. Each block waits for consensus in a Wait state or can be Refused. When the miners reach consensus a block will be Committed.

After the validation, each miner keep a list of pending transactions and create the relative blocks to be proposed as next of the blockchain. While a block, and so the included transactions, is not committed immediately on the blockchain, initially it is kept in a waiting state. Thus, to guarantee atomicity on the transactions order, a consensus

protocol is run between miners to establish the next block to be committed in the shared state. We identify a special class of nodes, i.e. *leaders*, which are miners authorised to propose a block at a certain point in time. Figure 4.1 illustrates an example of our modelled blockchain network.

When miners achieve consensus the current leader propose a new block. This can be accepted and committed to the blockchain by all the nodes of the network, or refused (Figure 4.2 is illustrated the life-cycle of transactions and blocks).

The blockchain system can be splitted into two layers. Miners of the network executes, in a first layer, an algorithm for the collection of transactions submitted by clients and for the management of blocks. This level is supported by a second layer which runs the consensus protocol. This modules are connected each other, granting the execution of transactions and the update of the shared ledger.

To formalise the functioning of the first layer we model the scenario described above as follows.

Let  $B$  be a blockchain system,  $B := \langle \mathcal{M}, \mathcal{L}, \mathcal{C}, \Gamma, \Omega, \alpha, \beta \rangle$  where:

- $\mathcal{M}$  is the set of miners
- $\mathcal{L}$  is the set of possible leaders,  $\mathcal{L} \subseteq \mathcal{M}$
- $\mathcal{C}$  is the set of clients
- $\Gamma$  is the set of transactions
- $\Omega$  is the set of blocks
- $\alpha$  is the consensus algorithm adopted, it determine:
  1. when a new leader is elected from miners:
    - if the current leader is byzantine (PBFT);
    - at each block (Mining Rotation);
  2. the subset of miners which can be leader:
    - all miners,  $\mathcal{L} = \mathcal{M}$  (PBFT);
    - a subset of miners,  $\mathcal{L} \subseteq \mathcal{M}$  (Mining Rotation).
- $\beta : (\Gamma \times \mathcal{M}) \rightarrow \Omega$ , is a function executed by miners, which takes in input the submitted transactions and return a new block.

Referencing to the semantics defined, in the Algorithm 1 we propose the standardisation of the first layer, where miners collect transactions and prepare blocks. This algorithm rely on perfect links for communications and on a reliable broadcast module to guarantee the correct delivery of transaction. Moreover the algorithm rely on a consensus protocol module to manage the leaders' proposals.

### 4.1.1 Blockchain Properites

The atomic broadcast is the most feasible consensus in permissioned blockchain. However, besides its fundamental properties (see Section 2.3), the peculiarities of blockchain require the analyse of additional properties of interest. We list them in the following.

Given a blockchain system with a set of nodes such that:  $M := \{m_1, \dots, m_i\}$  miners, and  $C := \{c_1, \dots, c_j\}$  clients, we identify as pending transaction  $tx \in \Gamma$  and a pending block  $b \in \Omega$  s.t. these properties are valid:

#### Integrity (*safety*)

- *No duplication*:  $tx$  is not committed to the blockchain more than once.
- *No creation*: If  $tx$  is committed to the blockchain, than it was previously broadcasted by a correct miner  $m_j$ .

**Validity (*safety*)** If a correct node commits a transaction  $tx$  to the blockchain, in a block  $b$ , then  $tx$  is committed, in the same block  $b$ , by every correct node.

**Finality (*safety*)** In BFT-based blockchain, *consensus finality* is the possibility of encounter forks during the chain construction. Roughly speaking finality is reached if all valid blocks appended to the blockchain at some point in time, will be never removed from the blockchain (see Section 2.4 for a formal definition).

**Termination (*liveness*)** For every transaction  $tx$ , if at least one correct node commits  $tx$ . then all correct nodes (eventually) commit  $tx$ .

