TBO (Total Byzantine Order): Scalable epidemic probabilistic total order resilient to Byzantine faults

Nuno Anselmo
Instituto Superior Técnico, Universidade de Lisboa

***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

I. Introduction

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 [1]. 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 [2]. 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 [3], 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) [4]–[7], Delegated Proof-of-Stake (DPoS) [8]–[11], Proof-of-Space (PoSpace) [12], and Proof-of-Capacity (PoC) [13].

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 associated 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 [14]. 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 [15], [16] 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.

To aid in the collaborative development of an open source enterprise grade distributed ledger framework, the Linux Foundation created the Hyperledger umbrella project [17], of which the Hyperledger Fabric project . The implementation discussed throughout this work was initially designed to take place on the Hyperledger Fabric project [18], a permissioned blockchain system and one of the many projects under the Hyperledger umbrella.

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 [15]. 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. Initially tailored only to Hyperledger, it is usable in any permissioned network.

The main contribution being made is the development of this new BFT total order algorithm usable on Hyperledger Fabric, TBO, based on EpTO [19], 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.

II. Background

In this chapter we will begin by discussing blockchains, specifically permissioned blockchains, and then smart-contract platforms. Following that, we will discuss EpTO, the most relevant algorithm for Total Order in the context of the work that was done.

A. 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 [3], 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 [20], [21], 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 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.

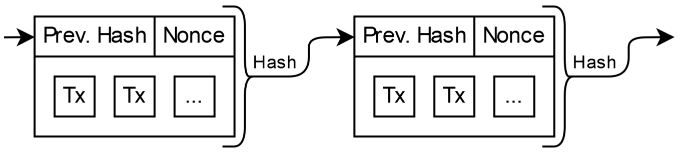


Figure 1: Example of a simple blockchain, with blocks containing only transactions, a nonce, and a hash of the previous block.

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 financial 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.

1) Permissioned Blockchains: In contrast with permissionless blockchains, permissioned blockchains give each node and client[2] 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.

The main 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[3]. 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 [2] as identities need to be authorized in some way before they can be used, and cannot be generated at will by attackers.

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

[2]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

[3]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.

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 [8], [22]–[24]. This poor scalability in number of nodes is, first and foremost, due to an increased number of messages 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%[4] 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.

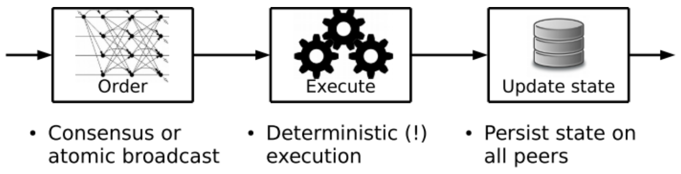
Permissioned blockchains, by nature, require an underlying membership or identity management system to be present, which can be either managed separately from the blockchain or integrated with it, by embedding memberships and permissions in the blockchain. This membership system serves to manage the 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) Blockchains as Smart Contract Platforms: Using blockchains to create cryptocurrencies is the most common model, but it is possible to generalize it, 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.

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; 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.

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


Order
Execute
Update state
• Consensus or atomic broadcast
• Deterministic (!) execution
• Persist state on all peers
Figure 2: Order-execute architecture, as taken from [18].

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.

B. EpTO [19]

EpTO 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). 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 5, operates in a round-based fashion: every δ 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 δ 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 6 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.

III. Algorithm

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 [19], initially introduced in Section II-B, 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.

A. 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.

B. EpTO

EpTO, as presented here, is unchanged from the original [19], which was explained in some depth in Section II-B.

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

1) Dissemination (Algorithm 5): 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 6): 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.

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.

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.

C. 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.

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 5. To mitigate this attack, the aforementioned lines would need to be replaced by the lines described in Algorithm 1.

