TÉCNICO LISBOA

# TBO: Total Byzantine Order

Scalable epidemic probabilistic total order resilient to Byzantine faults

**Nuno André Estêvão Anselmo**

Thesis to obtain the Master of Science Degree in

**Information Systems and Computer Engineering**

Supervisors: Prof. João Pedro Faria Mendonça Barreto
Prof. Miguel Ângelo Marques de Matos

## Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. João Pedro Faria Mendonça Barreto
Member of the Committee: Prof. Vinicius Vielmo Cogo

**January 2021**

# Acknowledgments

I would like to thank my supervisors, Prof. João Barreto and Prof. Miguel Matos, for the constant support and the in-depth discussions that allowed the work shown in this document to reach the level of depth it did.

I would also like to thank every Professor who helped shape my academic journey, and my University for providing me the opportunity to embark on said journey in the first place.

To all my colleagues, but especially to those who I spent countless hours working on projects with: André, Pedro, Liliana, Nelson, and many others, thank you. Without great colleagues such as yourselves this would have been an impossible achievement.

To the communities that initially fostered my love for computer science, thank you. Without your influence, I have no idea where would I have wound up. The hundreds or thousands of hours spent writing code and researching for a niche community helped form the foundation for everything I learned.

To my family and friends who encouraged and supported me throughout my education, and pushed me to continue studying, thank you. It was your support that kept me motivated and allowed me to now complete this stage of my studies. Thank you mom, dad, uncle, aunt, and brothers for pushing me forward. Thank you Lúcia for motivating me and keeping me going whenever I felt like giving up.

To everyone who helped me on this journey, thank you.

# Abstract

As blockchain technologies have taken the stage, they are being introduced into a multitude of industries in mission-critical applications. This surge in interest has led to the development of Hyperledger Fabric, an extensible blockchain system for distributed applications, which features modular consensus protocols and components. While consensus components have been developed for Fabric, none of the existing components is able to scale with the number of nodes participating in the consensus protocol. This document analyzes existing work in the area of blockchain technologies and consensus mechanisms, along with where they fall short, and proposes a new algorithm, TBO. This algorithm can be used as the consensus mechanism for Hyperledger Fabric, with the goal of being able to scale in both the number of transactions and the number of nodes by leveraging weaker correctness guarantees.

# Keywords

Distributed Total Order; Distributed Consensus; Blockchain; Byzantine Fault Tolerance; Hyperledger Fabric.

# Resumo

À medida que as tecnologias blockchain ganham prominência, têm estado a ser introduzidas numa multitude de indústrias e aplicações críticas. Este aumento no seu interesse levou ao desenvolvimento de Hyperledger Fabric, um sistema de blockchain extensível para aplicações distribuídas. Apesar do desenvolvimento de vários componentes de ordenação para o Fabric, nenhum destes componentes existentes tem a capacidade de escalar com o número de nós que participem no protocolo de consenso. Este documento analiza trabalho existente na área de tecnologias de blockchain e mecanismos de consenso, e também as suas limitações, e propõe um novo algoritmo, TBO. Este algoritmo pode ser utilizado como mecanismo de consenso para o Hyperledger Fabric, com o objectivo de conseguir escalar tanto em número de transacções como em número de participantes através da utilização de garantias de correcção mais fracas.

# Palavras Chave

Ordem Total Distribuída; Consenso Distribuído; Blockchain; Tolerância a Falhas Bizantinas; Hyperledger Fabric.

# Contents

# List of Figures

# List of Tables



# List of Algorithms

# Acronyms

| | |
|---|---|
| **BFT** | Byzantine Fault Tolerant |
| **PoW** | Proof-of-Work |
| **PoS** | Proof-of-Stake |
| **DPoS** | Delegated Proof-of-Stake |
| **PoSpace** | Proof-of-Space |
| **PoC** | Proof-of-Capacity |

# 1

# Introduction

## Contents

With the advent of cryptocurrencies, blockchain technologies have taken the stage and are being introduced into a multitude of industries, replacing trust-based systems with others that operate on a trustless model [15]. Unfortunately, due to the high-latency consensus algorithms used by most blockchains, their throughput is often low. For these technologies to replace existing large-scale trust-based systems (such as those found in the financial industry) with their trustless counterparts, scalability needs to be ensured without compromising the network's throughput, as it would otherwise lead to a degradation of service quality.

This is not an easy balance to achieve, as it is the very nature of these systems that leads them to exhibit these characteristics and, in particular, it is their nature that constrains their scalability, be it in transactions or in the number of nodes. To be able to fully grasp the problem, the defining characteristics of these technologies, as well as their limitations, need to be understood.

Permissionless blockchains, due to their permissionless nature, need to be able to operate under the assumption that attackers can generate identities at nearly no cost, and thus are able to subvert algorithms that rely on voting [16]. Solutions for scalable permissionless consensus therefore need to abstract themselves from participants' identities, relying instead on resources that cannot be forged such as a machine's resources or assets provided by the network.

The first widely adopted cryptocurrency, Bitcoin [1], solved the issue of scalable permissionless consensus through Proof-of-Work (PoW), whereby participants in the network reach consensus by participating in a lottery based on computational power, but it is not the only one. Alternative solutions to this consensus problem exist, such as Proof-of-Stake (PoS) [2, 10, 11, 17], Delegated Proof-of-Stake (DPoS) [8, 12, 18, 19], Proof-of-Space (PoSpace) [20], and Proof-of-Capacity (PoC) [21].

All these solutions have one aspect in common: since attackers can generate identities, the network needs to be secured through unforgeable resources.

With the introduction of permissioned blockchains, attackers become unable to generate additional identities as the identity associated with each peer has to be authorized before it is allowed to participate in the network.

This allows for consensus algorithms to once again rely on the identities of participants, leading to the removal of now-unnecessary expensive operations (in PoW/PoSpace/PoCapacity) or of complex systems and constraints (in PoS/DPoS). However, the scalability in number of nodes of permissioned blockchains remains limited by the consensus algorithm being used, which may need to be able to scale to at least hundreds or thousands of network participants for the replacement of existing systems to be possible.

Permissioned blockchains are of particular interest for systems where all participants can be easily identified, or for whom an identity is emitted by a few trusted entities. A prime example of such a system would be an European Union (EU) permissioned blockchain, where all citizens have an identity associ-

ated with their nation's identity card, emitted by said nation's government, in the form of an asymmetric key pair.

The motivation for the scale requirement immediately comes from the number of citizens that could be actively participating in the network, having over 500 million citizens spread across the EU [22]. With each nation participating in the cooperative project, it is not untenable that nations would want to run their own nodes to participate in the consensus algorithm. However, state-of-the-art consensus algorithms in permissioned blockchains [9, 23] are unable to scale in the number of nodes. This poor scalability in the number of nodes is a direct limitation on the degree to which the system can be decentralized, even if the number of users could scale.

The motivation for requiring Byzantine fault tolerance comes from the lack of trust in users, and the partial trust between the nations that would operate the system. Trust between nodes (or nations) would only be required when it comes to the emission of identities, but a system to do so is already in place today. Such a network would also be inherently more resistant to attacks as a single node (or even an entire nation) being compromised would not lead to a catastrophic failure scenario.

Another possible deployment scenario would be the financial industry, with a permissioned blockchain governing the exchanges, transfers, and operations between financial institutions [24]. In this scenario, the requirement for scalability in the number of nodes becomes even more prominent, as institutions would seek to maintain independence and prominence in this new system, by requiring that they be able to take part in the consensus protocol. With thousands of financial institutions, even if only major financial entities wanted to do so, the number of nodes would exceed the capabilities of current permissioned blockchain systems.

In this alternative deployment scenario, the requirement for Byzantine fault tolerance is immediate: when enormous sums of money are on the line, it would only be a matter of time until one of the many participants in the system went rogue. In addition to that, if the goal were to replace the current financial systems with those that operate on a trustless basis, then reintroducing trust by not tolerating Byzantine faults would defeat the purpose of their implementation.

To aid in the collaborative development of an open source enterprise grade distributed ledger framework, the Linux Foundation created the Hyperledger umbrella project [25]. The implementation discussed throughout this work was initially designed to take place on the Hyperledger Fabric project [14], a permissioned blockchain system and one of the many projects under the Hyperledger umbrella.

A notable characteristic of Hyperledger Fabric is its disassociation between components and low level of coupling between them, leading to increased modularity. This allows for components to be replaced at will, depending on the requirements and context of the deployment, enabling for different guarantees to be provided without requiring a complete overhaul of the entire architecture. This modularity is not limited to only a few modules, and even the ordering service can be replaced.

This ordering service is the component within Hyperledger Fabric responsible for the ordering of transactions, by atomically broadcasting them, ensuring that the order of transactions is consistent across nodes. In addition to that, it is also responsible for batching those transactions into blocks, and chaining them together, to form the blockchain.

While Hyperledger Fabric is capable of scaling in terms of transactions, and also in terms of network participants, current implementations of ordering services that operate with the presence of Byzantine failures do not scale in the number of ordering nodes [23]. This need for a more scalable yet Byzantine Fault Tolerant (BFT) ordering component was the main motivation behind the development of TBO, a total order algorithm that seeks to scale in number of nodes, while not compromising the scalability in the number of transactions, to be able to power deployment scenarios such as the one provided above. Initially tailored only to Hyperledger, it is usable in any permissioned network.

The main contribution of this thesis is the development of this new BFT total order algorithm usable on Hyperledger Fabric, TBO, based on EpTO [4], an epidemic total-order algorithm. TBO should enable scalability to a potentially arbitrary number of ordering nodes, by leveraging weaker correctness guarantees that do not compromise the integrity of the solution in realistic scenarios.

The main challenge in this development was striking a balance between the scalability in regards to the number of transactions, and in regards to the number of nodes, as would be required in the context of an EU blockchain, or a financial blockchain, as previously described.

## 1.1 Organization of the Document

This thesis is is organized as follows: Chapter 2 discusses related work and the state of the art, Chapter 3 describes the algorithm being presented, Chapter 4 will go over how the algorithm was evaluated, and Chapter 5 will be for the concluding remarks.

# 2

# Related Work

## Contents

In this chapter we will discuss existing work in the area of blockchains and also of Byzantine Fault Tolerant (BFT) total order and consensus algorithms. We will begin by discussing blockchains, first in regards to their architecture (permissioned and permissionless) and then in regards to their functionality (cryptocurrencies and smart-contract platforms). Following that, we will discuss Hyperledger Fabric and its architecture, followed by existing relevant BFT algorithms for Total Order, including those in use by certain blockchains. Last but not least, we will address the strengths and shortcomings of some of the work that was done in order to better emphasize the gap that will be filled by the proposed work.

## 2.1 Blockchains

A blockchain is, as the name suggests, a chain of blocks, with each block containing application-specific data:

- In the case of cryptocurrencies such as Bitcoin [1], the information being persisted is usually a transaction ledger, containing the transactions that have been made between participants on the network;
- In the case of smart contract platforms such as Ethereum [26, 27], the information being persisted is generally also a transaction ledger, with transactions that may be coupled with data representing function calls or other operations to be executed.

The key principle in any blockchain is not each individual block, but rather how these blocks are linked: by having future blocks containing references to previous blocks, usually by including the hash of the previous block within the new block as in Figure 2.1, a tamper-proof blockchain is created. A blockchain built in this way is tamper-proof as the hash of block $n$ becomes part of block $n+1$, and modifying block $n$ will therefore require modifying block $n+1$ to change this hash, which would in turn require modifying block $n+2$. Therefore, to modify block $n$, it is necessary to modify every block appended after it.

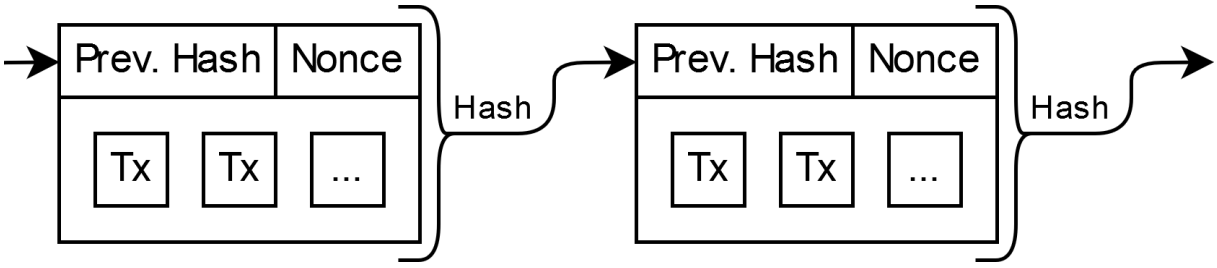



**Figure 2.1:** Example of a simple blockchain, with blocks containing only transactions, a nonce, and a hash of the previous block.

This allows for a blockchain to guarantee that blocks, once old enough, are virtually immutable. This assumption can be used to secure information on a blockchain, the most common being finan-

cial transactions, by using the blockchain as the foundation for a distributed ledger: network participants can continuously generate new transactions that get included into blocks, with the guarantee that once a transaction has been included in a block, and that block reaches a certain age (or depth in the blockchain), it is essentially irreversible.

It is necessary to allow for a certain age or depth due to the possibility of forks[1], which can be briefly described as valid chains that may or may not be the longest. When reading from the blockchain, network participants will read from the longest valid chain, as that will be the one that is most up-to-date. Should there be two forks of equal length, and one of them appends a block before the other (becoming the longest), then every peer will suddenly switch over to the longest chain. Since different forks may contain different transactions, it is entirely possible that the peers who switched had read a transaction in their previous fork that is not present in the longest fork, making it, in essence, a reverted transaction.

### 2.1.1 Permissionless Blockchains

An important distinction between permissionless blockchains and some of the other BFT distributed systems, is the lack of an entity that emits or verifies identities: each node can create its own identity, and identities do not need to be approved by any network entity.

This leads to the possibility of a Sybil attack [16], where a malicious attacker running a potentially infinite number of virtual network nodes participates in the consensus protocol as if they were independent, in an attempt to subvert the network or to tamper with the blockchain. To mitigate this problem, consensus on the distributed ledger needs to be reached based on some resource that cannot be generated at will by network participants, as otherwise these attackers can arbitrarily modify the ledger through their superior numbers.

To solve this problem, Bitcoin [1] implemented a Proof-of-Work (PoW) system based on Hashcash [28] formally called Nakamoto Consensus, which requires blocks to contain some proof that computational power was spent on generating that block. This is done by requiring the block to contain a nonce, and that the hash of the block begins with a certain number of zero bits, before it can be appended. Since hashes are irreversible by nature, determining a nonce that will cause the hash of the block to start with a number of zero bits is a computationally expensive operation, one that requires continuously generating hashes of blocks with different nonces, until a valid hash (and its corresponding nonce) is found. This nonce can then be used as a proof-of-work: for the nonce to have been discovered, computational resources have to have been spent, so any correct nonce is proof of computational work having been done to find it.

By requiring nodes to spend computational power generating these proofs, a computational cost is introduced into the procedure of generating new blocks. Since to modify a block on the network, one

[1] The notions of soft and hard forks will be disregarded, as the focus will be solely on forks that occur during normal operation.

needs to redo all the blocks that come after it, this means that the computational power that was spent generating those blocks in the first place will need to be spent once again, as the hashes of the previous blocks within them will change to point to the new hashes of the previous blocks.

