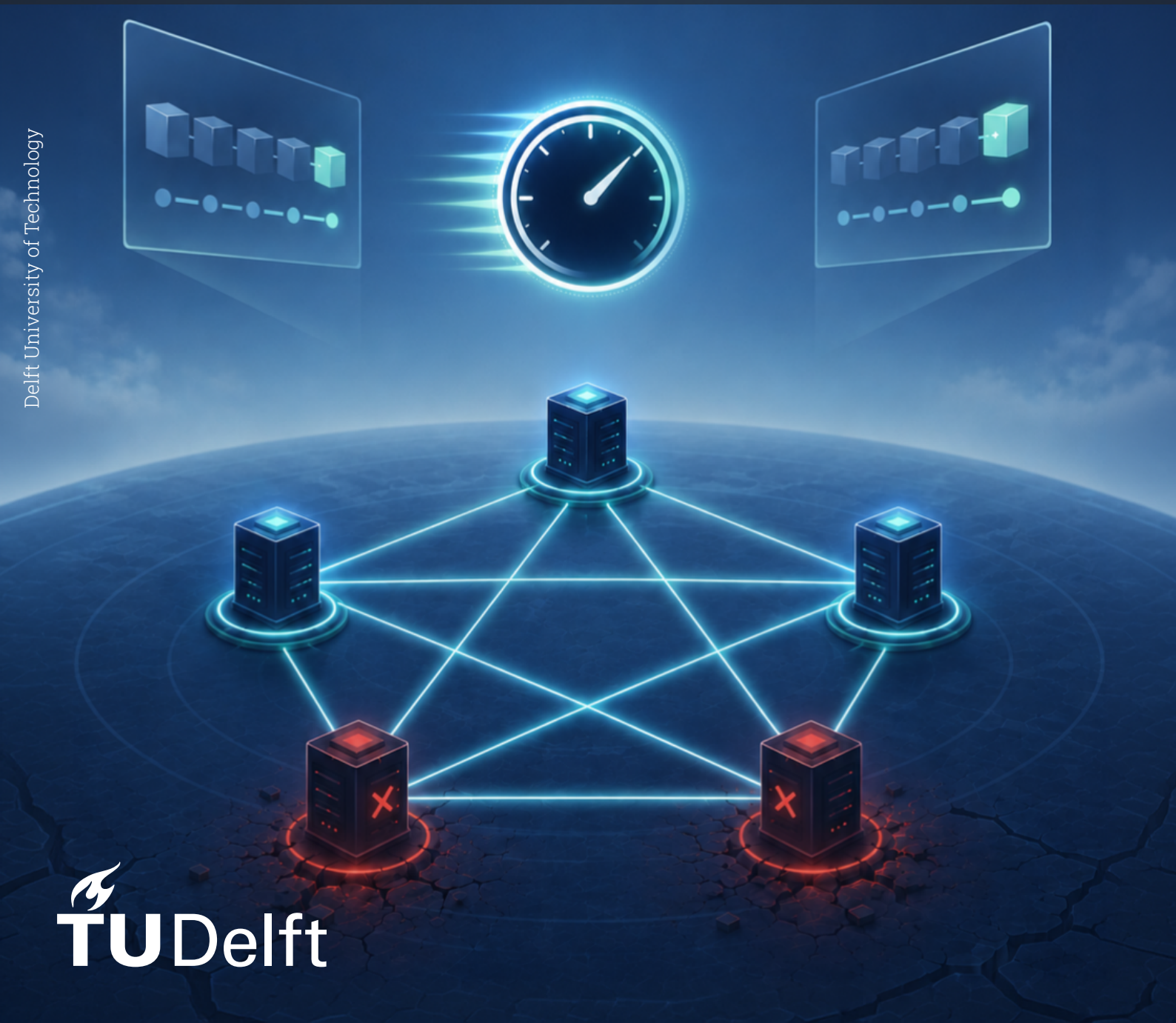
Latency-Optimal View Synchronization for 2f+1 BFT Consensus in the Presence of Faults

Daniel Lihotský



Delft University of Technology

TU Delft

Latency-Optimal View Synchronization for 2f+1 BFT Consensus in the Presence of Faults

by

Daniel Lihotský

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday July 14, 2026 at 10:00 AM.

