# Brief Announcement: Fast TetraBFT – Optimizing Latency Where It Matters

**Antonio J. Fernández-Pinto**
IMDEA Software Institute, Madrid, Spain
Universidad Politécnica de Madrid, Spain

**Manuel Bravo**
Informal Systems, Madrid, Spain

**Gregory Chockler**
University of Surrey, Guildford, UK

**Alexey Gotsman**
IMDEA Software Institute, Madrid, Spain

## Abstract

Unauthenticated Byzantine consensus protocols achieve optimal failure resilience while relying only on authenticated point-to-point channels, not authenticated messages. They are an attractive building block for blockchains without global trust assumptions as well as for future post-quantum settings. We consider unauthenticated Byzantine consensus in the partial synchrony model and focus on optimizing its *good-case latency* – the worst-case time for correct processes to reach a decision under favorable conditions. A recently proposed ForgetIT protocol achieves an optimal good-case latency of 3 message delays but employs a highly complex design. We show that this complexity is unnecessary. To this end, we present Fast TetraBFT – an unauthenticated Byzantine consensus protocol that achieves optimal good-case latency by augmenting an existing TetraBFT protocol with a simple fast-path wrapper. Our solution lowers the good-case latency of TetraBFT from 5 to 3 message delays while preserving its bounded space requirements and low communication complexity.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models

**Keywords and phrases** Consensus, partial synchrony, Byzantine fault tolerance

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2026.56

**Related Version** *Extended Version*: `https://arxiv.org/abs/2606.03754` [11]

**Funding** This work was partially supported by the projects SACO and María de Maeztu, funded by MICIU/AEI, and REDONDA, funded by the EU CHIST-ERA network.

## 1 Introduction

We consider Byzantine consensus in the *partial synchrony* model [10]: the system is guaranteed to eventually reach a *Global Stabilization Time* (GST) after which message delays are bounded by an unknown constant $\delta$ and process clocks track real time. Motivated by the blockchain use case, in recent years researchers have developed a plethora of Byzantine consensus protocols in this model. Most of these are *authenticated*, i.e., they send messages authenticated using cryptographic signatures. These allow a process $p_1$ to prove to a process $p_2$ that a third process $p_3$ has sent a given message. In contrast, *unauthenticated* (aka *information-theoretic*) protocols rely only on authenticated point-to-point channels and thus disallow the above pattern. The minimal assumptions of such protocols make them applicable in a broader range of settings. These include post-quantum or computationally constrained environments, as well as blockchains without global trust assumptions, such as Stellar [13] and Ripple [5].


© Antonio J. Fernández-Pinto, Manuel Bravo, Gregory Chockler, and Alexey Gotsman;
licensed under Creative Commons License CC-BY 4.0



Since realistic networks behave synchronously most of the time, the efficiency of partially synchronous blockchain implementations is largely determined by their performance under stable network conditions. In more detail, blockchains often run a sequence of Byzantine consensus instances, each deciding on the next block in the chain. A consensus instance proceeds through a sequence of *views* $0, 1, 2, \ldots$, each with a designated leader process responsible for driving the protocol towards a decision. Since in practice the leaders are selected to maximize the chances of correct behavior (e.g., by penalizing misbehavior via stake slashing), under stable network conditions decisions are usually reached in view 0. For example, data from the Cosmos Hub blockchain [1] running the Tendermint protocol [7] shows that 99.995% of the instances were decided in view 0 over a seven-day sample period. Thus, a performance-critical metric for Byzantine consensus protocols is their *good-case latency* – the worst-case latency to decision if the consensus instance is started during the synchronous period (after GST) and the initial leader is correct.

In this paper we consider the problem of optimizing the good-case latency of unauthenticated Byzantine consensus. Assuming the optimal resilience – $n \geq 3f + 1$ processes to tolerate $f$ Byzantine failures – the lower bound on the good-case latency is $3\delta$ [3]. The first protocol to match this bound, a signature-free version of PBFT [8], was able to achieve this only at the expense of a high message complexity – $O(n^3)$ per view. Information-theoretic HotStuff [4] and TetraBFT [15] then lowered the communication complexity to $O(n^2)$ per view, but only at the expense of a higher good-case latency – $6\delta$ and $5\delta$, respectively. A very recent proposal, Forget-IT [2], achieved both the optimal good-case latency of $3\delta$ and the communication complexity of $O(n^2)$ per view. Unfortunately, the protocol has a complicated design, in part because it also minimizes another latency metric, which its authors call *robust good-case latency*. This is the worst-case latency of a decision measured from the first time after GST when a correct leader enters a view (whether 0 or higher).