If more computational power is controlled by honest nodes than by malicious nodes that could be cooperating in an attack, then the original chain will grow faster than the modified chain, as more computational power will be available to generate blocks. This means that the chain is secure against tampering as long as a majority of computational power is controlled by honest nodes that continue to append blocks to the blockchain. In other words, the chain is secure if less than 50%[2] of the network's computational power is compromised or malicious.

Through a PoW-based algorithm, Bitcoin can achieve consensus on the state of the distributed ledger, even when network nodes generate an arbitrary number of identities. The main problem with this approach, however, is that computational resources have been spent on nothing more than securing the network, and can be seen as wasted. The amount of energy consumed by this process is not to be scoffed at either: energy consumption for Bitcoin alone peaked at 73 TWh/year [29] for most of 2018. When compared to the worldwide power consumption rankings, this puts Bitcoin between Venezuela (38th) and Austria (39th) [30].

To reduce the dependency on computational resources to secure the network, other algorithms have been developed or are under development. While less common, these algorithms seek to reduce or entirely remove the computational power required to secure the network, which will in turn lead to a significantly lower power consumption.

The most prominent alternative is Proof-of-Stake (PoS) [2, 10, 11, 17], which replaces the role of computational resources by that of a certain amount of cryptocurrency, called a stake. Rather than proving that they have computational resources like miners do, stakers instead prove that they own a stake in the form of currency in the protocol, and one of these stakers will be able to create a block. Multiple algorithms exist to select which staker will do so, but the simplest method is a weighted random selection, where the stake owned by a staker, divided by the total stakes, will yield the probability of said staker being chosen to generate the next block.

PoS faces some hurdles however, specifically it faces the *nothing-at-stake* problem where a node has nothing to lose by participating in the consensus process of multiple forks simultaneously, which in the worst case scenario can stop consensus from ever being reached[3]. This problem exists since, unlike in PoW, there are no significant computational resources being used when participating in the consensus process, and it is therefore most profitable to participate in all forks to ensure block rewards might be received regardless of what the longest chain is. In PoW, doing so would require splitting computational

[2] Depending on the blockchain being discussed, certain attacks have been developed that lower this threshold by taking advantage of block propagation delays.

[3] It does not necessarily stop blocks from being generated, but since multiple competing chains all keep producing blocks, there is no agreement, and therefore no consensus.

resources between both forks, which would not maximize profit.

The nothing-at-stake problem has multiple proposed solutions, some of which are currently being used by some permissionless blockchains, but the majority of solutions can be placed in one of two groups:

- Solutions that apply some sort of checkpoint mechanism [17], which stops blocks from being replaced by a different fork once they reach a certain depth even if a longer valid chain exists. Note however that this does not stop competing chains from existing for longer than said number of blocks, as it only stops a client (or a node) from switching forks once a checkpoint has been crossed.

- Solutions that rely on punishing [2, 10, 11], where stakers can be punished for participating in multiple forks simultaneously, often through the partial or total destruction of their stake, or temporary suspension of their ability to stake if an attack is attempted.

A variation of PoS, called Delegated Proof-of-Stake (DPoS) [8, 12, 18, 19], operates by having stakers not directly generating blocks, but instead voting on which nodes can generate blocks. This establishes a method through which these nodes can be removed, should they misbehave, while ensuring they are semi-trusted as they have an incentive to behave correctly.

Both of these PoS versions suffer from a problem wherein the staking nodes, in the cases of cryptocurrencies, are collecting additional currency from the network as a reward for staking, in turn growing their own stake. These staking processes will, over time, control a larger and larger portion of the circulating currency supply. Setting a low barrier of entry to staking generally makes this problem less prominent, as the rewards will be distributed over a larger portion of the network participants, not leading to network centralization.

Other alternatives, such as Proof-of-Space (PoSpace) [20] and Proof-of-Capacity (PoC) [21], rather than relying on proving computational power, rely on proving the ownership of storage (be it memory or storage further down the memory hierarchy). This neither entirely eliminates energy consumption as it replaces computational operations with memory or I/O operations, nor does it reduce it to the degree that PoS does. However, the energy required to keep said storage resources running is comparably lower than the energy requirements to constantly compute hashes as in PoW, making it more energy-efficient.

It becomes however clear that there are no simple solutions to implement consensus in a permissionless blockchain, due to the lack of identities. PoW-based algorithms do not depend on identities in any way, but lead to computational resources being wasted solely on securing the network. Those based on PoS do not require nearly as much computational power, but introduce a dependency on a cryptocurrency, and increase network complexity. Using DPoS has less complexity than PoS, but leads to slightly increased centralization, with the same drawback of requiring a cryptocurrency. The ones

that use PoSpace and PoC, similarly to PoW, lead to storage/memory resources being wasted solely on securing the network.

### 2.1.2 Permissioned Blockchains

In contrast with permissionless blockchains, permissioned blockchains give each node and client[4] an identity that can be verified, and that cannot be generated without going through some procedure that requires agreement from the network, or from some trusted entity.

One of the advantages that comes from the presence of these identities is the ability to setup roles. For example, a permissioned blockchain can have 3 segregated roles:

- Reader - The reader can only read the blockchain state;
- Miner - The miner can only mine blocks and read the information required to do so from the blockchain;
- Client - The client can only create transactions, which miners can then include in blocks;

Although some segregation of permissions can be done in semi-permissioned blockchains, e.g. in DPoS with miners being an elected subset of all nodes, fully-permissionless blockchains are unable to setup any kind of role system.

Another advantage that comes from the presence of these identities is that, as every client and node is securely identifiable, it becomes possible to use standard BFT algorithms to achieve consensus[5]. This allows for consensus to be reached without relying on any of the previously mentioned algorithms that are independent from network identities, out of necessity.

This does not allow for Sybil attacks [16] as identities need to be authorized in some way before they can be used, and cannot be generated at will by attackers.

One of the main advantages of using BFT algorithms instead of any of the aforementioned ones is that they require neither significant resources, be it computational or storage/memory, nor an underlying cryptocurrency, reducing the cost for participating in the network. In addition to that, BFT algorithms often allow for a much higher throughput to be reached, improving the scalability of the blockchain in the number of transactions (or operations) per second.

The use of these algorithms does come with a disadvantage: while scalability in the number of transactions is generally better, the scalability in the number of nodes is generally worse [6–8, 13]. This poor scalability in number of nodes is, first and foremost, due to an increased number of messages

[4] Clients are the network participants that emit and read transactions, while nodes participate in the protocol by propagating and/or validating transactions and/or creating blocks with them

[5] It is also possible, if the environment is appropriate and some degree of trust exists, to ditch BFT entirely and use a fail-stop failure model, but this use-case will not be discussed in this work.

required to reach consensus. In a great number of BFT algorithms, such as the ones outlined above, the number of messages grows superlinearly with the number of nodes, limiting the maximum number of nodes, at the very least due to limitations in bandwidth. In addition to this, unlike the consensus algorithms used in permissionless blockchains that only require 51%[6] of nodes to be correct, most other BFT algorithms require 2/3 of the nodes to be correct.

This poor scalability in the number of nodes is a direct limitation on the degree to which the system can be decentralized, even if the number of clients could scale. In an ecosystem where dozens of organizations may be participating it is not absurd to expect each organization to want to run its own node, even if only to not be dependent on its competitors. If hundreds of organizations participate, as in the case of a financial network for banks, then the lack of scalability in the number of nodes quickly becomes a limiting factor, and will negatively impact the performance or even the functionality of the entire system.

Permissioned blockchains, by nature, require an underlying membership or identity management system to be present, which can be either managed separately from the blockchain or integrated with it, by embedding memberships and permissions in the blockchain. This membership system serves to manage the identities of network participants, and the permissions associated with said identities, in other to restrict the actions that can be performed.

Generally speaking, the only requirement that needs to be met by this membership system is that participants need to be able to authenticate themselves when performing certain actions, and that participants can authenticate and authorize one another (if given permission to do so). An example of these requirements playing out would be a miner having to be authenticated when submitting a block to other nodes, which could accept or reject said block depending on whether the miner was authorized to mine.

### 2.1.3 Blockchains as Cryptocurrencies

In contrast to the two previous points, the focus will now be not on the permission model but rather on the purpose and functionality of the blockchain itself. The first and simplest model is that of a cryptocurrency: blockchains being used to record transactions that use a virtual currency.

The first example of such a system was Bitcoin [1], with the virtual currency being called a bitcoin, and with all transactions representing a transfer of some amount of bitcoin. With each and every transaction being recorded on the blockchain, the balance of each address (account, in other terms) can be determined by replaying the transactions involving it.

Since each address has a key pair associated with it, the public key can be used to validate all transactions that are supposedly being made by said address, while the private key can be used to create those transactions. This ensures that no user can touch another user's funds without having

[6] Depending on the blockchain this percentage may be higher, due to attacks taking advantage of propagation delays.

control over their private key, which stops any potentially malicious actor from exerting any control over other users' addresses.

This architecture does not require keeping a state, as transactions can be replayed to obtain the current state of the network, but one is generally kept for performance reasons. This state only needs to contain the unspent transaction outputs (sometimes referred to as UTXOs), enabling new transactions to be quickly verified by ensuring that they may only use a given output once.

Bitcoin is the most popular cryptocurrency as of writing, but it is by no means alone: at the time of writing, one of the most cryptocurrency tracking websites is tracking 2425 cryptocurrencies across several exchanges, while another is tracking 2073, of which 750 have over 1 million dollars in market cap.

### 2.1.4 Blockchains as Smart Contract Platforms

Using blockchains to create cryptocurrencies is the most common model, but it is possible to generalize it further, allowing for execution of arbitrary code in a distributed manner. This model is based on smart contracts: blockchains are used to record operations on programs, referred to as smart contracts, which execute on a virtual environment.

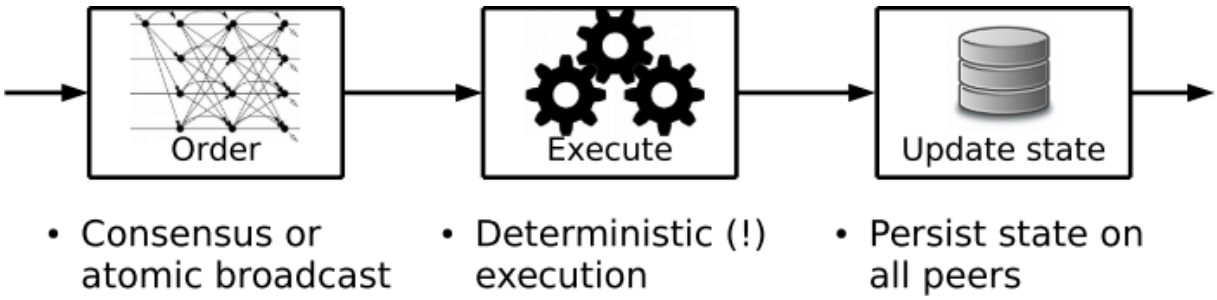



**Figure 2.2:** Order-execute architecture, as taken from [14].

An important aspect of most blockchains used for smart contracts is that they follow an *order-execute* architecture such as the one in Figure 2.2; transactions (which represent operations) are submitted and are ordered by being included in blocks prior to being executed, which reduces performance as every operation needs to be executed on every node. But another limitation is also introduced: since they are ordered before being executed, it is necessary that, from the same ordering, the output be the same, otherwise the state would naturally diverge. In essence, all operations under an order-execute architecture need to be deterministic.

While at first this may not seem like a significant limitation, most general-purpose programming languages provide non-deterministic operations as part of their core functionality, or are implemented in a non-deterministic manner, limiting which programming languages can be used. To ensure determinism,

these platforms often use a specialized deterministic instruction set, and use domain-specific programming languages that can be compiled to this specific instruction set.

By having every smart contract converted into deterministic operations, contracts can be modelled as state machines. Since they are state machines, it also means that state (other than the initial one) does not need to be kept: to reach the final state, all that needs to be done is to execute all deterministic operations in the correct sequence. The current state is usually kept for performance reasons, just like in conventional cryptocurrencies, but it is not strictly necessary.

The most popular smart contract platform is Ethereum [26,27], which executes code on the Ethereum Virtual Machine (EVM), and that is capable of reaching 26 transactions per second in its current state. The EVM follows a stack-based architecture, which executes domain-specific bytecode, which in turn performs operations that modify one of three data structures:

- Stack - A last-in-first-out (LIFO) container that is non-persistent;
- Memory - An infinitely expandable byte array that is non-persistent;
- Storage - Long term key-value store that persists even after the computation ends, and that can be used in future computations;

In addition to modifying data, these instructions may also change the execution flow or call other functions or even functions in other smart contracts, allowing for smart contracts to be interconnected.

In order to assist the development of smart contracts, two major programming languages exist that compile directly to EVM bytecode: Solidity [31] and Vyper [32]. Solidity is as of writing the most popular language for programming smart contracts for Ethereum, with strong object-oriented primitives, with Vyper being a security-focused pythonic programming language that is still in beta.

An important aspect of execution in the EVM is that of gas: every instruction has an associated cost in gas, and the contract will only keep executing while sufficient gas is present. Should the execution run out of gas, the execution is aborted and its effects rolled back. Gas exists for the same primary purpose as Bitcoin's transaction fees: to give miners an incentive to include a given transaction in the block. It also serves to introduce a scalable cost so that more expensive operations, and more expensive contract executions, have a higher cost. The final purpose is that of a halting condition: since it is impossible to know whether execution will eventually halt [33], gas ensures that it eventually will, since it will eventually run out, mitigating attacks which would execute an infinite loop.

Ethereum is the most popular smart contract platform at the moment, with one of the most popular cryptocurrency tracking websites tracking 1199 cryptocurrencies developed as smart contracts on top of it. Out of the 2073 cryptocurrencies which this website tracks, 1199 exist as smart contracts on Ethereum.

## 2.2 Hyperledger Fabric

Hyperledger Fabric [14] is a permissioned blockchain system, and one of the many projects under the Hyperledger umbrella. Unlike the majority of blockchains that follow an order-execute architecture where transactions are first ordered and then executed, Hyperledger Fabric follows an *execute-order-validate* architecture where transactions are first executed, then they are ordered, and then they are validated, as illustrated in Figure 2.3.