Student number:	5518199	
Project duration:	November 12, 2025 – July 14, 2026	
Thesis committee:	Dr. Jérémie Decouchant,	TU Delft, supervisor
	Dr. ir. Harm Griffioen	TU Delft

Cover: Generated using OpenAI DALL·E and edited by the author (2026)

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Byzantine fault-tolerant (BFT) consensus underpins modern state machine replication and blockchain systems. Recent consensus protocols reduce the classical $3f+1$ replica requirement to $2f+1$ by leveraging trusted execution environments, but view synchronization in this setting has received little systematic attention and typically relies on exponential timeout backoff, which has unbounded worst-case latency. On the other hand, in the $3f+1$ setting, recent view synchronization work has focused on minimizing asymptotic message complexity, with limited consideration of the latency cost this imposes. This work addresses both gaps. We first adapt the message-optimal LP22 and Lumiere synchronizers to the $2f+1$ setting. We then design two latency-optimized synchronizers tailored to the $2f+1$ setting: a broadcast-based protocol and a leader-based protocol, which we call *Babette*. We implemented all four algorithms on top of the OneShot consensus protocol and evaluated them on the DAS-5 cluster across network sizes up to $n = 51$ and fault scenarios ranging from no failures to the maximum tolerated f. With f failures, *Babette* achieves around 5.5× higher throughput and correspondingly lower latency than Lumiere across all tested network sizes, demonstrating that asymptotic message complexity-optimality does not immediately translate to performance improvement when timeout constants dominate end-to-end performance.

CCS Concepts

• **Computing methodologies** → **Distributed algorithms**; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks.*

Keywords

Byzantine view synchronization, Byzantine fault tolerance, Consensus, Trusted component, Partial synchrony

1 Introduction

State machine replication (SMR) algorithms ensure that a set of replicas execute the same sequence of commands despite failures. They have become increasingly relevant in recent years due to their central role in blockchain systems, where maintaining a globally consistent ordering of transactions across distributed nodes is critical. The core building block of SMR are Byzantine fault-tolerant (BFT) consensus algorithms. These decide what the next command or block of transactions should be.

Most practical BFT protocols operate under the partially synchronous network model [15]. Unlike purely asynchronous models, partial synchrony allows the system to make progress once the network stabilizes. The partially synchronous network model assumes that messages are eventually delivered but without a fixed bound during periods of asynchrony. BFT protocols provide two key guarantees. *Safety* ensures that correct replicas never commit conflicting commands, and *liveness*, which ensures that the system keeps making progress despite failures or delays.

To ensure *liveness* most BFT consensus algorithms structure execution into *views*, which are logical rounds of the protocol during which a specific replica acts as the leader and is responsible for proposing the next command or block. For the protocol to make progress, correct replicas must agree on the current view, as it determines which messages the replicas should be responding to. Leader failures or partial network asynchrony can lead to replicas finding themselves in different views. This can lead to competing proposals and hinder progress, as replicas may follow different leaders or wait for messages associated with different views.

A *view synchronizer* is a software component of SMR that works alongside a leader-based consensus protocol to ensure that replicas eventually agree on the current view and remain in it long enough to make progress. In practice, view synchronization often becomes a performance bottleneck in modern SMR protocols. While many consensus protocols report high steady-state throughput under stable leadership, their evaluations typically do not account for the cost of view changes. However, in adverse conditions, view synchronization can be significantly more expensive than normal-case consensus, dominating overall performance.

Existing work on view synchronization focuses primarily on asymptotic analysis and minimizing message complexity under worst-case adversarial conditions, providing limited insight into practical performance [27–29, 34]. As a result, the cost of view changes and their impact on SMR efficiency under realistic conditions remain poorly understood.

Classical BFT consensus requires at least $n = 3f + 1$ replicas to be able to tolerate f Byzantine faults and guarantee liveness and safety [36]. The reason is that faulty replicas can equivocate, i.e., send conflicting messages to different participants, which could prevent correct replicas from agreeing on a value if there were only $2f + 1$. Recent work has shown that the replication requirement can be reduced to $2f + 1$ by strengthening the system model, most prominently through the use of Trusted Execution Environments (TEEs) [10, 11, 39]. TEEs enforce correct protocol behavior in hardware, preventing equivocation.