### 4.1.2 Threat Model

Assessing the security of blockchain consensus protocols is a challenging issue due to the variety of attacks faced by blockchain system. Defining an attacker model is a first step toward a comprehensive experimental evaluation of the resiliency of blockchain consensus protocols.

We model an attacker by identifying three more significant attacks for a blockchain system. As an approach, we started from the Byzantine failure model [17] (Section 2.2), where failures may occur or nodes may not behave in the proper manner because subverted. Specifically, three classes of failures are there identified: *omission failure*, *timing failure* and *Byzantine failure*. These classes are also specialised for sub-class of authenticated protocols, likewise digitally signing a block or a transaction in a blockchain system.

**Attacker model** We assume a blockchain system formed by  $n$  independent nodes participating to the consensus algorithm with at most  $f < \frac{n}{k}$ , ( $k = 2, 3, \dots$ ) Byzantine faulty nodes. We identify three attacks:

- (i) *leader misbehaviour*, a timing fault due to a miner broadcasting inconsistent blocks at the same time;
- (ii) *forgery*, a Byzantine fault of a miner trying to forge fake transactions and/or blocks;
- (iii) *Denial of Service*, an omission fault due to a miner avoiding the signing or broadcasting of a transaction.

An attacker aims at compromising *safety* and *liveness* of an algorithm by exploiting these attacks. Specifically, by referring to the evaluation properties reported before, we have that (i) leader misbehaviour violates finality; (ii) forgery violates Integrity, validity and external validity; (iii) Denial of Service violates termination.

## 4.2 Benchmark

In this section we define a general methodology to standardise the evaluation of permissioned blockchain systems. To benchmark such a system we should focus on the underling consensus algorithm. Usually for the evaluation metrics of consensus the most relevant properties to be considered are (i) *performance* and, (ii) *security*.

To evaluate performances we consider throughput and latencies of transactions and, scalability. Furthermore, because blockchains are *asynchronous services* [19], transactions submitted to the system are not processed immediately. To measure the latency of transactions we refer to the time between the submission and the commit of the block including the transaction.

To evaluate the impact of the attacks listed in the system model, Section 4.1.2, we refer to the following properties.

## Evaluation Metrics

- *Throughput*: measured as the number of transactions successfully committed per second.
- *Latency*: measured as the time between the transaction submission and the commit of the relative block.
- *Scalability*: measured as the changes of throughput and latency, increasing the number of nodes of the network.
- *Safety*: identified with the Integrity, Validity and Finality properties listed in Section 4.1.1
- *Liveness*: identified with the Termination property listed in Section 4.1.1

For each attack we propose to measure a sort of *index-of-tolerance* which identify how the threat impacts the metrics, i.e. a performance breakdown or violation of safety and liveness.

## 4.3 Byzantine client for blockchain

For evaluation purposes, whereas possible, we illustrate the best practice to integrate a Byzantine client in a real blockchain framework. Most of these blockchain systems follow a similar structure which covers the same (or similar) actors.

The Byzantine client we propose to integrate should simulate the attacker model given in Section 4.1.2. Each attack impacts some properties and functions of the architectural implementation.

We list below the best practice to follow for achieving this task and identify the correct functions to modify in a blockchain framework.

1. Use the system logs to understand the the behaviour of the relevant actors. Specifically we suggest to use the trace level of the logger and focus on (i) transactions, (ii) miners and, (iii) consensus engine;
2. Allow a blockchain client to run as Byzantine from the CLI. From the source code add the flag option to the CLI manager;
3. Once identified the miner options, update their configurations to be prepared for a Byzantine set-up. Indeed this functions usually manage the miner behaviours, i.e.



the block creation, the block sealing and dispatching. Likewise a Byzantine miner must be provided of specific functions to simulate the attacks above;

4. Update the consensus engine module which interact with the functions at 3. This module needs to be prepared at Byzantine behaviours and, because it manage the achievement of consensus, new functions for the possible attacks management are required.