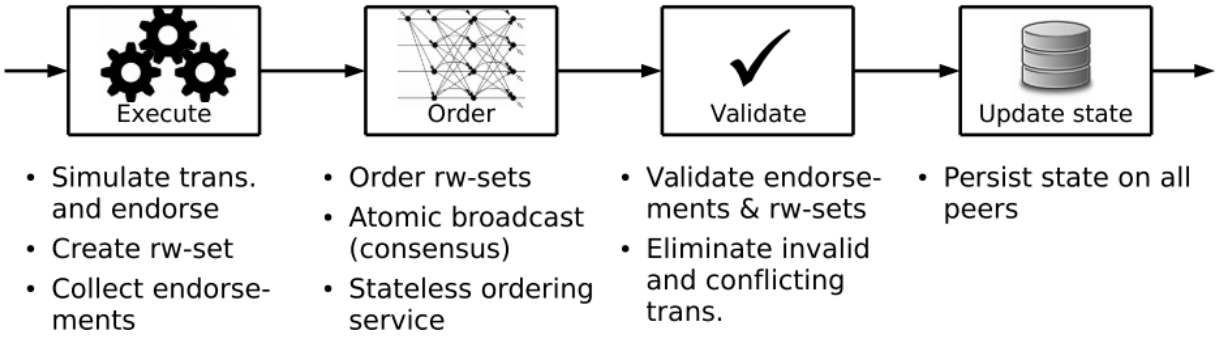



**Figure 2.3:** Execute-order-validate architecture of Hyperledger Fabric, as taken from [14].

This altered architecture comes with significant advantages compared to the traditional order-execute architecture:

- The execution of transactions does not need to be performed by every node on the network, as a number of nodes will execute the transactions and calculate the write-sets (the changes to the state). Once they are included in the blocks they only need to be validated, with their write-sets then being applied, rather than the transaction executed.
- Since transactions will be executed before being included in the blockchain, they do not need to be deterministic as their non-deterministic behavior will be replicated throughout all nodes. The result of the non-deterministic operation will be stored on the blockchain, in the form of read-sets (used for validation) and write-sets (used for applying the results), instead of only storing the operation itself.
- As the requirement for determinism is lifted, it becomes possible to use general-purpose programming languages, completely eliminating the requirement for a domain-specific programming language.

To better explain how this architecture works, a closer inspection of the individual phases of the transaction flow is required.

First, a transaction proposal containing the identity of the client, the smart contract (referred to as chaincode in Hyperledger Fabric terminology) identifier, the operation to be performed and its param-

eters, as well as a random nonce and a transaction identifier, is created and signed by a client, and is sent to the set of nodes responsible for the execution of that specific chaincode, called the endorsers.

The endorsers, after simulating the execution of the operation, produce a read-set and a write-set, which correspond to the keys read (and their versions) and the keys that were written to (and the values being written). This read-set and write-set are then included into an endorsement, along with the transaction ID, the endorser's identity, and a signature from the endorser. This endorsement is sent to the client which submitted the transaction proposal.

Once a given number of endorsements is reached (as specified by properties related to the chaincode), the client can assemble a transaction containing the information found in the transaction proposal, as well as the endorsements that have been obtained. This transaction is then submitted to the ordering service, to be included in a block.

The nodes that make up the ordering service will, after the conditions set by the ordering service (which may be a BFT algorithm) are met, output a block that then needs to be delivered to every node. Since the number of nodes participating in the ordering service is usually much lower than the number of total nodes in the system, a gossip service is often used to help disseminate these blocks.

Once a node receives a block, the final stage of the transaction flow is reached: validation. Despite having been included in blocks, transactions are not necessarily valid, and require additional validation by every node receiving them. For example, it is entirely possible for two conflicting transactions to have been included in the same block. To perform the validation, nodes first validate the endorsements, including whether the required endorsements are present. Afterwards, the read-set of each transaction is checked and, if the version for any of the keys is not a match for the version found in the peer's local state, the transaction is marked as invalid. Lastly, the block is stored locally and the write-sets are applied to the local state, modifying the stored keys as necessary. The block is now considered processed.

With the general flow of transaction processing described, we will now focus on studying the different components of Hyperledger Fabric, with a particular emphasis on the components we will extend or improve (the ordering and possibly dissemination components), and the components we will use (the membership and dissemination components). The architecture of a sample Hyperledger Fabric network is present in Figure 2.4, with the different components, along with multiple chaincodes, outlined. It is important to note that, in Hyperledger Fabric, the architecture is highly modular and components can be swapped, as long as the specified interfaces are still present. All components that will now be discussed can be altered without requiring any modification to other components, as long as their defined interfaces are correctly implemented.

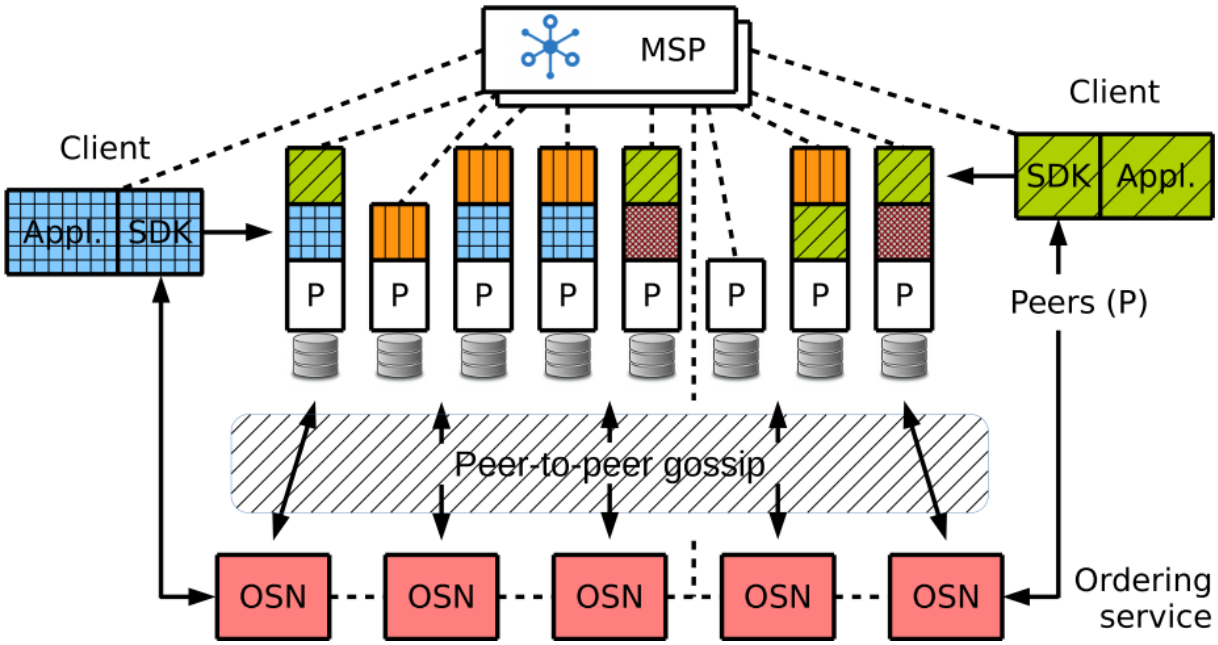



**Figure 2.4:** The architecture of a Hyperledger Fabric network, as taken from [14]. The differently shaded and colored squares reflect different chaincodes.

### 2.2.1 The membership service

The membership service is responsible for managing the identities and permissions of every node and participant in the system, and for verifying said identities. Communication in the Hyperledger Fabric network requires authentication at every step, and it is the membership service that handles this authentication. Each client and node communicates with the membership service through a local membership component, that serves only as an interface. This local membership component performs all operations related to authenticating and verifying the integrity of transactions, signing and authenticating endorsements, and also authenticating other blockchain operations.

By default, the underlying system is a public key infrastructure. It is possible to use either a commercial certification authority as part of this infrastructure, or a stand-alone certification authority included with Hyperledger Fabric, called Fabric-CA, depending on what may be required.

In addition to this, it is also possible to perform identity federation, with the authentication for members of an organization being deferred to a specific but independent membership service. This would typically be required when dealing with a larger ecosystem with multiple organizations, where keeping a single centralized membership service would be unfeasible. Federation would enable all authentication and authorization to be done through a central membership system that would then access each organization's private one. By doing this, a common interface for all membership operation is established, while allowing each organization to run its own membership service.

### 2.2.2 The dissemination component

The official Hyperledger Fabric documentation does not delve into much detail regarding the dissemination component. This dissemination component is responsible for efficiently broadcasting blocks that have been output by the ordering service, so that nodes can then proceed to validate the transactions within them.

This dissemination component uses an underlying communication layer based on gRPC [34], which in itself means gRPC Remote Procedure Calls, with mutual authentication required prior to any exchange of information.

Each node in the network maintains an up-to-date membership view of online nodes, by independently managing a local view based on dissemination messages that are periodically sent. Failed nodes, following recovery, are also able to reconnect to a membership view. Two phases are used for communication: push and pull. During push, a random set of active neighbors is chosen from the membership view and a given message is broadcast to them; during pull, missing messages are requested from another random set of active nodes.

In addition to this, the dissemination component is also responsible for handling the election of a leader, which exists solely to minimize bandwidth. This is done by, rather than having each node in the ordering service broadcast the output block to other non-ordering nodes in the network, delegating that task to the elected leader.

### 2.2.3 The ordering service

The ordering service is responsible, most importantly, for atomically broadcasting transactions, ensuring that they are seen in the same order by every node. In addition to that, it is also responsible for grouping transactions into batches, referred to as blocks, and chaining them together through hashes in an append-only blockchain.

Blocks are formed by grouping transactions received within a given period of time. This is done by starting a timer the moment the first transaction is received, and finalizing the block once either the timer has elapsed, the maximum size of the block is reached, or the maximum number of transactions in it is reached. Once the block is finalized, all transactions which have been included in the block so far will be included in it, but any transaction received afterwards may only be included in future blocks.

Since blocks need to be generated deterministically from the same set of transactions, as otherwise the state of the nodes in the ordering service would diverge, it is necessary that transactions be included in the exact same order on every node participating in the ordering service. To ensure this for transactions is simple, as it only requires that they are included in the same order as they were atomically broadcast. Since there is also a condition for finalizing blocks based on a timer, it is also necessary to

synchronize this timer: when the timer elapses on a given node, rather than immediately finalizing the block, it instead atomically broadcasts a special *time-to-cut* transaction, which signals to the ordering nodes that that the block should be finalized. This ensures that this transaction is received just like any other, and the order is therefore consistent.

To conclude, the ordering service is responsible for, when given a set of transactions, ordering them in a deterministic way and batching them into blocks of variable size. Multiple implementations for this ordering service already exist: Solo, that runs a single centralized node to handle these operations (unsuitable for production environments); a service based on Apache Kafka; and a proof-of-concept based on BFT-SMaRt [6, 23] that enables BFT, which will be discussed in greater detail later on.

## 2.3 Algorithms for total order and consensus

In this section, we compare existing algorithms for total order as well as for consensus, some of which are BFT and some of which are not.

### 2.3.1 Nakamoto Consensus [1]

This consensus algorithm is the one behind Bitcoin. It relies on PoW for its core process, which is the lottery behind its leader election. In essence, it runs a lottery through PoW to elect a leader, with the leader being granted the ability to generate a block.

### 2.3.2 Casper [2, 3]

The consensus algorithm behind the specification of Ethereum 2.0, which is now live in its first stage with a beacon chain deployment [35]. Currently, it exists in two different architectures: "Casper the Friendly Finality Gadget" (FFG) [2], which uses a hybrid PoW/PoS consensus; "Casper the Friendly GHOST[7]: Correct By Construction" (CBC) [3], which uses a purely PoS consensus mechanism. Both are in active development, and subject to changes, but FFG has already been deployed in the first stage of the Ethereum 2.0 rollout. In the future, FFG is planned to be replaced with CBC which is more robust and complete, along with the addition of new features such as sharding which would significantly increase network throughput. [35]

### 2.3.3 EpTO [4]

This algorithm is an epidemic total order algorithm designed for large-scale systems. As such, it provides scalability in both the number of processes and the number of transactions (events in EpTO terminology).

[7] Greediest Heaviest Observed Sub-Tree

However, unlike most other algorithms being discussed in this section, it does not operate under the presence of Byzantine faults, and instead operates only under a fail-stop failure model. In addition to that, it does not provide agreement, only probabilistic agreement: "If a correct process decides to `EpTO-Deliver` an event $e$, then with high probability all correct processes eventually `EpTO-Deliver` $e$".

The EpTO algorithm is composed of two separate but interconnected components: the dissemination component and the ordering component. The dissemination component is responsible for satisfying the probabilistic agreement property by disseminating events to every correct process, while the ordering component is responsible for ensuring total order is satisfied. Each process executes each component independently from other processes, communicating only through sets of events, referred to as balls.

The dissemination component, which is found in full in Algorithm 3.1, operates in a round-based fashion: every $\delta$ units of time a function, `execute_every_delta`, is executed. A ball, `next_ball`, contains the events that are to be relayed during the next round.

Upon receiving a ball, the events within it are iterated by the process receiving it: if the event has not yet been disseminated enough times, as indicated by its $ttl$ (time to live), it is added to the ball for the next round. If an identical event is already in `next_ball`, then the one with the highest $ttl$ is kept, to minimize excessive retransmissions. Once every event has been iterated, the internal clock is updated.

Once $\delta$ units of time have elapsed, the `execute_every_delta` function is called. It iterates over all events in `next_ball` and increases their $ttl$, before selecting a random subset of correct processes and relaying `next_ball` to those processes. Afterwards, the ordering component is called, and `next_ball` is reset.

The ordering component, keeps track of two sets, `received` and `delivered`, containing the received and delivered events respectively. To keep track of the last timestamp to be delivered, another variable, `last_delivered_ts`, is used.

The ordering component, presented in full in Algorithm 3.2 begins by incrementing the $ttl$ of every event in `received`. Then, it iterates over every event in `next_ball`: if the event has not been delivered, and neither has another event with a higher (more recent) timestamp, then it is added to the `received` set. If it is already present in the set, the event with the highest $ttl$ is kept.

The `received` set is then iterated over, and every event that no longer needs to be disseminated (as given by its $ttl$) are considered deliverable and added to a `deliverable` set. Afterwards, events in the `deliverable` set that have a timestamp higher (more recent) than the lowest (oldest) timestamp among undeliverable events, are removed.

Events that remain in the `deliverable` set are then ordered based on their timestamp, removed from the `received` set, added to the `delivered` set, and delivered to the application.

Through the above algorithm the probabilistic agreement property and the total order property are ensured. Combining these two properties, then with high probability every correct process eventually

delivers the same events, and does so in the same order.

### 2.3.4 PBFT [5]

This algorithm for total order operates in three phases: pre-prepare, prepare, and commit. When the primary replica receives a request from a client, it assigns it a sequence number and broadcasts a PRE-PREPARE message containing the message, its sequence number, and the view. After receiving a PRE-PREPARE message, nodes respond by multicasting a PREPARE message containing the sequence number, the view, and a digest of the message. Once $2f$ PREPARE messages have been received, nodes broadcast a COMMIT message, once again containing the sequence number, view, and digest of the message. Once $2f + 1$ COMMIT messages are received, replicas know that a quorum has been reached and the message can be decided.

### 2.3.5 BFT-SMaRt [6]

This total order algorithm is similar to PBFT and operates following a PROPOSE-WRITE-ACCEPT three-phase protocol. It begins with a node proposing a batch of requests to be decided, sending a PROPOSE message to every node. Every node receiving a PROPOSE message then sends a WRITE message containing the hash of the proposal, and every node receiving a sufficient number of WRITE messages (depending on the fault model in use) sends an ACCEPT message containing the proposal's hash to every node. Finally, every node that receives the sufficient number of ACCEPT messages can decide on the proposal.

### 2.3.6 RBFT [7]

Like the previous algorithm, RBFT is also similar to PBFT. The main difference is that it runs multiple instances of the protocol in parallel: a master protocol and multiple backup protocols, with the backup protocols having different primary nodes. This allows nodes to detect whether the performance of the current primary in the master protocol is sub-par, and perform an election to switch primary.