The question of view synchronization within TEE-based protocols has not been systematically studied making the

knowledge gap about their performance even more pronounced. In this paper, we address these gaps by designing and evaluating novel view synchronization algorithms specifically tailored for TEE-based $2f+1$ SMR protocols. Our work makes three main **contributions**:

- We show that the message complexity-optimal view synchronizers LP22 [27] and Lumiere [29] transfer directly to the $2f+1$ setting with only quorum-threshold adjustments, whereas other $3f+1$ synchronizers such as Cogsworth [34] do not.
- We identify the latency cost that message complexity-optimal synchronizers incur through their required per-view timeouts, and design two new $2f+1$ view synchronizers with tighter timeout bounds: a broadcast-based protocol and a leader-based protocol we call *Babette*.
- We implement all four algorithms on top of a consensus protocol called OneShot [11] and provide a systematic empirical evaluation of their throughput, latency, and message overhead on the DAS-5 cluster [1] across network sizes up to $n = 51$ and fault scenarios ranging from no failures to the maximum tolerated f.

2 Background

2.1 Byzantine Faults and SMR

The term *Byzantine fault* was introduced to describe arbitrary faults in distributed systems, including malicious or contradictory behavior that cannot be categorized as simple crash or omission errors. The term comes from the *Byzantine generals problem*, an allegory introduced by Lamport et al. [25] to illustrate the challenges of achieving agreement despite unreliable or malicious participants. Even before the coining of the *Byzantine fault* it was proven by Pease et al. [36] that deterministic Byzantine agreement in a synchronous network requires at least $3f + 1$ nodes to tolerate f arbitrary faults. These papers assume a negligible or a fixed, known upper bound on message delay. This is not realistic when compared to real-world conditions where messages get delayed and lost all the time.

Subsequent research explored the limitations of distributed consensus under different network assumptions, namely asynchrony, where messages between nodes can be sent and arrive with arbitrary delays. A key result is the FLP impossibility theorem [16], which shows that deterministic consensus cannot be achieved in a purely asynchronous system even with just a single faulty process. This result motivated the introduction of weaker timing assumptions. The partially synchronous model, proposed by Dwork et al. [15], assumes that messages can have arbitrarily long delays before some unknown Global Stabilization Time (GST), after which all messages arrive within a bounded delay Δ. This is the model used by most BFT protocols in practice as it reflects the behavior of real-world networks more closely than strict synchrony, while circumventing the FLP impossibility result.

The practical use of BFT consensus is in *State Machine Replication* (SMR) [26, 38]. In this approach, multiple replicas maintain identical copies of a deterministic state machine and process the same sequence of commands. Consensus protocols are used to ensure that all correct replicas agree on a single global ordering of client requests, which guarantees that their states remain consistent despite faults. This abstraction allows distributed systems to provide the illusion of a single reliable service, even when some replicas behave incorrectly.

2.2 PBFT, Views and View Change

The first practical BFT protocol for SMR was introduced by Castro and Liskov with the Practical Byzantine Fault Tolerance (PBFT) algorithm [4], which operates under the aforementioned partial synchrony network model. Consensus on a request proceeds in three communication phases: `pre-prepare`, in which the leader broadcasts its proposal to all replicas; `prepare`, in which replicas exchange messages to confirm that they have all seen the same proposal from the leader; and `commit`, in which replicas exchange a second round of messages to confirm that a quorum of them is ready to commit. A replica accepts a request as decided once it has collected a quorum of $2f + 1$ matching messages in the commit phase. The two acknowledgment phases (rather than one) are necessary because a Byzantine leader can equivocate by sending different proposals to different replicas, and the prepare phase is what allows correct replicas to detect agreement on a single proposal before committing to it. PBFT structures execution into views, each having a designated leader responsible for proposing the next batch of client requests. In order to guarantee progress, replicas must move on when a view does not reach a decision after a certain timeout period.

Most subsequent practical BFT protocols were developed using this view-based paradigm [3, 22, 24]. This view-based solution comes with a cost, however: the *view change* mechanism. Most BFT protocols optimize for performance after GST, without leader failures, without the account for view synchronization. But the view change, especially with consecutive leader timeouts, is a huge performance bottleneck, as pointed out by Clement et al. [7]. When a timeout happens, the nodes need to move together to the next view and that requires additional communication, effectively stalling the main algorithm's progress. Castro and Liskov's PBFT described an explicit view change protocol to replace a faulty

leader: When a node's timer for view v expires, it stops responding to messages regarding consensus and sends out a `VIEW-CHANGE(v+1)` to all other nodes. When the leader l of $v+1$ receives $2f$ of these view change messages it forms and sends out a `NEW-VIEW(v+1)`. If a node does not move to view $v+1$ by receiving `NEW-VIEW(v+1)` from l within a certain timeout period T, it doubles T and tries to move to view $v+2$. The timeout doubling ensures nodes will eventually be in the same view long enough for consensus to resume, guaranteeing liveness.