Currently we have implemented a demo of this Byzantine client inside Parity Ethereum's client. We simulate the denial-of-service attack forcing a miner to ignore the block sealing procedure. For the implementation details we refer to the Appendix A.

**Algorithm 1** First layer blockchain**Uses:**PerfectPointToPointLink  $pl$ ;ReliableBroadcast  $rb$ ;ConsensusProtocol  $\alpha$ .

```

1: upon event  $\langle Init \rangle$  do
2:    $submitted := 0$ 
3:    $waiting := 0$ 
4:    $refused := 0$ 
5:    $committed := 0$ 
6:    $BLOCK\_SIZE := k$  ▷ Predefined block size
7: upon event  $\langle rb, Deliver|c, tx \rangle$  do
8:   if  $external\_validity(tx)$  then
9:      $submitted := submitted \cup \{tx\}$ 
10:  else
11:     $trigger \langle pl, Refused|c, tx \rangle$ 
12: upon  $|submitted| == BLOCK\_SIZE$  do
13:    $block\_tx = submitted$ 
14:    $block = \beta(block\_tx)$ 
15:    $submitted := submitted \setminus \{block\_tx\}$ 
16:    $waiting := waiting \cup \{block\}$ 
17:    $trigger \langle rb, Broadcast|self, block \rangle$ 
18: upon event  $\langle rb, Deliver|m, block \rangle$  do
19:   if  $verify(block)$  then ▷ Function for block validity verification
20:      $trigger \langle \alpha, Propose|m, block \rangle$ 
21:   else
22:      $refused := refused \cup \{block\}$ 
23:      $waiting := waiting \setminus \{block\}$ 
24: upon event  $\langle \alpha, Deliver|block, state \rangle$  do
25:   if  $state == accepted$  then
26:      $committed := committed \cup \{block\}$ 
27:      $waiting := waiting \setminus \{block\}$ 
28:   else if  $state == refused$  then
29:      $refused := refused \cup \{block\}$ 
30:      $waiting := waiting \setminus \{block\}$ 
31: function  $\beta(block\_tx)$  return  $\Omega$ 
32:   for all  $tx \in block\_tx$  do
33:      $block := block \cup \{tx\}$ 
34: return  $block$ 

```

# Chapter 5

## Conclusions and Future Directions

In this thesis we approached some techniques to analyse the permissioned blockchain consensus protocols. Specifically we studied the security requirements relative to safety and liveness properties and we analyse their performance guarantees. The thesis brings two main contributions. The first contribution is a qualitative analysis of two consensus algorithms against CAP (consistency, availability and partition tolerance) properties. The second contribution is a formalisation of permissioned blockchains, for defining a benchmarking framework and evaluating the consensus algorithms with a more formal approach.

In the first contribution we derive the functioning of two prominent consensus algorithms for permissioned blockchains based on the PoA paradigm, namely Aura and Clique. We provide a qualitative comparison of them with respect to PBFT in terms of consistency, availability and performance, by considering a deployment over the Internet where the network is realistically modelled as eventually synchronous rather than synchronous. By applying the CAP Theorem, we claim that in this setting PoA algorithms can give up consistency for availability when considering the presence of Byzantine nodes. This can prove to be unacceptable in scenarios where the integrity of transactions has to be absolutely kept (which is likely to be the actual reason why a blockchain-based solution is used). On the other hand, PBFT keeps the blockchain consistent at the cost of availability, even when the network behaves temporarily asynchronously and Byzantine nodes are present; this behaviour is much more desirable when data integrity is a priority. Despite one of the most praised advantages of PoA algorithms is their performance, our qualitative analysis shows that in terms of latency the expected loss of PBFT is bounded, and can be offset by the gain in consistency guarantees.