The rationale for the above metric was to capture latency guarantees under adversarial conditions, when the protocol is started before GST. We argue that the metric does not fulfil this function. In practice, what matters under adversarial conditions is how quickly the protocol recovers after the onset of the synchronous period, i.e., the time to decision measured from GST. To take a decision in this case, processes need to first exit the views they were in before GST and enter a "good" view with a correct leader. Processes exit a view when their timers expire, and the duration of these timers is typically orders of magnitude larger than $\delta$. Thus, optimizing the latency to the decision after entering a good view is irrelevant, since the time to decision after GST will be dominated by timer durations.

We set out to find a simpler protocol that achieves the optimal good-case latency and low communication complexity. Our answer is *Fast TetraBFT* – a protocol with the good-case latency of $3\delta$ and the communication complexity of $O(n^2)$ per view. The protocol guarantees optimal resilience and optimistic responsiveness [14], and can be implemented in bounded space [11, §A]. It is a wrapper around TetraBFT [15] that uses a special code path for view 0 and executes a slightly modified TetraBFT in higher views. Thus, our proposal improves the good-case latency of TetraBFT while preserving its performance under adversarial conditions.

## 2 System Model

We consider a distributed message-passing system of $n \geq 3f + 1$ processes, of which at most $f$ can be Byzantine. The system is partially synchronous [9, 10]: for every execution, there exist a *Global Stabilization Time* (GST) and a duration $\delta$ such that, after GST, message delays between every pair of correct processes are bounded by $\delta$; before GST, messages can be

```
   // The following handlers are executed only if
      started_tetra = true
 1 function init_tetra(x)
 2 |  val ← x
 3 |  Start the view synchronizer
 4 upon new_view(v)
 5 |  view ← v
 6 |  voted ← false
 7 |  send SUGGEST(view, V2, prev_V2, V3)
   |    to leader(view)
 8 |  send PROOF(view, V1, prev_V1, V4) to all
 9 when received f + 1 VOTE2(_, x_j),
     directly or within SUGGEST messages
10 |  pre: lock ≠ ⊥ ∧ ∀j. lock ≠ x_j
11 |  lock ← ⊥
12 when received n − f SUGGEST(view, _, _, _)
13 |  pre: p_i = leader(view) ∧ ∃x. valid(x) ∧
   |       safe_val_leader(x) ∧ (x = val ∨
   |       x is in one of the SUGGEST messages)
14 |  send PROPOSE(view, x) to all
15 when received PROPOSE(view, x) from
     leader(view) and n − f PROOF(view, _, _, _)
16 |  pre: safe_val_follower(x) ∧
   |       valid(x) ∧ ¬voted
17 |  voted ← true
18 |  if the value in V1 ≠ x then prev_V1 ← V1
19 |  V1 ← VOTE1(view, x)
20 |  send V1 to all
21 when received n − f VOTE1(view, x)
22 |  if the value in V2 ≠ x then prev_V2 ← V2
23 |  V2 ← VOTE2(view, x)
24 |  send V2 to all
25 when received n − f VOTE2(view, x)
26 |  V3 ← VOTE3(view, x)
27 |  send V3 to all
28 when received n − f VOTE3(view, x)
29 |  V4 ← VOTE4(view, x)
30 |  send V4 to all
31 when received n − f VOTE4(view, x)
32 |  decide x
```

**Figure 1** TetraBFT at process $p_i$. Initially, V1, V2, V3, V4, prev_V1, prev_V2 are $\perp$.

arbitrarily delayed or lost. We also assume a known upper bound $\Delta$ on the maximum value of $\delta$ across all executions, i.e., $\delta < \Delta$ [12]. Processes are equipped with hardware clocks that can drift arbitrarily from real time before GST, but do not drift thereafter. Communication channels are *authenticated*: if a correct process $p_j$ receives a message $m$ claiming to have been sent by a process $p_i$, and $p_i$ is correct, then $m$ was previously sent by $p_i$ to $p_j$.