Algorithm 1: 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
```

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 1 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.

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 6 in conjunction with lines 3-4 of Algorithm 7. To mitigate this attack, the aforementioned lines of Algorithm 6 would need to be amended, to be as the lines presented in Algorithm 2.

Algorithm 2: 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.

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 is possible due to the way we mitigated the vulnerability described in Section III-C1, in Algorithm 1. In order to make this attack impossible, lines 4-5 were added in Algorithm 3.

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 4.

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 III-C1, trying not to stop the dissemination but instead

Algorithm 3: Preventing infinite dissemination

```
1  if $event \notin event\_cutoff$ 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 $event.id \in next\_ball$ then
8          if $next\_ball[event.id].ttl > event.ttl$ then
9              next_ball[$event.id$].ttl ← $event.ttl$
10     else
11         next_ball[$event.id$] ← $event$
```

Algorithm 4: Preventing infinite dissemination

```
1  run task every $\delta$ units of time
2      foreach $event \in next\_ball$ do
3          event.ttl ← $event.ttl + 1$
```

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. 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.

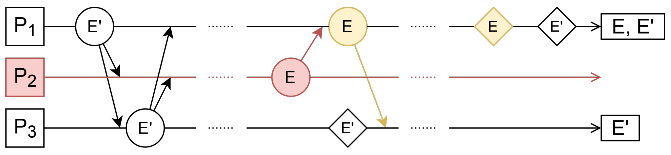


Figure 3: Exemplified attack, with an attacker (P2) using delayed dissemination (circle) of an event (E), leading to incorrect delivery (rhombus) order for processes that only see it after delivery of a future event (E').

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 [25] compliant service, would not be sufficient, as a faulty process could then delay the dissemination of the event with a guaranteed-valid timestamp.

D. TBO

The proposed solution to the problems presented in Section III-C3 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 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.

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 [26] or a special case of rendezvous hashing, consistent hashing [27].

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. There are however 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 III-C3 and shown in Figure 3, 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 threshold of 50% may initially seem like it is optimal, as it is the farthest between the two failure scenarios described. However, due to the attacks described in Section III-E, 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 9, incorporates the changes outlined in Section III-C, 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 10, also incorporates the changes outlined in Section III-C. 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.

E. Known attack vectors

The attacker 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 8), violating its provided guarantees, therefore invalidating the guarantees offered by the dissemination component (Algorithm 9) and subsequently the ones ensured by the ordering component (Algorithm 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[5] (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 when motivating the need for an ordering component.

1) Crafting an attack: Any successful attack will need to be attacking the endorsement component (Algorithm 8). The reason this is true is because the only other component subject to attack is the dissemination component[6] (Algorithm 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.

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[7], 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: noitemsep
   a) Do nothing;
   b) Send out one or more endorsements to one or more correct processes;

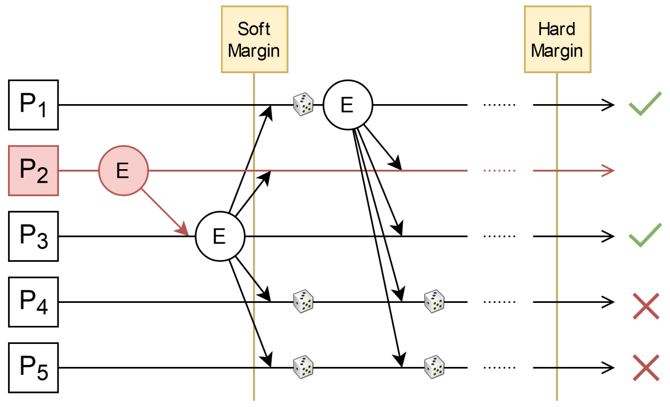


Figure 4: A successful attack with a 3-endorsement threshold where the attacker (B), who is both the creator and an endorser, sends a precisely timed endorsement request (but not endorsement), resulting in the attacker being the only process to have collected 3 endorsements.

One such attack is shown in Figure 4, 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.

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

[6]The ordering component (Algorithm 10) handles no communication, and fully relies on the other components to ensure its correctness, not being subject to direct attacks.

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

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. This analysis is done in Appendix B.

IV. Evaluation

In this chapter we will be going over the main contribution: a BFT Total Order algorithm, based on an existing epidemic algorithm, EpTO [19], introduced in Section II-B. 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.

A. Theoretical evaluation

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

The process through which this was evaluated is borderline *bruteforce*: for a wide range algorithm parameters all possible attacks were calculated. Then, for every possible attack, the analysis in Appendix B was calculated with a 1 000 decimal unit precision, and the probability of that attack succeeding was determined.

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 5 632 067 possible attacks.