### 2.3.7 Solida [8]

This algorithm is based on PBFT as well, and modifies the process through which nodes join the subset that performs the consensus protocol, called the committee. This committee reaches consensus on batches of transactions (blocks) to be appended through a 6-phase PBFT-style protocol. Nodes are able to join the committee by submitting a valid PoW, and the node that has been in the committee the longest is automatically removed when a new node joins, ensuring the committee is dynamic. In addition to that,

and similarly to RBFT, the process through which the primary is selected is also modified, by having the primary be the latest node to have joined the committee.

### 2.3.8 DBFT [9]

This is the algorithm being used by the Red Belly Blockchain. Unlike the three previous traditional BFT algorithms, it does not require a correct primary (leader) for termination and progress to be guaranteed. Instead, it uses a weak coordinator to accelerate the consensus, but correctly operates even if the coordinator misbehaves.

### 2.3.9 Tendermint [10, 11]

Based on PoS, it requires that the nodes participating in the consensus algorithm, called validators, lock coins of the cryptocurrency in exchange for being able to become validators. The voting power of each validator is defined by the number of coins they have locked. These validators sign commit votes for blocks and, once a block has 2/3 of the votes, it is finalized. If a validator is found to be signing commit votes for two competing blocks, their locked coins are destroyed.

### 2.3.10 Algorand [12]

This algorithm is based on PoS, is used in the Algorand cryptocurrency, and operates with proposers and a committee. Both the proposers and the committee are randomly chosen but perform different steps. The proposers are responsible for generating a block, and the committee members are responsible for reaching consensus on which of the proposed blocks is to be appended to the blockchain. The random selection of the proposers and the committee members, both of which are a subset of the users, is weighed by the amount of the Algorand cryptocurrency they own, which is what ensures it resists Sybil attacks.

### 2.3.11 HoneyBadgerBFT [13]

The key component in this algorithm is the notion of an Asynchronous Common Subset (ACS) primitive, with each node being able to propose values, with the guarantee that the output will contain the values proposed by at least a subset of $N - 2f$ nodes. Through this ACS primitive, atomic broadcast, and thus total order, is achievable, by having the nodes propose transactions (values), and output the result of the ACS primitive.

## 2.4 Discussion of algorithms for total order and consensus

In this section we will summarize some properties of the algorithms discussed in Section 2.3, and discuss some of their properties.

The summary of these algorithms may be found in Table 2.1.

In terms of scalability, there are three groups of algorithms: those that scale only in the number of transactions, those that scale in only the number of nodes, and those that scale in both.

The algorithms that scale only in the number of transactions are mostly traditional BFT algorithms. These generally lack scalability in the number of nodes due to a superlinear increase in the number of messages required to reach consensus as the number of nodes is increased.

On the other hand, the mentioned algorithms that scale only in the number of nodes generally have the number of messages scale sublinearly as the number of nodes is increased. However, as they are based on PoW and/or PoS, they do not scale in the number of transactions.

Lastly, the algorithms that feature scalability in both the number of transactions and the number of nodes come with their own drawbacks. For DBFT, its scalability seems to hit a ceiling at only a few hundred nodes according to the limited tests provided. For Casper CBC, its scalability in the number of transactions is not yet determined, and it hasn't yet been implemented. For Solida, its scalability in the number of transactions is uncertain, and its scalability in the number of nodes in its committee is quadratic, although the number of total nodes can grow sublinearly. For EpTO, it lacks Byzantine fault tolerance.

**Table 2.1:** Comparison of the presented total order and consensus algorithms.

| Algorithm | Efficient[8] | Scales in TXs | Scales in Nodes | BFT |
|---|---|---|---|---|
| Nakamoto Consensus | No | No | Yes | Yes |
| Casper FFG | No | Uncertain | Yes | Yes |
| Casper CBC | Yes | Uncertain | Yes | Yes |
| EpTO | Yes | Yes | Yes | No |
| PBFT | Yes | Yes | No | Yes |
| BFT-SMaRt | Yes | Yes | No | Yes |
| RBFT | Yes | Yes | No | Yes |
| Solida | No | Uncertain | Yes[9] | Yes |
| DBFT | Yes | Yes | Yes/No[10] | Yes |
| Tendermint | Yes | No | Yes | Yes |
| Algorand | Yes | No | Yes | Yes |
| HoneyBadgerBFT | Yes | Yes | No[11] | Yes |

[8] Power-efficient, meaning no significant amount of computational power should be wasted, e.g. generating a proof-of-work
[9] Committee scales fine up to hundreds of nodes, but scales quadratically
[10] Scales up to hundreds of nodes
[11] Scales up to hundreds of nodes

# 3

# Algorithm

## Contents

In this chapter we will be going over the main contribution: a Byzantine Fault Tolerant (BFT) Total Order algorithm, based on an existing epidemic algorithm, EpTO [4], initially introduced in Section 2.3.3, which we will refer to as TBO. We will begin by looking at the properties that we seek to ensure, followed by EpTO and, by making modifications in order to mitigate certain attacks, we will arrive at TBO. Afterwards, we will look at how TBO might be attacked, and demonstrate that these attacks, in very strict conditions, are possible.

## 3.1 Properties

Before delving into the algorithm, we will how first look at the properties that we seek to ensure:

1. Integrity: For any event $e$, previously `TBO-Broadcast` by a correct process, every correct process will `TBO-Deliver` $e$ at most once.
2. Probabilistic Validity: If an event $e$ is `TBO-Broadcast` by a correct process $p$, then $p$ will eventually `TBO-Deliver` $e$ with high probability.
3. Total Order: If correct processes $p$ and $q$ both `TBO-Deliver` events $e$ and $e'$, then $p$ will `TBO-Deliver` $e$ before $e'$ if and only if $q$ will also `TBO-Deliver` $e$ before $e'$.
4. Probabilistic Agreement: If a correct process decides to `TBO-Deliver` an event $e$, then with high probability all correct processes eventually `TBO-Deliver` $e$.

It is important to stress the difference between more traditional Total Order algorithms in regards to the validity and agreement properties: the agreement and validity offered by TBO are probabilistic in nature. In terms of properties ensured, TBO is strictly weaker than EpTO, as EpTO seeks to ensure validity rather than probabilistic validity, but they both seek to ensure probabilistic agreement.

## 3.2 EpTO

EpTO, as presented here, is unchanged from the original [4], which was explained in some depth in Section 2.3.3.

As a brief summary, the algorithm is layered into two components, and their guarantees are roughly as follows:

1. Dissemination (Algorithm 3.1): ensures that an event that is disseminated by at least one correct process is seen by every correct process within a certain period of time, helping satisfy the probabilistic agreement property;

2. Ordering (Algorithm 3.2): ensures that every correct process delivers the events they have seen in the same order, and within a certain period of time, satisfying the total order property.

**Algorithm 3.1:** EpTO Dissemination Component

```
1  initially
2      view ← ... // system parameter: set of uniformly random correct peers
3      K ← ... // system parameter: fanout
4      TTL ← ... // system parameter: number of rounds
5      next_ball ← ∅ // set of events to be relayed in the next round
6  procedure DISSEMINATE(event)
7      event.ttl ← 0
8      event.ts ← GETCLOCK()
9      next_ball[event.id] ← event
10 upon receive Ball : ball
11     foreach event ∈ ball do
12         if event.ttl < TTL then
13             if event.id ∈ next_ball then
14                 if next_ball[event.id].ttl < event.ttl then
15                     next_ball[event.id].ttl ← event.ttl
16             else
17                 next_ball[event.id] ← event
18         UPDATECLOCK(event.ts)
19 run task every δ units of time
20     foreach event ∈ next_ball do
21         event.ttl ← event.ttl + 1
22     if #next_ball > 0 then
23         peers ← RANDOM(view, K)
24         foreach peer ∈ peers do
25             send Ball : next_ball to peer
26     ORDEREVENTS(next_ball)
27     next_ball ← ∅
```

Events are disseminated in an epidemic manner, being broadcast to a random subset of the processes periodically. The stopping condition for this periodic broadcast is based on the $ttl$ of each event, and events are eligible for delivery once said $ttl$ reaches a certain threshold.

In addition to this $ttl$, events also have a timestamp, $ts$, and it is based on this timestamp that the order of delivery is chosen.

The dissemination component ensures events are seen by every correct process (with high probability), and the ordering component then ensures that every correct process delivers them in the same order.

EpTO is however not suitable for operation in environments subject to Byzantine faults, as it is designed with solely fail-stop in mind. For that reason, to make it suitable to be used in the environments being discussed, it needs to be adapted.

**Algorithm 3.2:** EpTO Ordering Component

```
1  initially
2      received ← ∅ // Map of event id → event
3      delivered ← ∅ // Set of delivered events
4      last_delivered ← 0
5  procedure OrderEvents(ball)
6      foreach event ∈ received do
7          received[event.id].ttl ← received[event.id].ttl + 1
8      foreach event ∈ ball do
9          if event.id ∉ delivered and event.ts ≥ last_delivered then
10             if event.id ∈ received then
11                 if received[event.id].ttl < event.ttl then
12                     received[event.id].ttl ← event.ttl
13             else
14                 received[event.id] ← event
15     min_queued ← ∞ // Oldest non-deliverable event
16     deliverable ← ∅
17     foreach event ∈ received do
18         if IsDeliverable(event) then
19             deliverable ← deliverable ∪ {event} // Preliminarily deliverable
20         else if event.ts < min_queued then
21             min_queued ← event.ts
22     foreach event ∈ deliverable do
23         if event.ts ≥ min_queued then
24             deliverable ← deliverable \ {event}
25         else
26             received ← received \ {event} // Will be delivered
27     foreach event ∈ Sort(deliverable) do
28         delivered ← delivered ∪ {event}
29         last_delivered ← event.ts
30         EpTO-Deliver(event)
```

**Algorithm 3.3:** EpTO Utilities

```
1  initially
2      clock ← 0
3  procedure IsDeliverable(event)
4      return event.ttl > TTL
5  procedure GetClock()
6      clock ← clock + 1
7      return clock
8  procedure UpdateClock(timestamp)
9      if timestamp > clock then
10         clock ← timestamp
```

## 3.3 Adapting EpTO

While not a functional standalone algorithm, this will serve as a stepping stone to building TBO, by strengthening EpTO for operation in an environment subject to Byzantine Faults. This will be done by taking a look at some ways processes could exhibit faulty behavior that would violate some of EpTO's properties.

### 3.3.1 Preventing shortening dissemination

An attack that would violate EpTO's guarantees, specifically probabilistic agreement, is to have a faulty process disseminate an event with a high $ttl$, one that would be high enough to ensure not every correct process will receive it before dissemination stops, e.g. $TTL$ or $TTL - 1$. As it would not reach every process prior to ending dissemination, only some correct processes would deliver it, violating the probabilistic agreement property.

This attack relies on two key conditions: dissemination stops before being seen by every correct process, and the correct processes deliver the event regardless of dissemination outcome. Since an event sent with a tampered $ttl$ is indistinguishable from an event only seen after $TTL - 1$ rounds, stopping the delivery of the event could lead to incorrect outcomes and is not feasible. The solution, therefore, is to ensure the dissemination never stops before being seen by every correct process.

There are two scenarios where a correct process may be misled into stopping dissemination due to input from a faulty process:

1. The correct process is observing the event for the first time, and therefore disseminates the incremented $ttl$, which may have been set by an attacker.
2. It observed the event twice in the current round, and uses the highest $ttl$, which may have been set by an attacker, to decide the $ttl$ for the disseminated event.

Both of these scenarios occur due to lines 13-17 of Algorithm 3.1. To mitigate this attack, the aforementioned lines would need to be replaced by the lines described in Algorithm 3.4.

Doing so will make this attack impossible by tackling the two situations described above:

1. When the event is being observed for the first time, the condition present in line 1 of Algorithm 3.4 will be true. Therefore, $ttl$ will be set to 1 in line 3, which will always be either lower or equal to the received $ttl$, making a faulty process incapable of shortening the dissemination in this situation.
2. When the event is being observed twice in the same round, with a higher $ttl$ than it is supposed to have, the issue is mitigated by instead choosing the lowest received $ttl$ instead of the highest as the $ttl$ to be used in the next dissemination, in lines 6-7.

**Algorithm 3.4:** Preventing shortening dissemination

```
1 if event ∉ event_cutoff then
2     event_cutoff[event] ← 1
3     event.ttl ← 1
4 if event.ttl < TTL then
5     if event.id ∈ next_ball then
6         if next_ball[event.id].ttl > event.ttl then
7             next_ball[event.id].ttl ← event.ttl
8     else
9         next_ball[event.id] ← event
```

These changes are trivially shown to be sufficient, in that they seek to make any faulty dissemination better than no dissemination, by always resulting in either no impact if being received concurrently, or in an additional dissemination otherwise.

However, while dissemination may no longer be stoppable, delivery may still be anticipated by propagating the event with a high $ttl$. When this is done, as the decision of whether a given event is deliverable or not is made based on whether the $ttl$ is equal to or greater than $TTL$, it may be delivered early. In a vacuum, a single event being delivered early cannot lead to issues (as long as dissemination is unhindered), but if other events are also being disseminated at the same time, this early delivery will make all preceding events undeliverable.

This vulnerability is present due to lines 8-14 of Algorithm 3.2 in conjunction with lines 3-4 of Algorithm 3.3. To mitigate this attack, the aforementioned lines of Algorithm 3.2 would need to be amended, to be as the lines presented in Algorithm 3.5.

**Algorithm 3.5:** Preventing early delivery

```
1 foreach event ∈ ball do
2     if event.id ∉ delivered and event.id ∉ received then
3         if event.ts ≥ last_delivered then
4             received[event.id] ← event
```

As the $ttl$ of a given event was reset to $1$ the first time it was seen, which will also be the first time it reaches the ordering component, an event will only become deliverable when $TTL$ or more rounds have elapsed since the event was first seen.

This way, regardless of any changes to the $ttl$ of a given event made by a faulty process, events will never be delivered early.

### 3.3.2 Preventing infinite dissemination

This attack is not damaging to the correctness of the algorithm, but it would make it open to an extremely simple denial of service, where correctness of the dissemination component would ensure amplification to at least the size of the network.

This attack would unfold by continuously resetting the $ttl$ of a given event to the minimum whenever it was received, before re-disseminating it. As the correctness of the dissemination ensures, with high probability, that the event will be seen by every correct process before $TTL$ rounds have elapsed, a single change by a faulty process therefore leads to a high amount of network traffic.

This attack could be carried out indefinitely, continuously re-disseminating every past event, and even simultaneous dissemination by a correct process would be overridden by the measures implemented in Section 3.3.2, as the lowest $ttl$ would be the one used in the next dissemination.

This attack is possible due to the way we mitigated the vulnerability described in Section 3.3.1, in Algorithm 3.4. In order to make this attack impossible, lines 4-5 were added in Algorithm 3.6.

**Algorithm 3.6:** Preventing infinite dissemination

```
1  if event ∉ event_cutoff then
2      event_cutoff[event] ← 1
3      event.ttl ← 1
4  if event.ttl < event_cutoff[event] then
5      event.ttl ← event_cutoff[event]
6  if event.ttl < TTL then
7      if event.id ∈ next_ball then
8          if next_ball[event.id].ttl > event.ttl then
9              next_ball[event.id].ttl ← event.ttl
10     else
11         next_ball[event.id] ← event
```

These added lines will make the minimum $ttl$ increase as $event\_cutoff$ increases. However, we have not specified how is $event\_cutoff$ incremented. That can be done by altering our periodic task so that it also increments the cutoffs. Concretely, it can be done by replacing line 19 with the lines found in Algorithm 3.7.