We assume a predicate valid() over consensus proposals. In Byzantine consensus, each correct process proposes a valid value and then has to decide on a value so that: (Validity) a correct process decides on a valid value; (Agreement) no two correct processes decide on different values; and (Liveness) every correct process eventually decides on a value.

## 3 TetraBFT Overview

We build on the TetraBFT protocol [15], whose pseudocode is given in Figures 1–2 (the code highlighted in red is necessary for Fast TetraBFT and should be ignored for now). The protocol proceeds in views, and each view $v$ has a designated leader leader($v$) whose role rotates round robin among all processes. We number TetraBFT views starting from 1, since we use view 0 for Fast TetraBFT. Processes enter a view $v$ upon an upcall `new_view`($v$) from a *view synchronizer* (line 4), whose implementation we omit. For concreteness, we could use the FastSync synchronizer of Bravo et al. [6], which does not rely on authenticated messages and requires only bounded space. The synchronizer ensures that for some view $\mathcal{V}$ entered after GST, all correct processes enter each view $v \geq \mathcal{V}$ and overlap in $v$ for a given duration (TetraBFT requires $6\delta$). The structure of each view in TetraBFT is as follows:

- Upon entering a new view, processes send a SUGGEST message to its leader with information to help it make a proposal (line 7). They also broadcast PROOF messages to help other processes check that the leader's proposal is correct (line 8).
- The leader determines its proposal based on a *quorum* of $n-f$ SUGGEST messages satisfying `safe_val_leader`, and broadcasts the proposal in a PROPOSE message (line 14).

```
33 function safe_suggest(x, v, SUGGEST(_, VOTE2(v2, x2), VOTE2(v2', x2'), _))
34     return v = 1 ∨ (v2 ≥ v ∧ x2 = x) ∨ (v2' ≥ v ∧ x2' ≠ x2)
35 function safe_proof(x, v, PROOF(_, VOTE1(v1, x1), VOTE1(v1', x1'), _))
36     return v = 1 ∨ (v1 ≥ v ∧ x1 = x) ∨ (v1' ≥ v ∧ x1' ≠ x1)
37 function blocking_safe_suggests(x, v)
38     return ∃B. |B| ≥ n − 2f ∧ ∀p ∈ B. received a SUGGEST message m from p s.t. safe_suggest(x, v, m)
39 function blocking_safe_proofs(x, v)
40     return ∃B. |B| ≥ n − 2f ∧ ∀p ∈ B. received a PROOF message m from p s.t. safe_proof(x, v, m)
41 function safe_val_leader(x)
42     if lock ≠ ⊥ ∧ lock ≠ x then return false
43     if view = 1 then return true
44     Examine the information about VOTE3 received in this view's SUGGEST messages:
45     if n − f processes have not sent any VOTE3 before view then return true
46     return ∃Q, v'. |Q| ≥ n − f ∧ v' < view ∧ blocking_safe_suggests(x, v')  ∧
               (no process in Q has sent a VOTE3 after v' or a VOTE3(v', x') for any x' ≠ x)
47 function safe_val_follower(x)
48     if lock ≠ ⊥ ∧ lock ≠ x then return false
49     if view = 1 then return true
50     Examine the information about VOTE4 received in this view's PROOF messages:
51     if n − f processes have not sent any VOTE4 before view then return true
52     return ∃Q, v'. |Q| ≥ n − f ∧ v' < view  ∧
               (no process in Q has sent a VOTE4 after v' or a VOTE4(v', x') for any x' ≠ x)  ∧
               (blocking_safe_proofs(x, v') ∨ (∃x1, x2, v1, v2. x1 ≠ x2 ∧ v' ≤ v1 < v2 < view  ∧
               blocking_safe_proofs(x1, v1) ∧ blocking_safe_proofs(x2, v2)))
```

**Figure 2** TetraBFT: auxiliary functions.

- A process accepts the leader's proposal if it receives a quorum of PROOF messages satisfying `safe_val_follower`. A process accepts only one proposal per view (cf. the voted flag).
- Processes go through four rounds of voting on the proposal, sending messages VOTE1 to VOTE4 (line 20–30). Votes in each succeeding round are sent in response to a quorum of matching votes from the previous round. When a process receives a quorum of matching VOTE4 messages, it decides (line 32).
- A process stores the last votes it issued in variables V1 to V4. For VOTE1 and VOTE2 it also keeps track of the last two votes for distinct values using prev_V1 and prev_V2 (lines 18 and 22). These votes are then reported in SUGGEST and PROOF messages (lines 7–8).