The introduction of HotStuff [40] made *streamlined* BFT protocols the norm. *Streamlined* protocols rotate the leader every view as part of normal execution, rather than only upon failure. This yields a cleaner, more uniform protocol structure and enables efficient pipelining, but it also makes the view synchronization mechanism performance-critical. Because view transitions are now on the common path, any inefficiency in view change directly impacts end-to-end latency and throughput. HotStuff was the first protocol to separate its core consensus logic from its view synchronizer, which is called the *Pacemaker*. However, the pacemaker implementation is left unspecified, making its performance in the presence of faults unknown.

Message complexity measures how many protocol messages are exchanged to complete one decision (or view transition), while latency captures how much time or how many message delays are/is needed before a decision is finalized. In BFT SMR, these metrics are complementary: message complexity reflects scalability and network load, whereas latency reflects confirmation time and responsiveness under failures. HotStuff's core consensus protocol (considered without Pacemaker costs) achieves linear communication, i.e., ($O(n)$) messages per decision in the steady state, and commits in three consecutive phases, giving a 3-delay optimistic commit path after GST under an honest leader. However, these guarantees concern the core agreement logic only. Since liveness is delegated to an unspecified Pacemaker, the fault-period cost of view synchronization is not captured by HotStuff's headline bounds.

2.3 View Synchronizer Evolution

The question of implementing the Pacemaker (view synchronizer) with optimal message complexity and latency became the subject of research by Naor et al. [34]. This is also the first paper that formalized and dubbed the problem as the *Byzantine View Synchronization* (BVS) problem. In a nutshell, view synchronization is the problem of coordinating views (leader changes) so that an honest leader eventually gets a long enough, uninterrupted window of synchronous communication with all peers. The flagship algorithm from this paper, *Cogsworth*, achieved synchronization in two rounds by nodes sending wishes to the next leader who returns a timeout certificate, after which nodes send votes to the leader who returns a quorum certificate. It incurred $O(n^2)$ expected message complexity and $O(n^3)$ worst-case message complexity in the presence of f Byzantine faults. Subsequent research focused mostly on improving these two metrics. An improved version of *Cogsworth* was described by Naor and Keidar [35] that improved the expected message complexity to $O(n)$ and slightly improved the worst-case, which remained $O(n^3)$.

A breakthrough came with two protocols published in 2022: RareSync [6] and LP22 [27]. These protocols improve the worst-case performance to the optimal $O(n^2)$ for message complexity. This matches the Dolev–Reischuk lower bound [14], which shows that any deterministic Byzantine agreement protocol tolerating f faults must exchange $\Omega(f^2)$ messages in the worst case. Both protocols use a similar approach of dividing execution into epochs that are each comprised of $f+1$ views. Instead of incurring $O(n^2)$ communication each time a leader fails, they synchronize perpetually, avoiding the f consecutive leader failure worst-case, which causes $O(n^3)$ in previous protocols. But without synchronizing immediately after detecting a failure, the protocol must rely on timeouts to traverse failed views, which means increased latency. Subsequent research by Lewis-Pye focused on lowering the worst-case latency first with Fever [28], which used initial clock-synchronization to achieve optimistic responsiveness and later Lumiere [29], which builds on the same principles but removed the need for clock synchronization.

2.4 Trusted Components and the $2f+1$ Setting

We previously stated that at least $3f+1$ total replicas are required to tolerate f Byzantine faults. The main reason for $3f+1$ instead of just $2f+1$ (the minimum amount where correct nodes form a majority), is that Byzantine leaders can equivocate, sending conflicting proposals to different replicas. In BFT protocols, decisions are made based on quorums, i.e., sufficiently large subsets of replicas whose agreement is required to accept a value. To ensure safety, any two quorums must overlap in at least one correct replica. Equivocation allows faulty replicas to appear consistent to different subsets of nodes, so quorums must be large enough to guarantee this overlap despite up to f faulty replicas, which leads to the $3f+1$ requirement. The problem is that the performance cost of f additional replicas is very high. For better defense against adversaries. replicas need to be diverse, have different operating systems, use different software, etc., to avoid sharing the same vulnerabilities. This involves considerable

additional costs per-replica, much more than just their hardware.