**Algorithm 3.7:** Preventing infinite dissemination

```
1  run task every δ units of time
2      foreach event ∈ next_ball do
3          event.ttl ← event.ttl + 1
```

### 3.3.3 Preventing delayed dissemination

There is one last vulnerability that cannot be fixed with just changes to existing components in the variation of EpTO being presented: the ordering component relies on a timestamp to determine ordering, but there is no mechanism to ensure the event's dissemination has not been delayed or the timestamp been tampered with.

This creates a vulnerability similar to the one described in Section 3.3.1, trying not to stop the dissemination but instead making it start far too late for the ordering component to behave correctly. The dissemination component will ensure that every correct process eventually observes an event, $E'$, if any correct process observes it, but this delayed dissemination will create a window during which $E'$ may be delivered before a preceding event, $E$, has been observed. As a consequence, some correct processes may deliver $E$, but others, by already having delivered $E'$, will not, violating the probabilistic agreement property.

An exploit of this vulnerability is illustrated in Figure 3.1. In that example, an event $E'$ was being disseminated throughout the network, and had been seen by both $P_1$ and $P_3$, and is nearing delivery eligibility. However, $P_2$, a faulty process, creates a second event, $E$, with a timestamp older than that of $E'$. What can happen at this point is that $P_2$ begins dissemination of $E$ towards $P_1$, but given the reduced amount of time left until $E'$ is delivered by $P_3$, $P_3$ delivers $E'$ without being aware of $E$.

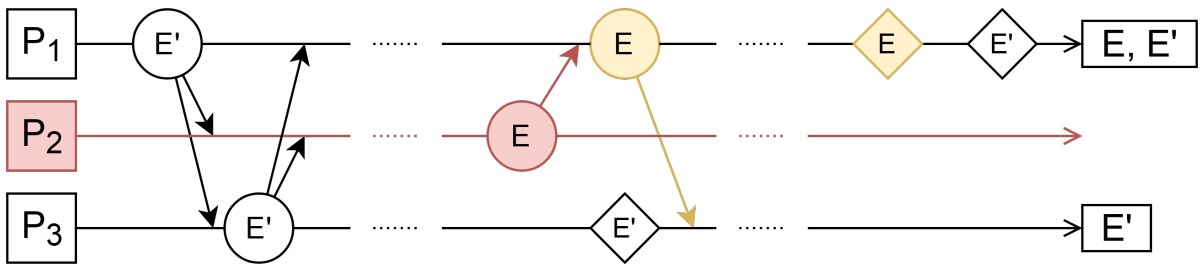



**Figure 3.1:** Exemplified attack, with an attacker (P2) using delayed dissemination (circle) of an event (E), leading to incorrect delivery (rhombus) order for processes that only see it after delivery of a future event (E').

As a consequence of this delivery, we have reached a state in which the probabilistic agreement property failed: at least one correct process has delivered $E$, but not all correct processes will eventually deliver $E$, as they have already delivered $E'$. If attempting to respect the probabilistic agreement property $P_3$ then decided to deliver $E$, it would instead violate the total order property, as $P_1$ and $P_3$, both correct processes, would have inconsistent delivery orders for $E$ and $E'$.

Do note, however, that there is no way for $P_1$ to know that the decision to deliver $E$ would lead to this outcome, as it is impossible for $P_1$ to determine that $E$ was forged and/or its dissemination delayed by a faulty process. To $P_1$, $E$ was an event that it had not yet seen, indistinct from any other. It does not even need to begin dissemination particularly late: it just needs to be late enough that, with the remaining

number of rounds before the delivery of $E'$, not every correct process will be made aware of $E$ before $E'$ is delivered.

This vulnerability has two ways in which it can be exploited: through a forged timestamp, in which an event is created with a timestamp that no correct process would produce at the time, and through delayed dissemination, in which an event is created with a correct timestamp but its dissemination is delayed. To a correct process these look exactly alike but they are distinct in that simply ensuring events are created with a valid timestamp, such as could be done by the use of an RFC3161 [36] compliant service, would not be sufficient, as a faulty process could then delay the dissemination of the event with a guaranteed-valid timestamp.

## 3.4 TBO

The proposed solution to the problems presented in Section 3.3.3 is the introduction of a third component: the endorsement component. It specifically seeks to ensure that the two described vulnerabilities are mitigated by ensuring the following:

1. If an event, $e$, has a timestamp that precedes that of any other event which will be deliverable before $e$ is observed by every correct process, then $e$ must not obtain sufficient endorsements;

2. If any process obtains sufficient endorsements to begin dissemination, then at least one correct process has also obtained sufficient endorsements to begin dissemination.

This new component is introduced in Algorithm 3.8, and it follows the simplest implementation that offers the guarantees required.

The chosen implementation, despite its high overhead due to a very high number of messages (at most $n^2$ messages, with $n$ being the number of endorsers), is a double-broadcast. From a high level perspective, the behavior is as follows:

1. When an event is seeking endorsements, an endorsement request is broadcast to every endorser;

2. Upon receiving an endorsement request or an endorsement, if the event is valid and being received for the first time, an endorsement is generated, stored, and all endorsements for that event received so far are broadcast to every endorser;

3. Upon receiving an endorsement, in addition to the previous point, it is also stored;

4. When an event has received $minimum\_endorsements$ endorsements, the process(es) that have seen sufficient endorsements begin dissemination.

**Algorithm 3.8:** TBO Endorsement Component

```
1  initially
2      endorsers ← ... // parameter: number of endorsers
3      minimum_endorsements ← ... // parameter: number of endorsements required before acceptance
4      hard_timestamp_margin ← ... // parameter: hard margin, always rejected if crossed
5      soft_timestamp_margin ← ... // parameter: soft margin, randomly rejected if crossed
6      soft_accept_probability ← ... // parameter: probability to reject once soft margin is crossed
7      received_endorsements ← ∅ // map of events → received endorsements
8  procedure TBO-BROADCAST(event)
9      event.ts ← GETCLOCK()
10     foreach position, endorser ∈ GETENDORSERS(event) do
11         send EndorsementRequest : event, position to endorser
12 upon receive EndorsementRequest : event, position
13     if position ∉ received_endorsements and GetEndorsers(event)[position] = self then
14         if
             event.ts > GetClock() - hard_timestamp_margin and event.ts < GetClock() + hard_timestamp_margin
           then
15             if event.ts > GetClock() - soft_timestamp_margin or GetRandom(0, 1) < soft_accept_probability
               then
16                 endorsement ← CREATEENDORSEMENT(event, position)
17                 received_endorsements[event][position] ← endorsement
18                 foreach pos, endorser ∈ GETENDORSERS(event) do
19                     send Endorsement : event, pos, received_endorsements[event] to endorser
20 upon receive Endorsement : event, position, endorsements
21     send EndorsementRequest : event, position to self
22     foreach endorsement ∈ endorsements do
23         event ← endorsement.event
24         position ← endorsement.position
25         if VALIDATEENDORSEMENTS([endorsement]) then
26             received_endorsements[event][position] ← endorsement
27     if #received_endorsements[event] ≥ minimum_endorsements then
28         event.endorsements ← received_endorsements[event]
29         DISSEMINATE(event)
```

This high level overview glosses over some important aspects that need to be considered, regarding how endorsers are chosen, when are events considered valid, and when are enough endorsements received.

An endorser is a process on the network tasked with approving a given event prior to dissemination can begin. The method through which they are chosen is not defined, but it does need to be, most crucially, deterministic. An approach that would satisfy these requirements is to use rendezvous (HRW) hashing [37] or a special case of rendezvous hashing, consistent hashing [38].

Whichever the chosen approach, the goal is the same: ensure that a certain event can be mapped to a certain group of processes, which will become its endorsers, in a deterministic manner. To generate an endorsement, all that needs to be done is that an endorser digitally signs the event, and by confirming that the signature is valid and the endorser is in the list of endorsers for that event, the endorsement is considered valid.

The number of endorsers, the variable $endorsers$, is a configurable value largely dependent on the desired security level of the network: the higher the number of endorsers, the more secure, but at the same time the higher the network bandwidth required for the endorsement. In Section 4.1 this will be analyzed in more detail, but there are diminishing returns, and having more than 250 endorsers has a negligible impact on security while massively increasing overhead due to the quadratic increase in number of messages.

Another critical aspect is that of how are events determined to be valid, specifically in respect to their timestamp. For this determination there are two key parameters: $hard\_timestamp\_margin$ and $soft\_timestamp\_margin$, which I will refer to as the hard and soft margins respectively.

The hard margin is the simplest of the two, with its only goal being to make sure timestamps need to be within a given margin for them to be endorsable, in order to ensure the first property of the endorsement component. More concretely the timestamp needs to be in the interval $]clock - hard\_timestamp\_margin; clock + hard\_timestamp\_margin[$ for the event's endorsement to even be considered.

The soft margin is more complex, and it seeks to ensure the second property (and indirectly the first), by attempting to mitigate an attack that would invalidate it. Similarly to the attack described in Section 3.3.3 and shown in Figure 3.1, in the absence of the soft margin, a faulty endorser could simply request endorsements from $minimum\_endorsements - 1$ endorsers, but this time right before the hard margin cut off future endorsements. The result is that $minimum\_endorsements - 1$ would be generated, but no future endorsements would be made by correct processes, meaning the event would not reach the required threshold. However, since the faulty process is also an endorser, they could afterwards generate an endorsement, putting the total number above the required threshold. This would replicate the aforementioned attack, bypassing the endorsement component, and putting a fully-endorsed event

under control of a faulty process that could begin its dissemination whenever it would like.

The solution to this problem is the introduction of a soft margin. Since this vulnerability would be exploited by requesting endorsements right before the hard margin, the soft margin exists to make behavior during this vulnerable period unpredictable. Rather than always endorsing (or never endorsing), while within the soft margin, processes have a probability to endorse. Whenever put in a situation where they would be endorsing an event, they instead generate a random number between 0 and 1, and if smaller than $soft\_accept\_probability$ they will endorse, but will do nothing otherwise. If 10 different endorsers request an endorsement, then this probability will be rolled for 10 times, meaning even if a low probability is used, once a certain critical mass of endorsers is reached, a cascading effect occurs and all or nearly all correct endorsers eventually endorse.

The last important factor is how many endorsers are required to endorse a given event, which is a configuration parameter, $minimum\_endorsements$. Looking at the extremes, it is trivial that neither a single endorser, nor all endorsers, are valid options: with 1 endorser required then neither property can be ensured, and with all endorsers required then the second property cannot be guaranteed, as any faulty endorser will be able to control when dissemination begins by only endorsing at that time.

A good initial threshold is 50% of endorsers in absence of attacks, precisely because it is the farthest between the two first failure scenarios, evidenced by both edge cases: faulty endorsers able to endorse on their own (if more faulty endorsers than the endorsement threshold), and faulty endorsers being able to decide whether an endorsement is successful (if less correct endorsers than the endorsement threshold). However, due to the attacks described in Section 3.5, the soft margin also comes into play and is crucial in deciding the ideal threshold.

As previously mentioned, the soft-margin helps introduce a cascading effect where all or nearly all correct processes endorse once a critical mass of them do. The threshold therefore needs to be higher than this critical mass, but at the same time lower than the number of correct processes to avoid the other edge cases in which endorsement may fail. This precise threshold will be largely dependent on the number of endorsers and the expected percentage of faulty processes, and needs to be calculated depending on these parameters, with a universal optimal value not existing.

The dissemination component, Algorithm 3.9, incorporates the changes outlined in Section 3.3, in addition to lines 12-13 which seek solely to ensure that the event has gone through the endorsement component and has been validated by it.

The ordering component, Algorithm 3.10, also incorporates the changes outlined in Section 3.3. An additional change that was not described elsewhere is the renaming of $ttl$, solely in the ordering component, into $age$, to have a name more fitting to its purpose.

**Algorithm 3.9:** TBO Dissemination Component

```
1  initially
2      view ← ... // system parameter: set of uniformly random peers
3      K ← ... // system parameter: fanout
4      TTL ← ... // system parameter: number of rounds
5      next_ball ← ∅ // set of events to be relayed in the next round
6      event_cutoff ← ∅ // map of events → minimum TTL expected
7  procedure DISSEMINATE(event)
8      event.ttl ← GETCLOCK()
9      next_ball[event.id] ← event
10 upon receive Ball : ball
11     foreach event ∈ ball do
12         if
             #event.endorsements < minimum_endorsements or not ValidateEndorsements(event.endorsements)
             then
13             continue// skip event
14         if event ∉ event_cutoff then
15             event_cutoff[event] ← 1
16             event.ttl ← 1
17         if event.ttl < event_cutoff[event] then
18             event.ttl ← event_cutoff[event]
19         if event.ttl < TTL then
20             if event.id ∈ next_ball then
21                 if next_ball[event.id].ttl > event.ttl then
22                     next_ball[event.id].ttl ← event.ttl
23             else
24                 next_ball[event.id] ← event
25         UPDATECLOCK(event.ts)
26 run task every δ units of time
27     foreach event ∈ next_ball do
28         event.ttl ← event.ttl + 1
29     foreach event ∈ event_cutoff do
30         event_cutoff[event] ← event_cutoff[event] + 1
31     if #next_ball > 0 then
32         peers ← RANDOM(view, K)
33         foreach peer ∈ peers do
34             send Ball : next_ball to peer
35     ORDEREVENTS(next_ball)
36     next_ball ← ∅
```

**Algorithm 3.10:** TBO Ordering Component

```
1  initially
2      received ← ∅ // Map of event id → event
3      delivered ← ∅ // Set of delivered events
4      last_delivered ← 0
5  procedure OrderEvents(ball)
6      foreach event ∈ received do
7          received[event.id].age ← received[event.id].age + 1
8      foreach event ∈ ball do
9          if event.id ∉ delivered and event.id ∉ received then
10             if event.ts ≥ last_delivered then
11                 event.age ← 1
12                 received[event.id] ← event
13     min_queued ← ∞ // Oldest non-deliverable event
14     deliverable ← ∅
15     foreach event ∈ received do
16         if IsDeliverable(event) then
17             deliverable ← deliverable ∪ {event} // Preliminarily deliverable
18         else if event.ts < min_queued then
19             min_queued ← event.ts
20     foreach event ∈ deliverable do
21         if event.ts ≥ min_queued then
22             deliverable ← deliverable \ {event}
23         else
24             received ← received \ {event} // Will be delivered
25     foreach event ∈ Sort(deliverable) do
26         delivered ← delivered ∪ {event}
27         last_delivered ← event.ts
28         TBO-Deliver(event)
```

**Algorithm 3.11:** TBO Utilities

```
1  initially
2      clock ← 0
3  procedure IsDeliverable(event)
4      return event.age > TTL
5  procedure GetClock()
6      return clock
7  procedure UpdateClock(timestamp)
8      if timestamp > clock then
9          clock ← timestamp
```

## 3.5 Known attack vectors

The attacker that is being considered is one that is fully aware of the latency of every point-to-point connection on the network and the state of each processes' internal clock, being able to send messages that will be delivered precisely when the attacker determines they should be. However, this attacker is unable to subvert cryptographic primitives, e.g. calculate hash collisions, as an attacker able to do so could defeat any mechanism that relies on digital signatures.