In the second contribution we detailed a formal model for permissioned blockchain. Specifically we outline the most general model for such a scenario, to be as feasible as

---

possible in real context. We assume a partially synchronous network which is subject to the most general Byzantine fault model. Thus we profile a possible attacker model that can be exploited over a blockchain network. To measure security, we assess the evaluation metrics in correct behaviour and under the attacks listed in the system model, Section 4.1.2. Specifically we propose to evaluate the impact of the attacks to the specified evaluation metrics, and for this purpose we integrate a first prototype of a Byzantine client node in a real blockchain, i.e. Parity.

As future work, we plan to deepen the analysis of PoA algorithms by engaging further reverse engineering tasks and thorough experimental evaluations. The final goal is to validate and possibly revise our claims on the availability and consistency guarantees of PoA and PBFT, by implementing the adverse scenarios we envisioned in Section 3.3.1. Furthermore with the specified benchmark guidelines, we want to create a real benchmarking architecture to collect performance measurements, both transaction latency and throughput, and to test scalability with respect to varying input transaction rates and number of nodes/authorities. For this purpose we aim to experiment the consensus technologies by testing the impact of the Byzantine client in consensus algorithms and outline an evaluation metric for security, i.e. *index-of-tolerance*, to measure how a specific algorithm is resilient to an adversarial context.

*"Life is what happens to you while you are making other plans."*

John Lennon

# Appendices

# Appendix A

## Parity Byzantine client

We propose the integration of a Byzantine client in Parity blockchain. It is the Ethereum's client based on the programming language Rust. Beyond the possibility of create a proper client on the public Ethereum blockchain, Parity offers the possibility to realise private permissioned blockchain based on lighter consensus engines. We integrate our client following the steps in Section 4.3. Here we show the implementation details for each of the steps.

First of all, once identified the flow of transactions and miners we integrate the possibility of run a node as Byzantine by updating the configurations of the CLI. More precisely we add the boolean flag `byzantine` by updating the file at `~ parity/cli/mod.rs` as follows.

```
...
// -- Sealing/Mining Options
flag_stratum: bool = false,
    or |c: &Config| Some(c.stratum.is_some()),
flag_stratum_interface: String = "local",
    or |c: &Config| otry!(c.stratum).interface.clone(),
flag_stratum_port: u16 = 8008u16,
    or |c: &Config| otry!(c.stratum).port.clone(),
flag_stratum_secret: Option<String> = None,
    or |c: &Config| otry!(c.stratum).secret.clone().map(Some),
flag_byzantine: bool = false,
    or |c: &Config| otry!(c.mining).byzantine.clone()
...
```

Listing A.1: Byzantine Flag integration.

```

...
struct Mining {
    author: Option<String>,
    engine_signer: Option<String>,
    force_sealing: Option<bool>,
    reseal_on_uncle: Option<bool>,
    reseal_on_txs: Option<String>,
    reseal_min_period: Option<u64>,
    ...
    ...
    tx_queue_strategy: Option<String>,
    tx_queue_ban_count: Option<u16>,
    tx_queue_ban_time: Option<u16>,
    remove_solved: Option<bool>,
    notify_work: Option<Vec<String>>,
    refuse_service_transactions: Option<bool>,
    byzantine: Option<bool>,
}
...

```

Listing A.2: Byzantine Flag for Mining options.

Then to update the miners configurations we load the byzantine flag in the `miner_options()` functions at line 517 of the file `~parity/configuration.rs` adding the following option:

```
byzantine: self.args.flag_byzantine,
```

hence we add the required miner configurations by editing the file at `~ethcore/src/miner/miner.rs` as follows.



```
...
pub struct MinerOptions {
    pub new_work_notify: Vec<String>,
    /// Byzantine node.
    pub byzantine: bool,
    /// Force the miner to reseal, even when nobody has asked for work
    pub force_sealing: bool,
    ...
    ...
    pub tx_queue_gas_limit: GasLimit,
    pub tx_queue_banning: Banning,
    pub refuse_service_transactions: bool,
}
...
```