## 4 Fast TetraBFT

Fast TetraBFT runs a special protocol in view 0 (Figure 3), which is driven by an *initial leader*. When this leader is correct and the network is synchronous, the protocol decides within $3\delta$. Hence, at the beginning of the protocol each process sets a timer for a conservative bound of $3\Delta$ (line 57). When the timer expires (line 60), the process switches to TetraBFT, disabling the actions in Figure 3 and enabling those in Figure 1 via a flag started_tetra.

The initial leader starts Fast TetraBFT by broadcasting its consensus proposal in a FAST_PROPOSE message (line 59). Each process accepts only one such proposal (cf. the voted flag), which it broadcasts in a VOTE0 message (line 67). A process receiving a quorum of VOTE0s for the same value broadcasts it in a COMMIT message (line 71), and a process receiving a quorum of matching COMMITs decides (line 73). This ensures the good-case latency of $3\delta$.

A process sending a COMMIT message for a value $x$ agrees to $x$ being decided, and thus becomes *locked* on this value. This is recorded by storing $x$ in val and lock (line 70). The former variable determines the proposal fed into TetraBFT (line 63), and is initialized to the

```
53 function consensus(x)
54     val ← x
55     lock ← ⊥
56     started_tetra ← voted ← false
57     Set a timer for 3Δ
58     if p_i is the initial leader then
59         send FAST_PROPOSE(x) to all
60 when the timer expires
61     pre: ¬started_tetra
62     started_tetra ← true
63     init_tetra(val)
64 when received FAST_PROPOSE(x) from
     the initial leader
65     pre: valid(x) ∧ ¬voted ∧ ¬started_tetra
66     voted ← true
67     send VOTE0(x) to all
68 when received n − f VOTE0(x)
69     pre: ¬started_tetra
70     lock ← val ← x
71     send COMMIT(x) to all
72 when received n − f COMMIT(x)
73     decide x
```

**Figure 3** Fast TetraBFT code for view 0 at process $p_i$.

process's own proposal (line 54). The latter constrains the proposals that can be made by leaders and accepted by followers in TetraBFT: when lock is set to $x$, any proposals different from $x$ are rejected (lines 42 and 48). Since view 0 may fail to reach a decision, a process also needs to be able to *unlock* the value $x$ it recorded, to allow TetraBFT to decide. Our main insight is that the process can do this upon receiving $f + 1$ VOTE2 messages for values different from $x$, either directly or within SUGGEST messages (line 9).

## 5 Correctness

Validity holds trivially, so we now prove Agreement. The safety of TetraBFT rules out contradictory decisions in views $> 0$ [11, §B]. Lemmas 1-2 below prove agreement between values decided in view 0. Finally, Lemmas 3-4 prove agreement between the values decided in view 0 (line 73) and those decided in the subsequent views (line 32).

▶ **Lemma 1.** *If two correct processes lock values $x$ and $x'$, respectively, then $x = x'$.*

**Proof.** A correct process locks a value only after receiving a quorum of $n - f$ VOTE0 messages for it (line 68). Since any two quorums intersect in a correct process, locking different values $x$ and $x'$ requires some correct process to send both VOTE0$(x)$ and VOTE0$(x')$. This cannot happen since a process only sends a VOTE0 once (the voted flag at line 65). ◀

▶ **Lemma 2.** *Any two correct processes that decide in view 0 must decide the same value.*

**Proof.** A correct process decides a value $x$ in view 0 after receiving COMMIT$(x)$ from a quorum of $n - f$ processes (line 72). Out of these processes, at least one must be correct, and this process locks $x$ before sending its message. The required then follows from Lemma 1. ◀

▶ **Lemma 3.** *If $x$ is locked by $f + 1$ correct processes, then no correct process can unlock $x$.*