Every known attack that this attacker can perform revolves around attempting to compromise the endorsement component (Algorithm 3.8), violating its provided guarantees, therefore invalidating the guarantees offered by the dissemination component (Algorithm 3.9) and subsequently the ones ensured by the ordering component (Algorithm 3.10).

The guarantee that the attacker would want to violate is that if any process has collected the required amount of endorsements, then at least one correct process will also collect the required amount of endorsements in time for dissemination.

To show that violating this guarantee is sufficient, suppose, by contradiction, that an attacker managed to violate this property. In that case, the attacker is in control of an endorsed event[1] (E) that no other correct process has seen along with its endorsements. The attacker can then delay the dissemination of this event, beginning its dissemination precisely when a later event (E') would delivered by `TBO-Deliver`. By waiting until this critical moment, certain correct processes will deliver E' prior to seeing E, whereas other correct processes will deliver E and only then will they deliver E'. A simplified example of this attack is shown in Figure 3.1 when motivating the need for an ordering component.

### 3.5.1 Crafting an attack

Any successful attack will need to be attacking the endorsement component (Algorithm 3.8). The reason this is true is because the only other component subject to attack is the dissemination component[2] (Algorithm 3.9), yet any valid message the attacker sends in that situation will only help further disseminate events, which is undesirable towards the goal of violating agreement.

Therefore, we need to consider all the possible attacks that attempt to compromise the endorsement component. The goal of these attacks will always be to reach the end of the endorsement stage in a vulnerable situation: not enough processes have endorsed the event but, should the attacker decide to endorse it (and not necessarily broadcast said endorsement), then the required number of endorsements will be reached. Since the attacker is then in control of the endorsement, the dissemination can be delayed, compromising the algorithm.

[1] An event with sufficient endorsements to be disseminated.

[2] The ordering component (Algorithm 3.10) handles no communication, and fully relies on the other components to ensure its correctness, not being subject to direct attacks.

Looking at the endorsement component, the first step would be the creation of the event. It should be clear that if an attacker is the one creating the event, then the system is more vulnerable than with a correct process creating it. An event created by any correct process will always be correctly endorsed regardless of what the attacker does[3], and is not subject to attack.

The attacker therefore has to be the event creator. In addition to that, an attacker not sending out endorsement requests, or only sending them after the hard margin, leads to a failed attack as the event will not be endorsed. However, if the attacker sends out endorsement requests in the beginning of the endorsement, then this request will be re-broadcast and every correct endorser will endorse it, with the attack failing as well.

The attacker must therefore send requests only right before the soft margin, so that any re-broadcast will arrive after the soft margin, and/or after it but before the hard margin. The instant(s) in which to make such requests, or their receiver(s), are determined by the attacker.

The exact combination of attack parameters (i.e. when and who to send endorsement requests or endorsements to) that would lead to the strongest attack is not easily determinable, as that would depend on parameters such as the number of endorsers. However, it can be concluded that the strongest attack will be as follows:

1. Create an event and do not broadcast endorsement requests;
2. As the soft margin approaches, possibly send out an endorsement request to one or more correct endorsers, such that their own endorsement requests will not reach other correct endorsers prior to the soft margin;
3. While between the soft margin and the hard margin, in any given instant, do one of the following:
   (a) Do nothing;
   (b) Send out one or more endorsements to one or more correct processes;

One such attack is shown in Figure 3.2, in that case being successful, as the attacker has gained control of the endorsement process by becoming the endorser capable of determining the success or failure of the endorsement, being able to delay the endorsement and therefore dissemination. Do note however that the attack was only successful because the attacker was lucky, and neither D nor E randomly endorsed after seeing A and C's endorsements.

While this attack may sound overly generic, which it is, a competent attacker is able to calculate all possible attack parameters, and perform the attack using the combination of parameters most likely to lead to a successful attack. Therefore, we need to calculate the likelihood of success of such an attack to understand how vulnerable the algorithm as described is.

[3]The only exception is when the attacker controls enough endorsers to make the transaction unendorsable.

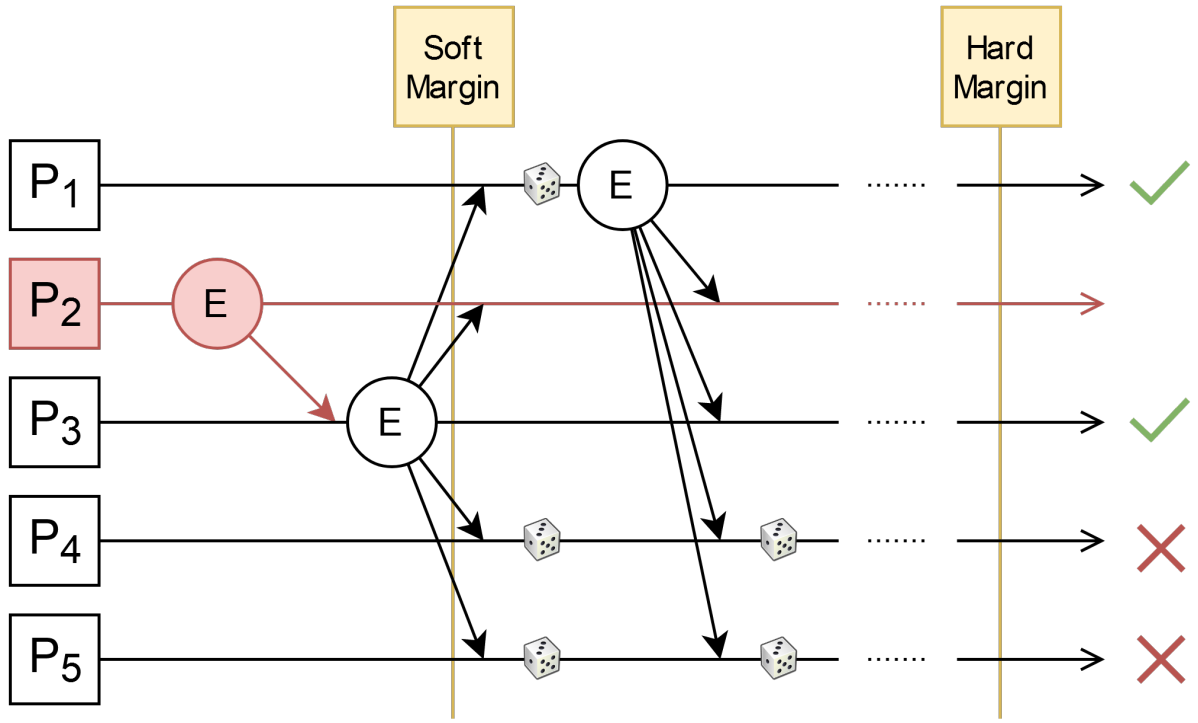



**Figure 3.2:** A successful attack with a 3-endorsement threshold where the attacker (B), who is both the creator and an endorser, sends a precisely timed endorsement request (but not endorsement), resulting in the attacker being the only process to have collected 3 endorsements.

### 3.5.2 Analysis

The thought process throughout the rest of this subsection is to understand how the system evolves as an attack as described above takes place. As the system is dynamic, not only will certain probabilities change, but so will the way in which they are calculated.

The following are the symbols that will be most frequently used throughout this subsection:

- $N \leftarrow$ endorsers, regardless of whether they have endorsed or not
- $C \leftarrow$ correct endorsers, regardless of whether they have endorsed or not
- $f_U \leftarrow$ faulty endorsers who have not sent any endorsement to any correct process
- $f_E \leftarrow$ faulty endorsers who have sent an endorsement to one or more correct processes
- $\#T \leftarrow$ required number of endorsements before event is considered endorsed
- $P(E_1) \leftarrow$ probability of a given correct endorser endorsing after seeing one endorsement request
- $P(E_N) \leftarrow$ probability of a given correct endorser endorsing
- $NM \leftarrow$ correct endorser that receives an endorsement request from the attacker, before the soft margin
- $SM \leftarrow$ correct endorser that receives an endorsement from the attacker, between the soft and hard margins

- $P(A) \leftarrow$ probability of the attack being successful

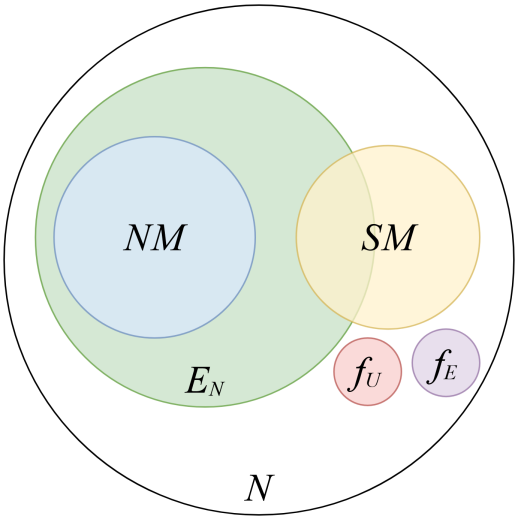



**Figure 3.3:** A simplified visualization of the different sets, and how they overlap. $SM \cap E_N$ may be $\emptyset$, $SM$, or a subset of $SM$. $T \subseteq N$, but is unrepresented as its relation with the other sets is non-trivial and dynamic.

In Figure 3.3 the groups as described above are illustrated, to help visualize how they interact and overlap. The attack is successful if, at the end of the endorsement, $\#E_N + \#f_E < \#T$ and $\#E_N + \#f_E + \#f_U \geq \#T$. In other words the attack is successful if and only if an insufficient number of correct processes have endorsed the event, but the endorsement by all faulty processes would push the number of endorsements above the threshold.

In addition to the above symbols, there are some probabilities that will appear that are trivially solved:

- $P(E_1|NM) = 1 \implies P(E_N|NM) = 1$, as before the soft margin correct processes will always endorse after receiving an endorsement request.
- $NM \cap SM = \emptyset \implies P(NM|SM) = 0 \wedge P(SM|NM) = 0$, since $NM$ and $SM$ are disjoint.

At the beginning of the endorsement stage, the attacker will request endorsements from as many endorsers as it desires, including possibly none. These endorsement requests will be sent such that they are received precisely before the soft margin, with the intent that any following endorsement requests will reach other processes only after the soft margin. If this were not true, and those endorsement requests arrived before the soft margin, then the event would be endorsed by all correct processes and the attack would fail.

The probability of any correct process having endorsed the event after the attacker has sent out its endorsement requests is therefore given by the probability of having been chosen to receive these endorsement requests:

$$\begin{aligned} P(E_N) &= P(E_N|NM) * P(NM) \\ &= P(NM) \end{aligned} \tag{3.1}$$

As the system is partially synchronous, requiring a known upper limit to network latency but no clock synchronization, correct processes may be executing out of step. As such, we will randomly group our processes into $S$ sub-groups, with messages delivered to the network having their effects delayed, such that subgroup $s+1$ will execute after subgroup $s$. The notation $C_s$ will be used to represent the correct processes in subgroup $s$, and $P(X)_s$ will be used to represent the probability of $X$ within subgroup $s$.

If any correct process endorses the event, it will also send out an endorsement request. Since every correct process will be in the soft margin, their probability of endorsing following the reception of any endorsement request can be given by the complimentary of the probability of not endorsing, which is the probability of not endorsing after seeing one request, $P(\neg E_1|\neg NM)$, to the power of the number of endorsers so far, $(\sum_{k=0}^{S} \#C_k * P(E_N)_k) + \#f_E$. The probability of endorsement for these correct processes can therefore be given by:

$$P(E_N|\neg NM)_s = 1 - P(\neg E_1|\neg NM)^{(\sum_{k=0}^{S} \#C_k * P(E_N)_k) + \#f_E} \tag{3.2}$$

However, the attacker may also send out $n$ endorsement requests to some of the processes. The probability of these processes endorsing the event after observing these endorsement requests follows the same thought process, and is:

$$P(E_N|SM) = 1 - P(\neg E_1|\neg NM)^n \tag{3.3}$$

As a requirement for sending endorsement requests within the soft margin is to also send an endorsement, $n$ processes have gone from being in $f_U$ to being in $f_E$, and so:

$$\begin{aligned} \#f_E &= \#f_E + n \\ \#f_U &= \#f_U - n \end{aligned} \tag{3.4}$$

The probability of receiving one such request is $P(SM|\neg E_N) = 1$, as once an endorsement is sent, every correct process will eventually see it. The probability of endorsing if the process was not among the ones who received the request prior to the soft margin is redefined as:

$$P(E_N|\neg NM)_s = P(E_N|\neg NM)_s + P(E_N|SM) * P(SM|\neg E_N) * P(\neg E_N|\neg NM)_s \tag{3.5}$$

This endorsement probability is only applicable to correct processes seeing the event after the soft margin, as every correct process that has seen it before the soft margin will have endorsed it, as

$P(E_N|NM) = 1$. The probability of correct processes endorsing can therefore be given by:

$$\begin{aligned} P(E_N)_s &= P(E_N|NM)_s * P(NM) + P(E_N|\neg NM)_s * P(\neg NM) \\ &= P(NM) + P(E_N|\neg NM)_s * P(\neg NM) \end{aligned} \tag{3.6}$$

Throughout the endorsement, new correct endorsers will broadcast additional endorsement requests whenever they themselves also endorse, and the attacker may also send endorsement requests to as many correct processes as it desires at any moment, and so Equations (3.2) to (3.6) need to be recalculated iteratively, and recalculated for each subgroup $s$ in each of those iterations.

Finally, we can calculate the value of $P(E_N)$, with each subgroup contributing proportionally to its size:

$$P(E_N) = \sum_{s=0}^{S} P(E_N)_s * (\#C_s/\#C) \tag{3.7}$$

As the system will evolve over a bounded period of time we need to calculate $P(E_N)$ after a certain number of iterations, and not the value for which all equations are true, which would represent $P(E_N)$ after a boundless period of time. The number of iterations should be approximately the difference between the soft margin and the hard margin, divided by the average latency on the network.

After the iterations, the correct processes will enter the hard margin and all further endorsement requests will be rejected, and so the number of correct endorsers will have stabilized. The probability of the attack being possible, $P(A)$, is the probability of the number of correct endorsers being within the interval in which the endorsement is vulnerable.

The minimum number of correct endorsers that have endorsed the event for the attack to be possible is:

$$\#E_{min} = max(\#T - \#f_U, 0) \tag{3.8}$$

And the maximum number is:

$$\#E_{max} = min(\#T - \#f_E - 1, \#C) \tag{3.9}$$

As a process either endorses or does not, a binomial distribution is applicable, and the probability of the attack being successful is the probability of the number of endorsers being between $\#E_{min}$ and $\#E_{max}$, which is:

$$P(A) = \sum_{k=\#E_{min}}^{\#E_{max}} \binom{\#C}{k} * P(E_N)^k * (1 - P(E_N))^{\#C-k} \tag{3.10}$$

# 4

# Evaluation

## Contents

In this chapter we will be going over the main contribution: a BFT Total Order algorithm, based on an existing epidemic algorithm, EpTO [4], introduced in Section 2.3.3. We will begin by looking at the properties that we seek to ensure, followed by EpTO and, by making modifications in order to mitigate certain attacks, we will arrive at TBO. After that, we will look at how TBO might be attacked, and demonstrate that these attacks, in very strict conditions, are possible.