If equivocation can be prevented, the required amount of replicas can be shrunk down to $2f+1$. Earliest BFT protocols entertaining this idea were proposed by Chun et al. [5], using trusted monotonic counters and attested append-only logs, and Correia et al. [9], using secure hardware modules to generate unforgeable, non-replayable messages. Both of these rely on specialized trusted components that are not widely deployed and require additional hardware assumptions beyond standard systems. Subsequent research aimed to simplify these assumptions making the trusted components more easily verifiable [23, 39], but still used a custom designed components, not something actually found on real-life CPUs. More recent work [10–12, 33] leverages Trusted Execution Environments (TEEs), hardware-protected enclaves available on many contemporary CPUs. TEEs provide a stronger and more flexible abstraction that can enforce arbitrary protocol logic inside the trusted boundary, simplifying protocol design and reducing reliance on ad hoc assumptions.

All of these protocols work with the same high level idea, the only difference is the execution. They all rely on some sort of monotonic counter that lives inside of a trusted component (so the malicious actor cannot tamper with it) that works as a sort of unique sequence number generator for messages. To send a message for a given consensus decision step, the node needs to include a signature on the sequence number that verifiably came from the trusted component. A given consensus step has an assigned sequence number, and the trusted component can never produce a signature for that given number more than once. This means that if an adversary wanted to send conflicting messages to two different peers for the same step, they would need to ask the trusted component for a signature two times for the same number, which is impossible by design.

2.5 View Synchronization under $2f + 1$

View synchronization mechanisms in $2f + 1$ BFT papers vary, and it is evident that handling view changes and view synchronization was not the primary focus of their research. Main struggle is to make sure that nodes left behind by a quorum of Byzantine nodes plus one correct node have the ability to "catch up" once the Byzantine nodes stop cooperating and leave the correct node in the forward view not able to continue, more detail on how this happens is shown in Section 5.2.

Correia et al. [9] eliminates the need for view changes entirely in the main protocol layer. There is no primary nor leader among the servers for ordering, that is delegated to the trusted component's TMO (Trusted Multicast Ordering) service, which runs inside a trusted, synchronous subsystem. Chun et al. [5] has two rounds of all-to-all view change messages, sending `WANTVIEWCHANGE` and gathering $f + 1$, then moving second round sending `VIEWCHANGE` and after gathering $f + 1$ starting the new view. There is a custom catch up mechanism to solve the issue where nodes that have fallen behind request a certification from the forward node that then sends back a "proof" of how it got there, this mechanism is recursive if node is more than one view ahead. Two protocols described in [39], MinBFT and MinZyzzyva, move to a new view after requiring just $f + 1$ once and ensure catch up with doubling timeouts, more explanation on how this works in Section 4.1. CheapBFT [23] falls back on MinBFT in unhappy periods, therefore having the same view change mechanism. FastBFT from [33] also moves view after one round of $f + 1$ view change messages and even acknowledges the issue of nodes falling behind, but gives no explanation on how nodes manage to catch up afterwards.

More recent protocols use the streamlined approach which rotates leaders round-robin after every consensus decision. This means nodes no longer explicitly have to send messages to advance view (they just move after executing a request), but still need a mechanism to replace a faulty leader and catch up if left behind. Two more recent streamlined $2f + 1$ protocols by Decouchant et al., Damysus [10] and OneShot [11], follow the same approach as their predecessor HotStuff [40] and leave the synchronization mechanism unspecified. They only state that it is done "through a mechanism that allows correct nodes to eventually be in the same view for long enough to make progress", with exponential backoff given as an example. This of course refers to the nodes timing out twice as long each consecutive time, similar to PBFT and protocols previously discussed. The most recent research on $2f+1$ BFT by Decouchant et al. [12], however, does also focus on view synchronization and dub their view change protocol *Aegis*, "the first efficient view synchronizer specifically designed for BFT protocols that utilize trusted components." They adapted the Cogsworth view synchronizer [34] to aid their rollback preventing rejoin mechanism.

3 Assumptions - The System Model

As mentioned before we assume a partially synchrony network model [15]. To reiterate, this model assumes that there is a period of asynchrony during which messages sent can have unbounded delays, followed by a period of synchrony where messages arrive within Δ. The network changes from asynchronous to synchronous at some unknown moment (GST). Let δ denote the actual time a message took to arrive at its destination, while Δ is a set constant. Let Δ be the maximum amount of time we allow a message to take to arrive. In other words, it is a prediction of $\max(\delta)$. Setting

Δ correctly is critical for the correctness of view synchronization algorithms, since messages arriving later than Δ may be treated as if they were never sent. This comes directly from the definition that messages sent at time t arrive by time $max(GST, t) + \Delta$. A direct consequence is that all timeout-based reasoning in the protocols we discuss is only sound after GST. We assume this implicitly in all timeout arguments that follow. This definition also implies that messages do not get lost before GST, only get delayed to arrive after GST. Processes can start execution at any point before GST and each of them has a local clock that can reliably keep track of time after GST without significant drift.