Listing A.3: Miner behaviour configurations.

```
...
impl Default for MinerOptions {
    fn default() -> Self {
        MinerOptions {
            new_work_notify: vec![],
            byzantine: false,
            force_sealing: false,
            ...
            ...
            tx_queue_banning: Banning::Disabled,
            refuse_service_transactions: false,
        }
    }
}
...
```

Listing A.4: Miner default behaviour configurations.

```

...
fn is_byzantine(&self) -> bool {
    self.options.byzantine || !self.notifiers.read().is_empty()
}
...
...
/// Attempts to perform internal sealing (one that does not require
    work) and handles the result depending on the type of Seal.
fn seal_and_import_block_internally(&self, chain: &
    MiningBlockChainClient, block: ClosedBlock) -> bool {
    if !block.transactions().is_empty() || self.forced_sealing() ||
    Instant::now() > *self.next_mandatory_reseal.read() {
        trace!(target: "miner", "seal_block_internally: attempting
            internal seal.");

        // ADD IF CONDITION FOR match with byzantine (controll with
        function self.is_byzantine())
        let seal_function = if self.is_byzantine() {
            self.engine.generate_seal_byzantine(block.block())
        } else {
            self.engine.generate_seal(block.block())
        };

        ...
        ...

```

Listing A.5: Byzantine function notifier and If statement for Byzantine client.

Once implemented the miner options we update the consensus engine, to simulate the DoS attack. Because Parity provide the possibility to integrate different consensus engines we firstly add the generic function to the interface (`~ethcore/src/engines/mod.rs`) and then we implement this function in our reference consensus, i.e. Aura Proof-of-Authority.

```
...
/// Byzantine seal. No blocks will be sealed.
/// Edited by @StefanoDeAngelis.
fn generate_seal_byzantine(&self, block: &ExecutedBlock) -> Seal {
    if !self.can_propose.load(AtomicOrdering::SeqCst) { return Seal::
        None; }

    let header = block.header();
    let step = self.step.load();

    let active_set;
    ...
    ...
    if is_step_proposer(validators, header.parent_hash(), step, header
        .author()) {
        trace!(target: "engine", "generate_seal: Byzantine proposer for
            step {}. No block sealed.", step);
    } else {
        trace!(target: "engine", "generate_seal: {} not a proposer for
            step {}.",
            header.author(), step);
    }
    Seal::None
}
...
```

Listing A.6: Byzantine sealing function. Exploitation of DoS attack.

# Appendix B

## Consensus Finality

*Consensus Finality* guarantees strong consistency on blockchain among replicas. Essentially it describes the possibility of having forks. In distributed systems is proved that for guaranteeing strong consistency two properties are required: *agreement* and *total order*. For demonstrating that finality strictly implies strong consistency, we reproduce two scenarios of forks which differently violates agreement and total order.

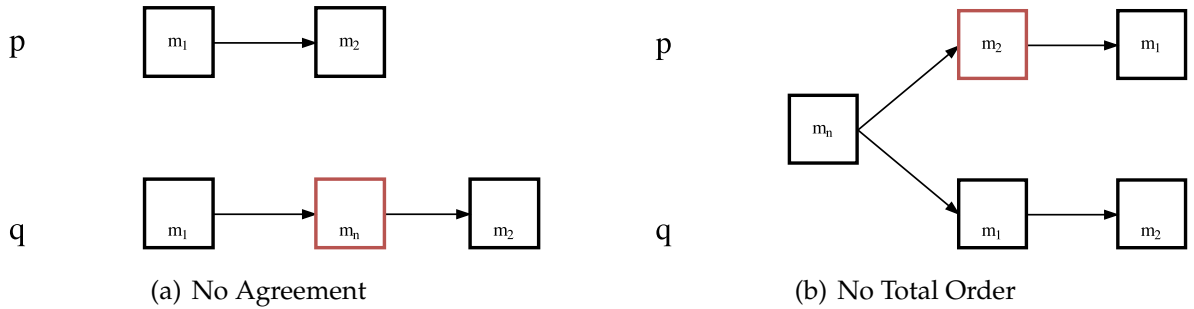


Figure B.1: Fork scenarios that violate agreement and total order.