The results were, however, not favorable, despite the low number of faulty processes.

In every tested algorithm parameter combination, an attack was always found that had a probability of success greater or equal to 0,1%, at which point we stopped testing further attacks.

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.

B. Experimental evaluation

The experimental evaluation is based on simulating the execution of the TBO algorithm, as presented in Algorithm 8, Algorithm 9, Algorithm 10, and Algorithm 11, and gathering statistics about its runtime operation.

The simulation was performed on Corten [28], an event-based simulator written in Rust. 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, but no correct process was ever made aware of this crash.

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).

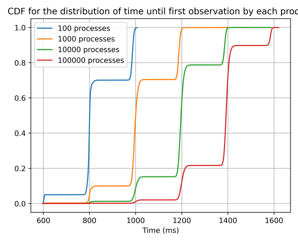
When varying solely the number of processes between 100, 1 000, 10 000, and 100 000, there are significant differences in the average number of messages being sent by each process, as shown in Table I. The number of endorsers is fixed at 50 for this comparison, throughout all sizes.

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.

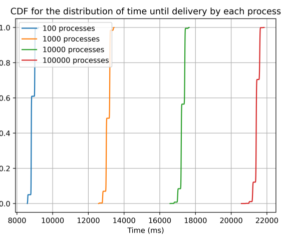
Table I: How the average number of messages changes as the number of processes changes.

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 5a and Figure 5b, presented as cumulative distribution functions, with the percentage of correct processes that have observed 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.



(a) Time until first observation.



(b) Time until delivery.

Figure 5: How the time until the event is first observed and delivered change as the number of processes varies.

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.

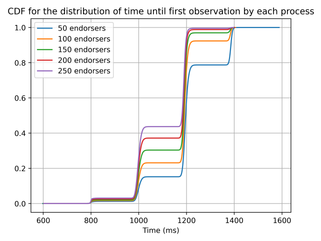
The increase in time until first delivery is however due to an entirely different reason: an increase in the number of processes requires a matching increase in the TTL, due to the equations specified for EpTO to determine fanout and TTL. A tenfold increase in the number of processes leads to, approximately, 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.

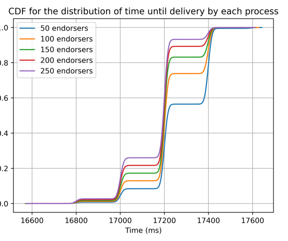
This suggests that the the formula giving the upper bound for the TTL are largely overestimating it, and it could likely be significantly reduced without an impact on correctness, but future work is required to confirm this possibility.

Varying the number of endorsers has a negligible impact on the average number of messages which remains largely unchanged. A slightly larger percentage of processes both observe (Figure 6a) and deliver (Figure 6b) the event earlier, explained by the fact that a larger number of endorsers means a larger number of processes begins dissemination earlier in the process.

As the percentage of faulty processes changes, the average number of messages is completely unchanged. The effects present in the time until first observation (Figure 7a) and on the time until delivery (Figure 7b) are however noticeable,



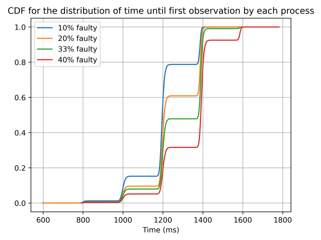
(a) Time until first observation.



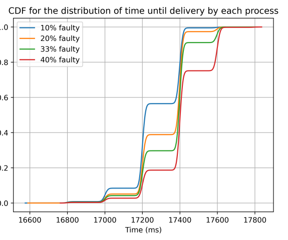
(b) Time until delivery.

Figure 6: How the time until the event is first observed and delivered change as the number of endorsers varies.

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. The effects of varying the packet loss and varying the percentage of faulty processes is largely the same.



(a) Time until first observation.



(b) Time until delivery.

Figure 7: How the time until the event is first observed and delivered change as the percentage of faulty processes varies.

V. Conclusion

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.

EpTO, an epidemic total order algorithm, presents desirable characteristics for use in Fabric, specifically 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. 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.

References

[1] 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

[2] 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

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

[4] J. Kwon, "TenderMint : Consensus without Mining," 2014.

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

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

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

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

[10] I. Grigg, "EOS - An Introduction," no. ii, pp. 1–8, 2017.

[11] 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

[12] 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

[13] 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

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

[15] 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

[16] 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

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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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.

[23] 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.

[24] 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

[25] 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

[26] 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.

[27] 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.

[28] I. Amaral Sequeira, "Large Scale Distributed Algorithms Simulator," Ph.D. dissertation, 2019.

Appendix

Appendix A
Algorithms

Algorithm 5: EpTO Dissemination Component