We assume that the system size is exactly $n = 2f + 1$, f denoting the maximum amount of Byzantine faults we can tolerate and f_a the number of actually faulty nodes corrupted by adversaries[1]. We assume the adversaries can cooperate and corrupt any subset of up to f replicas. The remaining nodes will be referred to as *honest* or *correct*. The network is also fully connected with point-to-point authenticated channels, meaning all nodes can send messages directly to each other node and every message is guaranteed to be attributed to its true sender, i.e., the receiver can reliably identify which node sent the message. We assume a public key infrastructure (PKI) cryptographic scheme is in place which each node can use to sign messages and verify that a content of a message is unaltered and came from a certain sender. We assume an adversary is polynomial-time bounded and the probability of breaking this cryptography scheme is negligible.

At the time of writing, modern BFT consensus protocols achieving a $2f + 1$ fault tolerance threshold rely on trusted execution environments (TEEs) or comparable trusted hardware assumptions. Our view synchronizers assume that the underlying consensus protocol operates with $n = 2f+1$ replicas and $f+1$ quorum thresholds, however, they do not invoke TEE operations themselves nor depend on the presence of trusted hardware. We assume the underlying consensus protocol execution is divided into views that advance after each consensus decision. Each view has a designated leader, determined by a schedule known to all nodes in advance. This last point is not strictly necessary and some older protocols like PBFT and Zyzzyva [24] only change views during a failure, however, our focus is on optimizing performance under periodically rotating leaders which is the standard in modern BFT protocols [10–12, 40]. We further assume a known upper bound on view completion: if at least $f + 1$ honest nodes enter the same view under an honest leader at time t, then all of them complete the view by time $t + x\delta$, where the multiplier x is derived from the number of communication phases the consensus protocol requires. This maximum view duration will later be used to set view timeouts, with unknown δ being substituted with Δ.

[1] $n \geq 2f + 1$ will also work, but is not standard

3.1 Byzantine View Synchronization Problem

We now describe the problem we aim to solve. Nodes start in view 0 but do not necessarily do so simultaneously. Within a view, nodes execute the underlying consensus protocol independently of the view synchronization layer. At the end of each view, the consensus layer signals the view synchronization layer that it wishes to advance. Consensus execution is then paused until the view synchronization layer signals back that the next view should begin. The view synchronization layer is required to eventually bring all honest nodes into the same view for long enough that consensus can make progress, thereby ensuring *liveness*. Without such a mechanism, Byzantine behavior and asynchrony can leave honest nodes scattered across different views indefinitely.

4 Existing Approaches

4.1 Existing $2f + 1$ View Synchronization

The existing $2f + 1$ view synchronization solutions discussed in Section 2.5 all rely on assumptions or mechanisms that make them unusable or highly inefficient in our setting:

- **Correia et al.** [9] eliminates view changes entirely by delegating message ordering to a synchronous trusted subsystem, which is a stronger and more intrusive hardware assumption than we are willing to adopt.
- **Chun et al.** [5] handles catch-up via point-to-point requests to potentially faulty replicas, offering no bound on the number of round trips required.
- **MinBFT** and **MinZyzzyva** [39], and by extension **CheapBFT** [23] which falls back on MinBFT, rely on exponential timeout doubling to let straggling nodes catch up, which has unbounded worst-case latency (discussed below).
- **FastBFT** [33] advances views after one round of $f + 1$ messages but leaves the catch-up problem unresolved, so it does not provide a complete view synchronization solution.
- **Damysus** [10] and **OneShot** [11] leave the synchronization mechanism unspecified, giving exponential backoff as an example.
- **Pallas** [12] introduces the *Aegis* synchronizer, a modified version of Cogsworth [34]. However it is tightly coupled to Pallas's custom rejoin mechanism and cannot be extracted for use with other $2f + 1$ consensus protocols.

The recurring theme in the above is exponential backoff, used by several of these protocols (MinBFT, MinZyzzyva,

CheapBFT, Damysus, OneShot) to ensure that nodes eventually wait long enough for stragglers to catch up. Each view is assigned a timeout window of Γ, and whenever this window expires without progress the timeout is doubled: 2Γ for the next view, then 4Γ, 8Γ, and so on. While this strategy guarantees liveness, its latency cost grows exponentially in the number of consecutive failed views. A node left behind in view 0 and attempting to catch up to view 5, for instance, must wait $\Gamma + 2\Gamma + 4\Gamma + 8\Gamma + 16\Gamma = 31\Gamma$ before synchronizing.