## 4.1 Theoretical evaluation

The theoretical evaluation is based on the analysis presented in Section 3.5.2, with the goal of understanding whether a feasible attack on the endorsement component could be found in every or nearly every parameter configuration.

In this analysis the attacker, much like throughout Section 3.5, is fully aware of network latencies, and processes are behaving synchronously and suffering minimal clock drift to the point where the attacker is fully aware of their internal clocks.

The process through which this was evaluated is borderline *bruteforce*: for a wide range algorithm parameters (endorsement threshold given by $minimum\_endorsements$, and soft margin endorsement probability given by $soft\_accept\_probability$), all possible attacks were calculated. Then, for every possible attack, the analysis in Section 3.5.2 was calculated with a 1 000 decimal unit precision, and the probability of that attack succeeding was determined.

It is important to note that due to how exponentially large (*factorially* large) the number of attacks grows as the number of faulty processes increases, this severely limited our ability to test with a large number of faulty processes. Concretely, for 50 processes out of which 45 are correct, there are 5 632 067 attacks under consideration. Increasing that to 60 processes, out of which 54 are correct, there are now 48 989 666 attacks under consideration.

When carrying out this testing, the system parameters were fixed at 45 correct processes and 5 faulty processes, for a faulty percentage of 10%. In total, 41 266 combinations of algorithm parameters were tested, each against the aforementioned 5 632 067 possible attacks.

To verify the results, individual randomly selected rows were taken and manually calculated, and the automated and manual results matched.

The results were, however, not favorable, despite the low number of faulty processes. These results are present in Table 4.1.

In every tested algorithm parameter combination, an attack was always found that had a probability of success greater or equal to 0,1%, which was used as a condition to terminate the calculations for that specific set of parameters early. In practice, this means that every tested combination of parameters was vulnerable to at least one attack with a success chance of 0.1% or higher.

**Table 4.1:** Portion of results of theoretical evaluation. Correct and Faulty refer to the number of Correct and Faulty processes, Threshold refers to the required number of endorsers before an event is endorsed, Soft Margin Endorse Prob. refers to the probability of endorsing in soft margin.

| Correct | Faulty | Threshold | Soft Margin Endorse Prob. | Attack Success Prob. |
|---|---|---|---|---|
| 45 | 5 | 26 | $5.54 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 31 | $8.62 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 41 | $5.79 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 37 | $4.76 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 42 | $9.32 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 27 | $7.91 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 32 | $6.72 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 34 | $9.28 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 31 | $4.46 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 36 | $6.45 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 43 | $7.85 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 32 | $6.31 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 25 | $2.98 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 38 | $6.39 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 28 | $9.65 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 34 | $3.85 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 31 | $8.02 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 27 | $1.07 \cdot 10^{-2}$ | $1 \cdot 10^{-3}$ |
| 45 | 5 | 24 | $9.6 \cdot 10^{-3}$ | $1 \cdot 10^{-3}$ |

However, this doesn't necessarily mean TBO is unsafe, but it does mean future work needs to be done to ascertain how easily can TBO be attacked in real-world scenarios. The motivation behind this claim is that the environment in which the theoretical analysis is done is unrealistically strong for the attacker: being able to exploit vulnerabilities that rely on very precise knowledge of the network latency and internal clocks of all network participants is unrealistic.

It is, however, not known exactly how much of this information is actually required to launch a successful attack, and whether a weaker scenario would only make the attack harder but feasible or whether it would make it nearly impossible.

## 4.2 Experimental evaluation

The experimental evaluation is based on simulating the execution of the TBO algorithm, as presented in Algorithm 3.8, Algorithm 3.9, Algorithm 3.10, and Algorithm 3.11, and gathering statistics about its runtime operation.

The simulation was performed on Corten [39], an event-based simulator written in Rust. A simulator-adapted version of TBO (lacking security verifications, proper signatures, among other non-essential features) was implemented in Corten, and a single event was disseminated per simulation. The simulator

does not have a set unit of time, but the values used were similar to those expected of a real world scenario if they were in milliseconds (ms), and that unit will be used throughout this section.

The simulator was configured to have the network latency set to 100ms, and network jitter being given by a log-normal distribution with a standard deviation of 10ms. The internal clocks of processes exhibited some drift, with this drift being randomly and uniformly distributed between -2.5ms and +2.5ms, applied whenever any call was made. The periodic task executed by each process is executed every 200ms.

The faulty processes did not exhibit any Byzantine behavior, instead behaving as crash-fault by not transmitting any messages. They were not, however, detected to have failed, and every correct process believed they were still operational and continued to send events to faulty processes.

In the graphs being presented only a single variable is changed, with the others remaining constant: 10 000 processes, 100 endorsers, 10% faulty processes, 60% endorsement threshold, and 1% packet loss.

On the first instant after simulation begins, one (and only one) correct process creates an event and executes the `TBO-Broadcast` function. The simulation then unfolds until every correct process delivers the event. This simulation was repeated at least 33 times with different seeds (0-32) for every test case, with some simulations being repeated up to 1 000 times. In none of the simulations did any correct process fail to deliver the event, even in the worst conditions tested (8% packet loss and 40% faulty processes).

When varying solely the number of processes between 100, 1 000, 10 000, and 100 000, there are significant differences in the average number of messages being sent by each process, as shown in Table 4.2. The number of endorsers is fixed at 50 for this comparison, throughout all sizes.

While at first the increase in the average number of messages amounts to almost doubling, this increase is logarithmic despite the network size increasing tenfold. This property is inherited from EpTO, with the average number of messages growing logarithmically with the number of processes.

**Table 4.2:** How the average number of messages changes as the number of processes changes.

| Processes | Avg. Messages Sent |
|---|---|
| 100 | 697.96 |
| 1,000 | 1,200.37 |
| 10,000 | 1,839.04 |
| $1 \cdot 10^5$ | 2,598.95 |

Another important set of metrics to observe are the time until first observation and the time until delivery, as the number of processes varies. These two metrics are shown in Figure 4.1 and Figure 4.2, presented as cumulative distribution functions, with the percentage of correct processes that have ob-

served or delivered the event (depending on the graph) up until that point represented in the Y-axis, and the time represented in the X-axis.

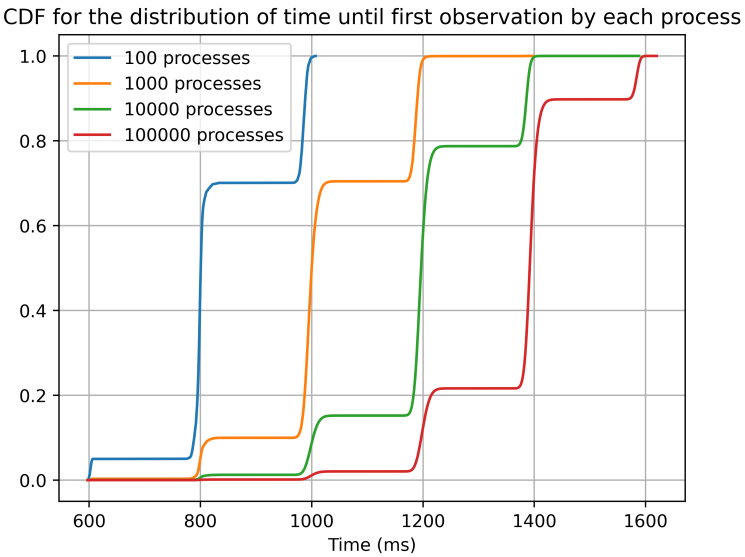



**Figure 4.1:** How the time until the event is first observed changes as the number of processes changes.

The increase in time until first observation is expected and largely due to the additional number of processes leading to another round (i.e. another execution of the periodic task) being required for as many processes to have observed the event. The fanout varies between 20 and 26 depending on the number of processes, and as the network size increases tenfold, this mismatch means that a round later (200ms) the percentage of processes that have endorsed is slightly higher, but the larger jumps are nonetheless offset by a single round.

The increase in time until first delivery is however due to an entirely different reason: an increase in the number of processes requires a matching increase in the $TTL$, due to the equations specified for EpTO to determine fanout and $TTL$. A tenfold increase in the number of processes leads to, approximately, an increase of 20 in the $TTL$, meaning an additional 4 000ms until delivery, which is what can be observed.

Another effect that can be observed, and that has a very significant impact on the latency of TBO, is the gap between first observation of a given event and delivery of said event. While the gap between first and last observations grew from approximately 0.4s to approximately 1s when going from 100 processes to 100 000 processes, the time until delivery grew from approximately 9s to nearly 22s.

This suggests that the the formula giving the upper bound for the $TTL$ is largely overestimating it, and it could likely be significantly reduced without an impact on correctness, but future work is required to confirm this possibility.

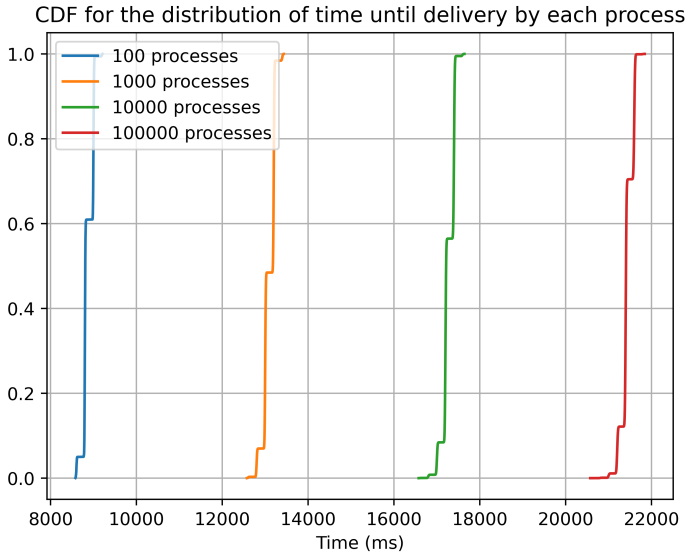



**Figure 4.2:** How the time until the event is delivered changes as the number of processes changes.

Varying the number of endorsers has, comparative to varying the number of processes, a much smaller impact. The average number of messages is largely unchanged, as shown in Table 4.3. A slightly larger percentage of processes both observe (Figure 4.3) and deliver (Figure 4.4) the event earlier, but this is easily explained by the fact that a larger number of endorsers means a larger number of processes is made aware of the event, and begins its dissemination, earlier in the dissemination process.

**Table 4.3:** How the average number of messages changes as the number of endorsers changes.

| Endorsers | Avg. Messages Sent |
|---|---|
| 50 | 1,839.04 |
| 100 | 1,839.75 |
| 150 | 1,841.04 |
| 200 | 1,842.83 |
| 250 | 1,845.16 |

As the percentage of faulty processes changes, the average number of messages is completely unchanged (Table 4.4).

The effects present in the time until first observation (Figure 4.5a) and on the time until delivery (Figure 4.5b) are however noticeable, but not unexpected, as a higher percentage of faulty processes essentially means a larger percentage of messages are dropped, as they are sent to processes that will do nothing with them.

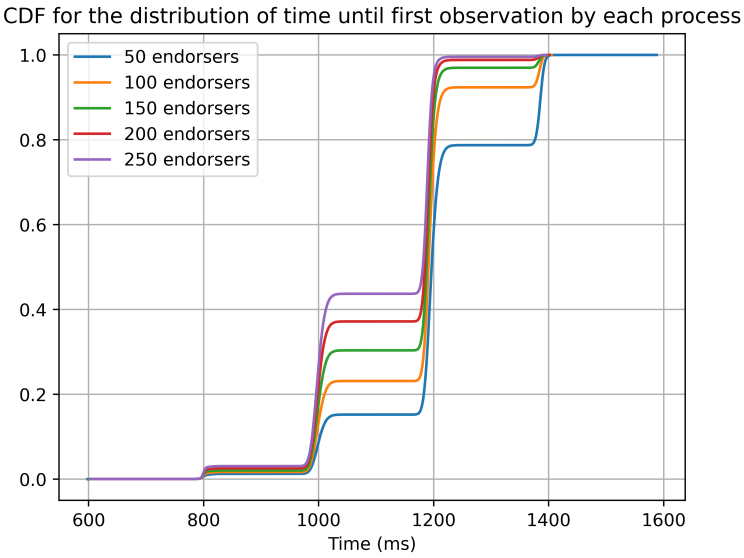



**Figure 4.3:** How the time until the event is first observed changes as the number of endorsers changes.

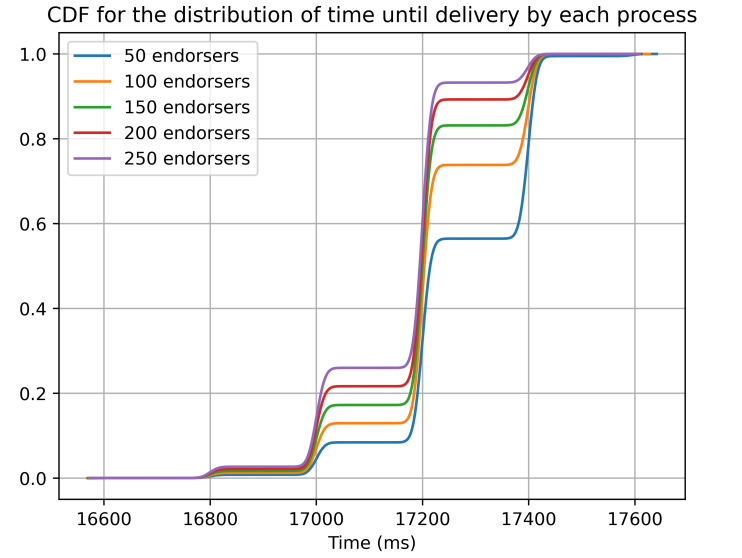
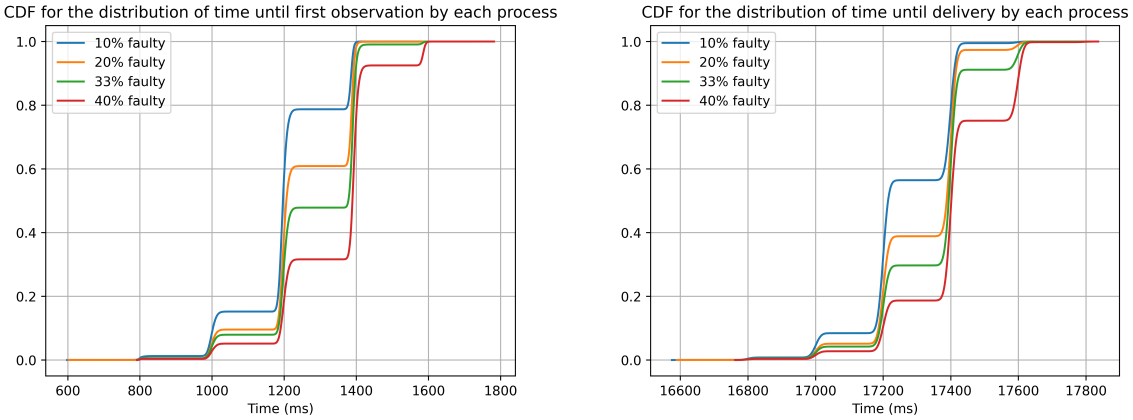



**Figure 4.4:** How the time until the event is delivered changes as the number of endorsers changes.

**Table 4.4:** How the average number of messages changes as the percentage of faulty processes changes.

| Faulty (Pct.) | Avg. Messages Sent |
|---|---|
| 10 | 1,839.04 |
| 20 | 1,838.91 |
| 33 | 1,838.46 |
| 40 | 1,838.13 |




**(a)** Time until the event is first observed.

**(b)** Time until the event is delivered.

**Figure 4.5:** How the time until the event is first observed and delivered change as the percentage of faulty processes varies.