**Proof.** Assume toward a contradiction that some correct process unlocks $x$. Let $t$ be the earliest time when this happens, and let $p_i$ be the unlocking process. Then by the time $t$, $p_i$ has received VOTE2$(_, x_j)$ from $f + 1$ processes $p_j$ such that every $x_j \neq x$ (line 9). Since at least one of these processes is correct, it must have sent its VOTE2 following the receipt of $n - f$ matching VOTE1s for some value $x' \neq x$ (line 21). We know that $f + 1$ correct processes lock $x$, so there exists a correct process $p$ which both locks $x$ at some time $t_l$ and broadcasts VOTE1$(_, x')$ at some time $t' < t$. Then safe_val_follower$(x', v')$ holds at $p$ at time $t'$. Since $x$ is the only value that $p$ can ever lock (Lemma 1), line 48 implies that $p$ has

$\mathsf{lock} = \bot$ at time $t'$. Furthermore, $\mathsf{started_tetra} =$ **false** at $p$ at $t_l$ (line 69) and must be **true** for $p$ to send `VOTE1` at $t'$. Then $t_l < t'$, so $p$ must unlock $x$ at some point between $t_l$ and $t'$. Since $t' < t$, this contradicts our assumption that no correct process does so before $t$. ◀

▶ **Lemma 4.** *If two correct processes decide $x$ in view 0 and $x'$ in a view $> 0$, then $x = x'$.*

**Proof.** A correct process decides $x$ in view 0 after receiving a quorum of $n - f$ `COMMIT`$(x)$ messages (line 72). Out of these, at least $n - 2f$ must be sent by correct processes, which have thus locked $x$ (line 70). By Lemma 3, these processes will never unlock $x$. Then `safe_val_follower` ensures that they will never send a `VOTE1` message for another value $x' \neq x$ in any view $> 0$ (lines 48 and 16). This makes it impossible to assemble a quorum of $n - f$ `VOTE1` messages for $x'$ needed to decide this value in a view $> 0$. ◀

▶ **Theorem 5** (Liveness)**.** *Every correct process eventually decides.*

**Proof.** Each correct process eventually times out and starts executing TetraBFT (line 63). If all correct processes eventually have $\mathsf{lock} = \bot$, then they end up executing unmodified TetraBFT, which we know is live (we formalize this in [11, §B]). Thus, we only have to consider the case when a non-empty set of correct processes $L$ get locked on a value and never unlock. By Lemma 1, they must all be locked on the same value $l$. By the properties of the view synchronizer (§ 3), there exists a view $v \geq \mathcal{V}$ such that: the leader of $v$ is a correct process $p \in L$; and all correct processes enter $v$ after $\mathsf{GST}$ and overlap in $v$ for at least $6\delta$. We now show that: *(i)* $p$ must propose $l$; and *(ii)* all correct processes accept this proposal. Then the interval of $6\delta$ during which the correct processes overlap in $v$ is enough for them to exchange the messages leading to a decision, as required.

To prove *(i)*, note that, by line 42, $p$ cannot propose any value other than $l$. We now prove that $p$ will send `PROPOSE`$(v, l)$. Assume this is not the case. Then by line 13, `safe_val_leader`$(l)$ returns **false**. Process $p$ will eventually receive `SUGGEST` messages for view $v$ from a quorum $Q$ of $n - f$ correct processes. By line 45, one of the processes in $Q$ must report in its `SUGGEST` message that it has previously sent a `VOTE3`: otherwise `safe_val_leader`$(l)$ would return **true**. Let $v' < v$ be the highest view where a process from $Q$ sent `VOTE3`, and $x$ be the value in this `VOTE3`. We now consider two cases:

- $x \neq l$. Correct processes include the two latest `VOTE2`s for distinct values into their `SUGGEST` messages (line 7). Since a correct process sent `VOTE3`$(v', x)$, $n - 2f$ correct processes sent `VOTE2`$(v', x)$. Then the `SUGGEST` messages these processes send to $p$ in view $v$ must include `VOTE2` for a value different from $l$. This causes $p$ to unlock (line 11), contradicting the fact that $p \in L$.
- $x = l$. Since a correct process sent `VOTE3`$(v', l)$, $n - 2f$ correct processes sent `VOTE2`$(v', l)$. Consider one of these processes, and let $m$ be the `SUGGEST` message it sends in view $v$. If between views $v'$ and $v$, it did not send `VOTE2` for a value $x \neq l$, then `safe_suggest`$(l, v', m)$ holds by the second disjunct in line 34. Otherwise, the predicate holds by the third disjunct. Thus, `safe_suggest`$(l, v', m)$ holds for $n - 2f$ `SUGGEST` messages $m$ sent by correct processes in $v$. Once $p$ receives these, `blocking_safe_suggest`$(l, v')$ holds. This validates the condition at line 46, so that `safe_val_leader`$(l)$ returns **true**: a contradiction.