Such scenarios are well within the capabilities of our adversarial model. An adversary can control f replicas that are leaders in f consecutive views, forcing a correct node through the full doubling sequence. More generally, f consecutive Byzantine leaders incur a worst-case latency of $(2^f - 1)\Gamma$, which grows exponentially with f.

Rather than adapt any of these protocols, we look to view synchronization techniques originally designed for the $3f+1$ setting, adapting them to $2f + 1$ where necessary. The latest such protocol is Lumiere [29], which we build up to by first examining its predecessor LP22.

4.2 LP22

To understand Lumiere, we first examine how LP22 [27] achieves the optimal $O(n^2)$ message-complexity lower bound, in contrast to earlier solutions that require all-to-all communication at every view change. Lewis-Pye et al. observe that if each view change incurs $O(n^2)$ view-sync messages, then f consecutive leader failures push the total cost of finding a correct leader and fully synchronizing to $O(n^3)$. To avoid this, they propose a Byzantine view synchronization (BVS) algorithm that divides execution into epochs of $f + 1$ views, each guaranteed to contain at least one correct leader.

Within an epoch, advancing from one view to the next proceeds as follows: nodes send wish messages to the next leader, who aggregates them into a view certificate (VC) and broadcasts it to all nodes. The VC serves as proof that a quorum of nodes wished to enter the next view, and any node entering the new view does so upon receiving it. At the end of each epoch, nodes send wishes to each of the $f + 1$ leaders of the upcoming epoch. Each such leader produces an epoch certificate (EC) and broadcasts it. Any node that receives an EC while still in a lower epoch forwards it to all other nodes. We refer to the intra-epoch exchange of wishes and VCs as *light* view synchronization, and to the inter-epoch exchange of wishes and ECs as *heavy* view synchronization. Figure 1 illustrates both. Horizontal lines indicate nodes P_{1-4}. Blue arrows indicate messages being sent from one node to the other, with the type of message being sent written above. The x axis denotes time: an arrow moving from left to right indicates it was sent before it arrived.

Under this paradigm, f consecutive leader failures incur maximum $n^2 + 3n(f + 1)$ messages, which is $O(n^2)$ because f is a constant fraction of n.

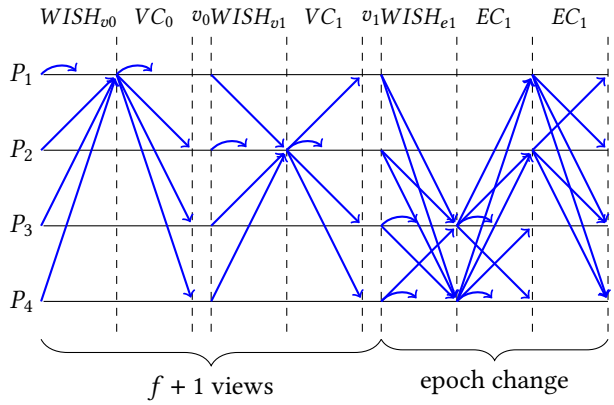


Figure 1: Visualization of the LP22 view synchronization algorithm under $n = 3f + 1$ with $f = 1$. Consensus messages are left out for brevity - would fall under v_0 and v_1

Because all-to-all communication occurs only at epoch boundaries, f consecutive faulty leaders incur only $O(n^2)$ messages. This comes at a cost in latency, however. Each view is assigned a fixed time slot of length Γ by which it must complete. If the slot for view v expires without progress, nodes time out and wish to advance to the next view. Conversely, if the final consensus step (typically referred to as the QC in BVS literature) is produced and delivered before the slot ends, nodes wish to advance immediately.

This design creates an undesirable asymmetry. If several views complete quickly at the start of an epoch and a faulty leader appears near its end, nodes cannot exploit the time saved earlier. They must still wait for the full slot of the failed view to elapse before timing out. In the worst case, a single faulty leader at the end of an epoch forces nodes to wait up to $(f + 1)\Gamma$ before synchronizing. Figure 2 illustrates this scenario. The algorithm, originally designed for $n = 3f + 1$, applies directly to our $n = 2f + 1$ setting just by altering quorum thresholds.