We consider two correct nodes  $p$  and  $q$ , which hold a blockchain. In Figure B.1(a) agreement is not guaranteed because  $q$  have different blocks to the blockchain with respect to  $p$ . Indeed Figure B.1(b) shows a case in which total order is violated, nodes have the same blocks but with different order. In both scenarios there is a fork between the blockchains of the two participants that violate consequently the finality property. Therefore we can define the following theorem:

**Theorem 1.** *A blockchain system achieve finality if and only if agreement and total order properties are guaranteed.*

**Corollary 1.** *If a blockchain system do not achieve finality, it can only guarantee eventual consistency.*

# List of Figures

1.1	<i>Blockchain representation. Blocks reference each other by the hash of the previous</i>	4
1.2	<i>Bitcoin Public Ledger.</i>	7
1.3	<i>Proof-of-Work as a computational puzzle to solve a block</i>	8
1.4	<i>Block tree where the longest chain is not the one selected by the GHOST protocol.</i>	11
2.1	<i>Byzantine failures model.</i>	16
3.1	<i>Message exchanges of Aura and Clique PoA for each step. In this example there are 4 authorities with id 0,1,2,3. The leader of the step is the authority 0.</i>	24
3.2	<i>Aura finality</i>	25
3.3	<i>Selection of authorities allowed to propose blocks in Clique.</i>	26
3.4	<i>A fork occurring in Clique. Authority <math>a_4</math> has the block proposed by <math>a_3</math> as second block, while <math>a_5</math> has the block proposed by <math>a_2</math>. Eventually, <math>a_4</math> replaces the block proposed by <math>a_3</math> with that proposed by <math>a_2</math> because the latter has a higher score.</i>	27
3.5	<i>Classification of Aura, Clique and PBFT according to the CAP Theorem</i>	29
3.6	<i>Example of out-of-synch authorities in Aura (each step for a set of authorities is labelled by the expected leader)</i>	30
4.1	<i>Example of blockchain network with six nodes which maintains the ledger. There are four miners of which two possible leaders. There are also two clients which submit transactions (bold arrows). Note that a node of the network can be a miner and a client simultaneously.</i>	34
4.2	<i>Transactions are Submitted by clients to the network. Then, once these transactions are collected in blocks, they change the state with the relative block. Each block waits for consensus in a Wait state or can be Refused. When the miners reach consensus a block will be Committed.</i>	35
B.1	<i>Fork scenarios that violate agreement and total order.</i>	51

# Bibliography

- [1] Joseph J. LaViola Jr. Andrew Miller. Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin. Tech report, University of Central Florida, April 2014.
- [2] Stefano De Angelis, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain. January 2017.
- [3] L. Aniello, R. Baldoni, E. Gaetani, F. Lombardi, A. Margheri, and V. Sassone. A prototype evaluation of a tamper-resistant high performance blockchain-based transaction log for a distributed database. In *EDCC*. IEEE, 2017.
- [4] Aura. <https://github.com/paritytech/parity/wiki/Aura>.
- [5] BFT-SMaRT. <https://github.com/bft-smart/library/>.
- [6] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121. IEEE, 2015.
- [7] E. Brewer. Cap twelve years later: How the “rules” have changed. *Computer*, 45(2):23–29, 2012.
- [8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook’s distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.
- [9] Vitalik Buterin. A proof of stake design philosophy.  
<https://medium.com/@VitalikButerin/a-proof-of-stake-design-philosophy-506585978d51>.  
Medium Blog.
- [10] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer, 2011.