It is therefore no surprise that the effects of varying the packet loss and varying the percentage of faulty processes is largely the same, as shown by Table 4.5, Figure 4.6a, and Figure 4.6b.

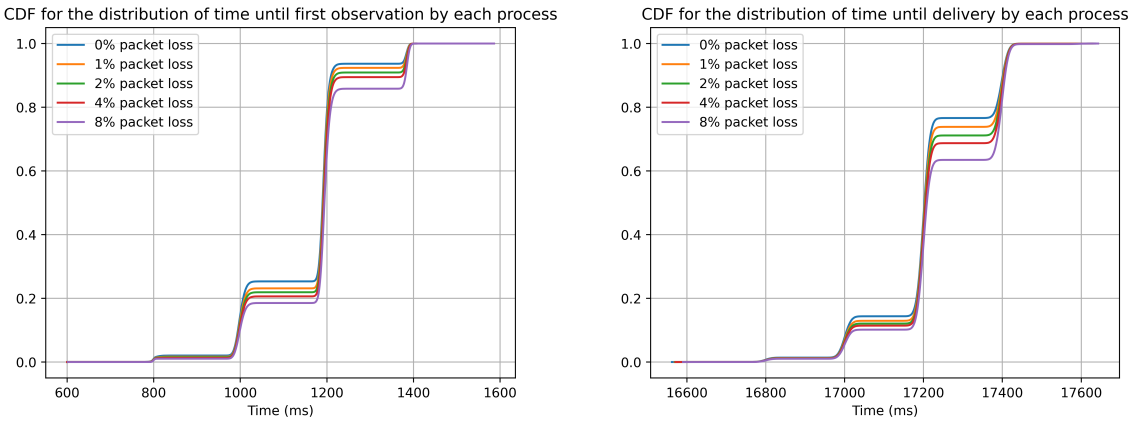
**Table 4.5:** How the average number of messages changes as the packet loss changes.

| Packet Loss (Pct.) | Avg. Messages Sent |
|---|---|
| 0 | 1,839.76 |
| 1 | 1,839.75 |
| 2 | 1,839.74 |
| 4 | 1,839.71 |
| 8 | 1,839.66 |

Last but not least, changes to the endorsement threshold also have no impact on the average number of messages (Table 4.6). They do, however, have an expected impact on both the time until first observation (Figure 4.7a) and in the time until delivery (Figure 4.7b). As the threshold for endorsement is lower, processes begin dissemination slightly earlier, meaning the observations and deliveries take place earlier, but not significantly earlier.

**Table 4.6:** How the average number of messages changes as the endorsement threshold changes.

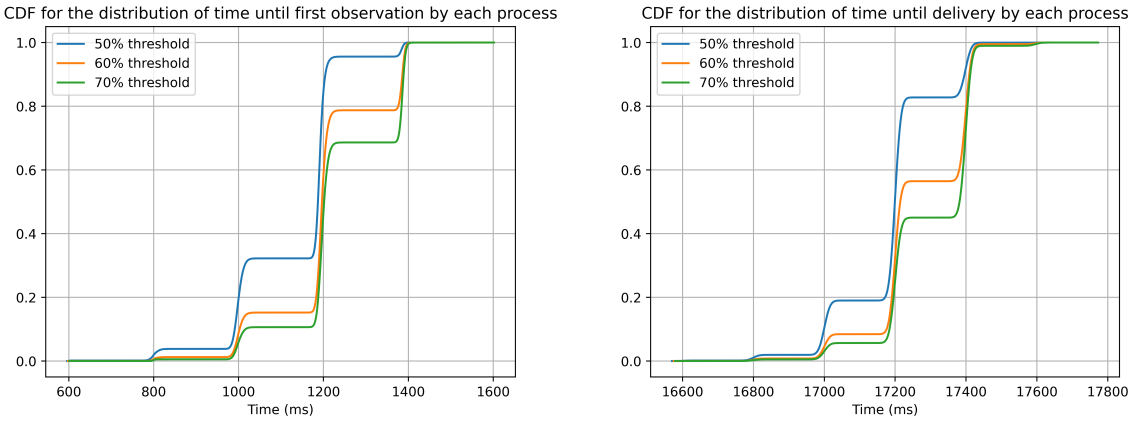
| Endorsement Threshold (Pct) | Avg. Messages Sent |
|---|---|
| 50 | 1,839 |
| 60 | 1,839.04 |
| 70 | 1,839.08 |




**(a)** Time until the event is first observed.

**(b)** Time until the event is delivered.

**Figure 4.6:** How the time until the event is first observed and delivered change as the packet loss varies.




**(a)** Time until the event is first observed.

**(b)** Time until the event is delivered.

**Figure 4.7:** How the time until the event is first observed and delivered change as the endorsement threshold varies.

# 5

# Conclusion

## Contents

The introduction of blockchain technologies, coupled with the surge in interest in them, has lead to significant developments in the field of distributed ledgers. As more and more industries adopt blockchain technologies for mission-critical applications, significant developments were made towards providing common frameworks for blockchain applications.

To aid in the development of an enterprise grade distributed ledger framework, the Hyperledger umbrella project was created by the Linux Foundation. One of the projects under the Hyperledger umbrella, Fabric, is an extensible blockchain system for distributed applications, featuring a modular architecture, including a modular consensus protocol.

While multiple algorithms for total order were studied throughout this document, EpTO, an epidemic total order algorithm, presented the most desirable characteristics. Specifically, it exhibits high scalability in both the number of transactions that can be processed, and the number of nodes that can participate in the protocol. However, EpTO does not withstand the presence of Byzantine faults, operating only under a fail-stop failure model.

For that reason TBO was developed, seeking to maintain all the advantageous qualities of EpTO but adding Byzantine fault tolerance with minimal impact on its other characteristics. While our short theoretical analysis did not support any assertions about the strength of the algorithm, specifically about its endorsement component, our experimental analysis showed very promising results in regards to its scalability, with a logarithmic increase in the number of messages as the number of nodes, in contrast with every other Byzantine Fault Tolerant algorithm discussed which presented a super-linear (often quadratic) increase.

## 5.1 Future Work

In this section we will briefly go over future work that would improve some aspects of TBO, and that are logical steps in which this work can be expanded.

### 5.1.1 Formal proof and other endorsement components

The absence of a formal proof makes immediate real-world use of TBO an ill-advised idea, unless coordinated attackers are of no concern.

Proving the correctness of the dissemination and ordering components should not be significantly difficult, but proving the endorsement component might. The modularity of the endorsement component makes it swappable by other algorithms that can provide the same guarantees it sought to provide, meaning other proven algorithms may be considered as a replacement to overcome this obstacle.

The endorsement component is also not very scalable, requiring a quadratic number of messages scaling with the number of endorsers. This should not pose a significant problem as the number of

endorsers doesn't need to scale significantly, but it can nonetheless be improved upon.

### 5.1.2 Signature overhead

The use of a high number of signatures in the endorsement component, and throughout the dissemination component as means of verification, results in these signatures presenting significant overhead in terms of network traffic and maybe even computational complexity. These impacts were not studied.

Existing work into multi-signatures should allow a significant reduction in the size and computational requirements behind these endorsements, but their impacts and properties were not researched.

Depending on the data contained within the events the overhead of these signatures may be negligible, but it is nonetheless overhead, and any reductions on overhead without a loss in any other aspect (i.e. security) is only beneficial.

The use of multi-signatures is promising, but the reduction in bandwidth overhead needs to be very carefully weighed against any increase in computational complexity, lest bandwidth overhead be replaced with computational overhead.

### 5.1.3 Stricter parameter bounds

The formulas used to calculate the $TTL$ and the fanout, both algorithm parameters, seem to be very lax and provide a very large margin. Calculating stricter bounds for these parameters would enable a significant reduction in the number of messages required.

### 5.1.4 Throughput and steady-state latency

The experimental analysis of TBO focused solely on latency on the delivery of a single event, paying no attention to throughput or how multiple events would behave. We theorize that the throughput of the algorithm should be nearly unchanged by any of the changes to algorithm and network parameters tested, and that multiple events should have no impact as they are largely independent in every aspect other than delivery order, but this would need to be confirmed.

The main concern regarding throughput is whether a large enough number of simultaneous events could introduce additional delays due to conflicting timestamps. These conflicts are expected to, worst case scenario, double the time until delivery, but experimental results would help observe the actual real-world impact of these conflicts.

# Bibliography


[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” *Www.Bitcoin.Org*, p. 9, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] V. Buterin and V. Griffith, “Casper the Friendly Finality Gadget,” pp. 1–10, 2017. [Online]. Available: http://arxiv.org/abs/1710.09437

[3] V. Zamfir, “CASPER The Friendly ...” pp. 1–16, 2018. [Online]. Available: https://github.com/ethereum/research/blob/master/papers/CasperTFG/CasperTFG.pdf

[4] M. Matos, H. Mercier, P. Felber, R. Oliveira, and J. Pereira, “EpTO,” *Proceedings of the 16th Annual Middleware Conference on - Middleware ’15*, pp. 100–111, 2015. [Online]. Available: http://dblp.uni-trier.de/db/conf/middleware/middleware2015.html#MatosMFOP15%0Ahttp://dl.acm.org/citation.cfm?doid=2814576.2814804

[5] M. Castro, M. Castro, B. Liskov, and B. Liskov, “Practical Byzantine fault tolerance,” *OSDI {}99: Proceedings of the third symposium on Operating systems design and implementation*, no. June, 1999.

[6] A. Bessani, J. Sousa, and E. E. Alchieri, “State machine replication for the masses with BFT-SMART,” *Proceedings - 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*, pp. 355–362, 2014.

[7] P. L. Aublin, S. B. Mokhtar, and V. Quema, “RBFT: Redundant byzantine fault tolerance,” *Proceedings - International Conference on Distributed Computing Systems*, pp. 297–306, 2013.

[8] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman, “Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus,” pp. 5–6, 2016. [Online]. Available: http://arxiv.org/abs/1612.02916

[9] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, “DBFT: Efficient Byzantine Consensus with a Weak Coordinator and its Application to Consortium Blockchains,” pp. 1–41, 2017. [Online]. Available: http://arxiv.org/abs/1702.03068

[10] J. Kwon, “TenderMint : Consensus without Mining,” 2014.

[11] E. Buchman, “Tendermint: Byzantine Fault Tolerance in the Age of Blockchains,” 2016. [Online]. Available: http://atrium.lib.uoguelph.ca/xmlui/handle/10214/9769

[12] J. Chen and S. Micali, “Algorand,” 2016. [Online]. Available: http://arxiv.org/abs/1607.01341

[13] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The Honey Badger of BFT Protocols,” *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS’16*, no. Section 3, pp. 31–42, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2976749.2978399

[14] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains,” 2018. [Online]. Available: http://arxiv.org/abs/1801.10228%0Ahttp://dx.doi.org/10.1145/3190508.3190538

[15] K. e. a. Croman, “On Scaling Decentralized Blockchains: (A Position Paper),” *Financial Cryptography and Data Security*, vol. 9604, no. February, pp. 106–125, 2016. [Online]. Available: https://doi.org/10.1007/978-3-662-53357-4

[16] J. R. Douceur, “The Sybil Attack.” Springer Berlin Heidelberg, 2002, pp. 251–260. [Online]. Available: http://link.springer.com/10.1007/3-540-45748-8_24

[17] S. King and S. Nadal, “Peercoin-Paper,” 2012. [Online]. Available: https://peercoin.net/assets/paper/peercoin-paper.pdf

[18] I. Grigg, “EOS - An Introduction,” no. ii, pp. 1–8, 2017.

[19] F. Schuh and D. Larimer, “Bitshares 2.0: General Overview,” *Cryptonomex*, vol. 3268, pp. 0–16, 2017. [Online]. Available: http://aisel.aisnet.org/icis2010_submissions/140

[20] J. Alwen, G. Fuchsbauer, P. Gazi, S. Park, and K. Pietrzak, “Spacecoin : A Cryptocurrency Based on Proofs of Space,” *IACR Cryptology ePrint Archive*, no. December 2017, pp. 1–26, 2015. [Online]. Available: https://pdfs.semanticscholar.org/8fa4/a782ad64bf5228e5c120e7ed00e136b4934d.pdf

[21] R. S. Seán Gauld Franz von Ancoina, “The Burst Dymaxion,” *Cryptorating.Eu*, pp. 1–23, 2017. [Online]. Available: https://cryptorating.eu/whitepapers/Burst/The-Burst-Dymaxion-1.00.pdf

[22] “Living in the EU - EUROPA — European Union.” [Online]. Available: https://europa.eu/european-union/about-eu/figures/living_en

[23] J. Sousa, A. Bessani, and M. Vukolić, "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform," no. 1, pp. 51–58, 2017. [Online]. Available: http://arxiv.org/abs/1709.06921

[24] G. W. Peters and E. Panayi, "Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money," *New Economic Windows*, pp. 239–278, 2016.

[25] "Hyperledger." [Online]. Available: https://www.hyperledger.org/

[26] V. Buterin, "A next-generation smart contract and decentralized application platform," *Etherum*, no. January, pp. 1–36, 2014. [Online]. Available: http://buyxpr.com/build/pdfs/EthereumWhitePaper.pdf

[27] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger Final Draft - Under Review," 2015. [Online]. Available: http://gavwood.com/Paper.pdf?TB_iframe=true&width=288&height=432

[28] A. Back, "Hashcash - A Denial of Service Counter-Measure," *Technical Report*, no. August, pp. 1–10, 2002.

[29] Digiconomist, "Bitcoin Energy Consumption Index - Digiconomist," pp. 1–8, 2018. [Online]. Available: https://digiconomist.net/bitcoin-energy-consumption

[30] "The World Factbook - Country Comparison: Electricity - Consumption," 2018. [Online]. Available: https://www.cia.gov/library/publications/the-world-factbook/rankorder/2233rank.html

[31] "ethereum/solidity: Solidity, the Contract-Oriented Programming Language." [Online]. Available: https://github.com/ethereum/solidity

[32] "ethereum/vyper: Pythonic Smart Contract Language for the EVM." [Online]. Available: https://github.com/ethereum/vyper

[33] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937. [Online]. Available: http://doi.wiley.com/10.1112/plms/s2-42.1.230

[34] "grpc / grpc.io." [Online]. Available: https://grpc.io/

[35] "The Eth2 upgrades — ethereum.org." [Online]. Available: https://ethereum.org/en/eth2/

[36] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato, "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)," RFC Editor, Tech. Rep., 2001. [Online]. Available: https://tools.ietf.org/html/rfc3161

[37] D. G. Thaler and C. V. Ravishankar, “Using name-based mappings to increase hit rates,” *IEEE/ACM Transactions on Networking*, vol. 6, no. 1, pp. 1–14, 1998.

[38] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web,” *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, no. January, pp. 654–663, 1997.

[39] I. Amaral Sequeira, “Large Scale Distributed Algorithms Simulator,” Ph.D. dissertation, 2019.