The proof of *(ii)* is analogous. Assume toward a contradiction that some correct process $q$ does not accept $p$'s proposal. Then by line 16, `safe_val_follower`$(l)$ returns **false**. By Lemma 1, this cannot be due to line 48. Process $q$ will eventually receive `PROOF` messages from a quorum $Q$ of correct processes. By line 51, one of the processes in $Q$ must report in its `PROOF` message that it had previously sent a `VOTE4`. Let $v' < v$ be the highest view where a process from $Q$ sent `VOTE4`, and $x$ be the value in this `VOTE4`. We now consider two cases:

- $x \neq l$. Correct processes include the two latest `VOTE2`s for distinct values into their `SUGGEST` messages (line 7). Since a correct process sent `VOTE4`$(v', x)$, $n - 2f$ correct processes sent `VOTE2`$(v', x)$. Then the `SUGGEST` messages these processes send to $p$ in view $v$ must include `VOTE2` for a value different from $l$. This causes $p$ to unlock (line 11), contradicting the fact that $p \in L$.
- $x = l$. Since a correct process sent `VOTE4`$(v', l)$, $n - 2f$ correct processes sent `VOTE1`$(v', l)$. Consider one of these processes, and let $m$ be the `PROOF` message it sends in view $v$. If between views $v'$ and $v$, it did not send `VOTE1` for a value $x \neq l$, then `safe_proof`$(l, v', m)$ holds by the second disjunct in line 36. Otherwise, the predicate holds by the third disjunct. Thus, `safe_proof`$(l, v', m)$ holds for $n - 2f$ `PROOF` messages $m$ sent by correct processes in $v$. Once $q$ receives these, `blocking_safe_proofs`$(l, v')$ holds. This validates the condition at line 52, so that `safe_val_leader`$(l)$ returns **true**: a contradiction. ◀

## References


1 `https://cosmos.network/`.

2 Ittai Abraham, Sourav Das, Yuval Efron, and Jovan Komatovic. Forget-IT: Optimal good-case latency for information-theoretic BFT. In *Symposium on Principles of Distributed Computing (PODC)*, 2026.

3 Ittai Abraham, Ling Ren, and Zhuolun Xiang. Good-case and bad-case latency of unauthenticated Byzantine broadcast: A complete categorization. In *Conference on Principles of Distributed Systems (OPODIS)*, 2021. `doi:10.4230/LIPIcs.OPODIS.2021.5`.

4 Ittai Abraham and Gilad Stern. Information theoretic HotStuff. In *Conference on Principles of Distributed Systems (OPODIS)*, 2020. `doi:10.4230/LIPIcs.OPODIS.2020.11`.

5 Ignacio Amores-Sesar, Christian Cachin, and Jovana Mićić. Security analysis of Ripple consensus. In *Conference on Principles of Distributed Systems (OPODIS)*, 2021. `doi:10.4230/LIPIcs.OPODIS.2020.10`.

6 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine consensus live. *Distributed Computing*, 35(6):503–532, 2022. `doi:10.1007/S00446-022-00432-Y`.

7 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *arXiv*, 2018. `arXiv:1807.04938`.

8 Miguel Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, Massachusetts Institute of Technology, 2001.

9 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. `doi:10.1145/226643.226647`.

10 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. `doi:10.1145/42282.42283`.

11 Antonio J. Fernández-Pinto, Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Fast TetraBFT: Optimizing latency where it matters. *arXiv*, 2026. `doi:10.48550/arXiv.2606.03754`.

12 Amir Herzberg and Shay Kutten. Fast isolation of arbitrary forwarding faults. In *Symposium on Principles of Distributed Computing (PODC)*, 1989.

13 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In *International Symposium on Distributed Computing (DISC)*, 2019. `doi:10.4230/LIPIcs.DISC.2019.27`.

14 Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2018.

15 Qianyu Yu, Giuliano Losa, and Xuechao Wang. TetraBFT: Reducing latency of unauthenticated, responsive BFT consensus. In *Symposium on Principles of Distributed Computing (PODC)*, 2024.