```
initially
    view ← . . . // system parameter: set of uniformly random correct peers
    K ← . . . // system parameter: fanout
    TTL ← . . . // system parameter: number of rounds
    next_ball ← ∅ // set of events to be relayed in the next round
procedure DISSEMINATE(event)
    event.ttl ← 0
    event.ts ← GETCLOCK()
    next_ball[event.id] ← event
upon receive Ball : ball
    foreach event ∈ ball do
        if event.ttl < TTL then
            if event.id ∈ next_ball then
                if next_ball[event.id].ttl < event.ttl then
                    next_ball[event.id].ttl ← event.ttl
            else
                next_ball[event.id] ← event
        UPDATECLOCK(event.ts)
run task every δ units of time
    foreach event ∈ next_ball do
        event.ttl ← event.ttl + 1
    if #next_ball > 0 then
        peers ← RANDOM(view, K)
        foreach peer ∈ peers do
            send Ball : next_ball to peer
    ORDEREVENTS(next_ball)
    next_ball ← ∅
```

Algorithm 6: EpTO Ordering Component

```
initially
    received ← ∅ // Map of event id → event
    delivered ← ∅ // Set of delivered events
    last_delivered ← 0
procedure ORDEREVENTS(ball)
    foreach event ∈ received do
        received[event.id].ttl ← received[event.id].ttl + 1
    foreach event ∈ ball do
        if event.id ∉ delivered and event.ts ≥ last_delivered then
            if event.id ∈ received then
                if received[event.id].ttl < event.ttl then
                    received[event.id].ttl ← event.ttl
            else
                received[event.id] ← event
    min_queued ← ∞ // Oldest non-deliverable event
    deliverable ← ∅
    foreach event ∈ received do
        if ISDELIVERABLE(event) then
            deliverable ← deliverable ∪ {event} // Preliminarily deliverable
        else if event.ts < min_queued then
            min_queued ← event.ts
    foreach event ∈ deliverable do
        if event.ts ≥ min_queued then
            deliverable ← deliverable \ {event}
        else
            received ← received \ {event} // Will be delivered
    foreach event ∈ SORT(deliverable) do
        delivered ← delivered ∪ {event}
        last_delivered ← event.ts
        EPTO-DELIVER(event)
```

Algorithm 7: EpTO Utilities

```
initially
    clock ← 0
procedure ISDELIVERABLE(event)
    return event.ttl > TTL
procedure GETCLOCK()
    clock ← clock + 1
    return clock
procedure UPDATECLOCK(timestamp)
    if timestamp > clock then
        clock ← timestamp
```

Algorithm 8: TBO Endorsement Component

```
initially
    endorsers ← ... // parameter: number of endorsers
    minimum_endorsements ← ... // parameter: number of endorsements required before acceptance
    hard_timestamp_margin ← ... // parameter: hard margin, always rejected if crossed
    soft_timestamp_margin ← ... // parameter: soft margin, randomly rejected if crossed
    soft_accept_probability ← ... // parameter: probability to reject once soft margin is crossed
    received_endorsements ← ∅ // map of events → received endorsements
procedure TBO-Broadcast(event)
    event.ts ← GetClock()
    foreach position, endorser ∈ GetEndorsers(event) do
        send EndorsementRequest : event, position to endorser
upon receive EndorsementRequest : event, position
    if position ∉ received_endorsements and GetEndorsers(event)[position] = self then
        if event.ts > GetClock() - hard_timestamp_margin and event.ts < GetClock() + hard_timestamp_margin then
            if event.ts > GetClock() - soft_timestamp_margin or GetRandom(0, 1) < soft_accept_probability then
                endorsement ← CreateEndorsement(event, position)
                received_endorsements[event][position] ← endorsement
                foreach pos, endorser ∈ GetEndorsers(event) do
                    send Endorsement : event, pos, received_endorsements[event] to endorser
upon receive Endorsement : event, position, endorsements
    send EndorsementRequest : event, position to self
    foreach endorsement ∈ endorsements do
        event ← endorsement.event
        position ← endorsement.position
        if ValidateEndorsements([endorsement]) then
            received_endorsements[event][position] ← endorsement
    if #received_endorsements[event] ≥ minimum_endorsements then
        event.endorsements ← received_endorsements[event]
        Disseminate(event)
```

Algorithm 9: TBO Dissemination Component