- [11] Christian Cachin, Angelo De Caro, Konstantinos Christidis, and Jason Yellick. Architecture of the hyperledger blockchain fabric. 2016.
- [12] Christian Cachin and Marko Vukolić. Blockchains consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.
- [13] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
- [14] Clique. <https://github.com/ethereum/EIPs/issues/225>.
- [15] Nxt community. Nxt whitepaper. 2014.
- [16] R3 Corda. <https://github.com/corda/corda/>.
- [17] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158 – 179, 1995.
- [18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.
- [19] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. Blockbench: A framework for analyzing private blockchains. In *SIGMOD*, pages 1085–1100. ACM, 2017.
- [20] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
- [21] Go Ethereum. <https://geth.ethereum.org>.
- [22] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.
- [23] Hyperledger Fabric. <https://www.hyperledger.org/projects/fabric>.
- [24] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
- [25] Linux Foundation. Hyperledger fabric v1.0 architecture. <https://hyperledger-fabric.readthedocs.io/en/release/arch-deep-dive.html>.
- [26] Edoardo Gaetani, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. Blockchain-based database to ensure data integrity in cloud computing environments. In *ITA-SEC*, volume 1816. CEUR-WS.org, 2017.

- [27] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. *The Bitcoin Backbone Protocol: Analysis and Applications*, pages 281–310. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.
- [28] BitFury Group. Incentive mechanisms for securing the bitcoin blockchain. *White Paper*, 2015.
- [29] BitFury Group and Jeff Garzik. Public versus private blockchains part 1: Permissioned blockchains. *White Paper*, 2015.
- [30] BitFury Group and Jeff Garzik. Public versus private blockchains part 2: Permissionless blockchains. *White Paper*, 2015.
- [31] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed systems (2nd Ed.)*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., 1993.
- [32] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [33] Garrick Hileman and Michel Rauchs. 2017 global blockchain benchmarking study. 2017.
- [34] Apache Kafka. <http://kafka.apache.org/>.
- [35] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 2014.
- [36] Hyperledger Sawtooth Lake. <https://github.com/hyperledger/sawtooth-core/>.
- [37] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.
- [38] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
- [39] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- [40] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.
- [41] J. Mattila. The blockchain phenomenon. *Berkeley Roundtable of the International Economy*, 2016.
- [42] D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qijun. A review on consensus algorithm of blockchain.
- [43] Multichain. <https://www.multichain.com/>.
- [44] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. Available at <https://bitcoin.org/bitcoin.pdf>.



- [45] Proof of Authority.  
<https://github.com/paritytech/parity/wiki/Proof-of-Authority-Chains>.
- [46] Ethereum Proof of Stake. <https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ>.
- [47] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.
- [48] Peercoin. <https://peercoin.net/>.
- [49] Serguei Popov. A probabilistic analysis of the nxt forging algorithm. *Ledger*, 1:69–83, 2016.
- [50] L. S. Sankar, M. Sindhu, and M. Sethumadhavan. Survey of consensus protocols on blockchain applications. In *ICACCS*, pages 1–5. IEEE, 2017.
- [51] Francesco Paolo Schiavo, Vladimiro Sassone, Luca Nicoletti, and Andrea Margheri. FaaS: Federation-as-a-Service, 2016. Technical Report. Available at <https://arxiv.org/abs/1612.03937>.
- [52] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
- [53] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, pages 507–527, 2015.
- [54] Tim Swanson. Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. *Report, available online, Apr, 2015*.
- [55] Parity Technologies. <https://www.parity.io>.
- [56] L. Tseng. Recent results on fault-tolerant consensus in message-passing networks. In *Int. Colloquium on Structural Information and Communication Complexity*, pages 92–108. Springer, 2016.
- [57] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.
- [58] Marko Vukolic. Eventually returning to strong consistency. *IEEE Data Eng. Bull.*, 39:39–44, 2016.
- [59] Ethereum Whitepaper. <https://github.com/ethereum/wiki/wiki/White-Paper>.
- [60] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.