```
initially
    view ← ... // system parameter: set of uniformly random peers
    K ← ... // system parameter: fanout
    TTL ← ... // system parameter: number of rounds
    next_ball ← ∅ // set of events to be relayed in the next round
    event_cutoff ← ∅ // map of events → minimum TTL expected
procedure Disseminate(event)
    event.ttl ← GetClock()
    next_ball[event.id] ← event
upon receive Ball : ball
    foreach event ∈ ball do
        if #event.endorsements < minimum_endorsements or not ValidateEndorsements(event.endorsements) then
            continue// skip event
        if event ∉ event_cutoff then
            event_cutoff[event] ← 1
            event.ttl ← 1
        if event.ttl < event_cutoff[event] then
            event.ttl ← event_cutoff[event]
        if event.ttl < TTL then
            if event.id ∈ next_ball then
                if next_ball[event.id].ttl > event.ttl then
                    next_ball[event.id].ttl ← event.ttl
            else
                next_ball[event.id] ← event
        UpdateClock(event.ts)
run task every δ units of time
    foreach event ∈ next_ball do
        event.ttl ← event.ttl + 1
    foreach event ∈ event_cutoff do
        event_cutoff[event] ← event_cutoff[event] + 1
    if #next_ball > 0 then
        peers ← Random(view, K)
        foreach peer ∈ peers do
            send Ball : next_ball to peer
    OrderEvents(next_ball)
    next_ball ← ∅
```

Algorithm 10: TBO Ordering Component

```
initially
    received ← ∅ // Map of event id → event
    delivered ← ∅ // Set of delivered events
    last_delivered ← 0
procedure OrderEvents(ball)
    foreach event ∈ received do
        received[event.id].age ← received[event.id].age + 1
    foreach event ∈ ball do
        if event.id ∉ delivered and event.id ∉ received then
            if event.ts ≥ last_delivered then
                event.age ← 1
                received[event.id] ← event
    min_queued ← ∞ // Oldest non-deliverable event
    deliverable ← ∅
    foreach event ∈ received do
        if IsDeliverable(event) then
            deliverable ← deliverable ∪ {event} //
              Preliminarily deliverable
        else if event.ts < min_queued then
            min_queued ← event.ts
    foreach event ∈ deliverable do
        if event.ts ≥ min_queued then
            deliverable ← deliverable \ {event}
        else
            received ← received \ {event} // Will be
              delivered
    foreach event ∈ Sort(deliverable) do
        delivered ← delivered ∪ {event}
        last_delivered ← event.ts
        TBO-Deliver(event)
```

Algorithm 11: TBO Utilities

```
initially
    clock ← 0
procedure IsDeliverable(event)
    return event.age > TTL
procedure GetClock()
    return clock
procedure UpdateClock(timestamp)
    if timestamp > clock then
        clock ← timestamp
```

Appendix B
Analysis

The purpose of this appendix is to understand how the system evolves as an attack as described in Section III-E1 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 appendix:

noitemsep

- $N \leftarrow$ 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

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:

noitemsep

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

$$\begin{aligned} P(E_N) &= P(E_N|NM) * P(NM) \\ &= P(NM) \end{aligned} \tag{1}$$

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:

$$P(E_N|\neg NM)_s = 1 - P(\neg E_1|\neg NM)^{(\sum_{k=0}^{S} \#C_k * P(E_N)_k) + \#f_E} \tag{2}$$

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:

$$P(E_N|SM) = 1 - P(\neg E_1|\neg NM)^n \tag{3}$$

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:

$$\begin{aligned} \#f_E &= \#f_E + n \\ \#f_U &= \#f_U - n \end{aligned} \tag{4}$$

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:

$$\begin{aligned} P(E_N|\neg NM)_s = P(E_N|\neg NM)_s + \\ + P(E_N|SM) * P(SM|\neg E_N) * P(\neg E_N|\neg NM)_s \end{aligned} \tag{5}$$

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:

$$\begin{aligned} P(E_N)_s = P(E_N|NM)_s * P(NM)+ \\ +P(E_N|\neg NM)_s * P(\neg NM) \\ = P(NM) + P(E_N|\neg NM)_s * P(\neg NM) \end{aligned} \tag{6}$$

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 (2) to (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:

$$P(E_N) = \sum_{s=0}^{S} P(E_N)_s * (\#C_s/\#C) \tag{7}$$

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:

$$\#E_{min} = max(\#T - \#f_U, 0) \tag{8}$$

And the maximum number is:

$$\#E_{max} = min(\#T - \#f_E - 1, \#C) \tag{9}$$

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:

$$P(A) = \sum_{k=\#E_{min}}^{\#E_{max}} \binom{\#C}{k} * P(E_N)^k * (1 - P(E_N))^{\#C-k} \tag{10}$$