4.3 Lumiere

The Fever protocol [28] addresses LP22's latency problem by advancing each node's timer forward upon every successful QC, ensuring that any single faulty leader incurs at most a one-view delay. However, it requires initial clock synchronization, which adds an undesirable assumption to our model. The Lumiere protocol [29] combines the ideas of LP22 and Fever into a responsive view synchronization protocol that resolves both protocols' main shortcomings

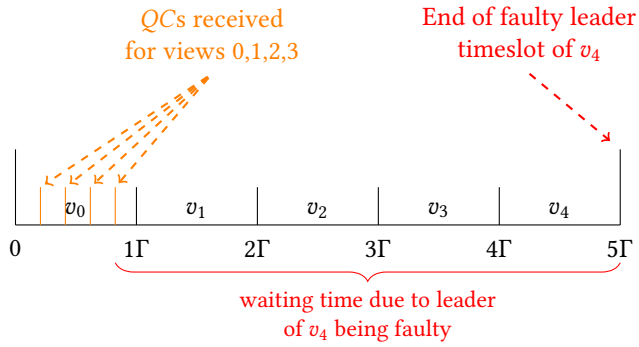


Figure 2: Visualization of one epoch of LP22, comprising 5 views, illustrating latency cost of one leader failure at the end of the epoch. Duration of one view is maximum Γ, indicated on the x axis

while preserving their strengths. Lumiere requires no clock synchronization, retains the optimal $O(n^2)$ message complexity, and applies directly to our $n = 2f + 1$ setting with only quorum-threshold adjustments.

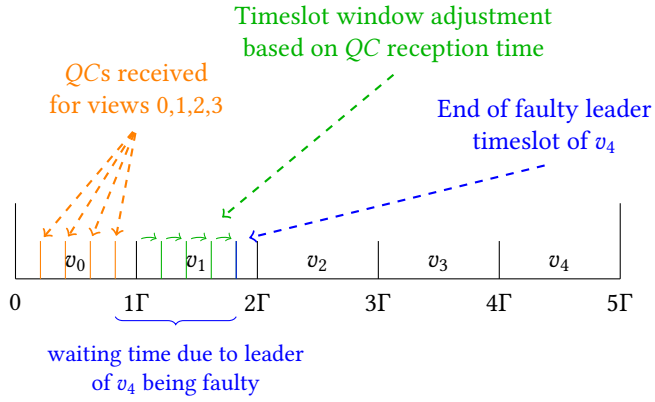


Figure 3: Visualization of one epoch of Lumiere, showing the adjustment of timeout windows, mitigating the problem from Figure 2

4.4 Latency and Timeouts of LP22 and Lumiere

LP22 and Lumiere set a time window for each view, by which the view must conclude (after GST) and use this window as the timeout value. In the LP22 paper [27] this window is set to 12Δ per view. The derivation of the constant 12 is not given, but we can infer it as follows: at least $x\Delta$ is needed for consensus completion, 2Δ for the view change, and Δ as a margin of entry, since nodes can enter an epoch up to Δ apart and reset their timer at that point. This gives a generalized formula of $(x + 3)\Delta$.

Lumiere sets the view timer to $\Gamma = 2(x + 2)\Delta$. We omit the full derivation, which is given in the original work [29, Section 5, Lemmas 5.8 and 5.12]. An important nuance is that each leader is allocated two consecutive views, one initial and one non-initial. The authors of Fever [28], from which this mechanism originates, note that Γ can in principle be reduced arbitrarily close to $(x + 1)\Delta$ by allocating $k > 2$ consecutive views per leader. This reduction is impractical for our goal for two reasons. First, a single faulty leader would waste $k\Gamma$ time rather than 2Γ, meaning the worst-case latency penalty from a single Byzantine leader grows linearly with k and is unbounded in the limit. Second, reducing Γ diminishes the per-leader gap-shrinking margin that Lumiere's epoch mechanism [29, Section 3.5] relies on, potentially requiring longer epochs and undermining the protocol's steady-state efficiency. In practice, Lumiere therefore operates with $\Gamma = 2(x + 2)\Delta$, which is strictly greater than the ideal $(x + 1)\Delta$ by a factor of roughly 2. Even in the best case, this technique can only match, never exceed, the $(x+1)\Delta$ latency minimum, while introducing an unbounded worst-case penalty.

5 Design

5.1 Motivation

The tradeoff for optimal message complexity are larger required timeouts. LP22, and by extension Lumiere, sacrifice tight synchronization in favor of deferring heavy synchronization until the end of the epoch and relying on timeout windows for correctness. We therefore design a view synchronization algorithm that keeps the minimum necessary timeout as short as possible, lowering latency in the presence of faults.

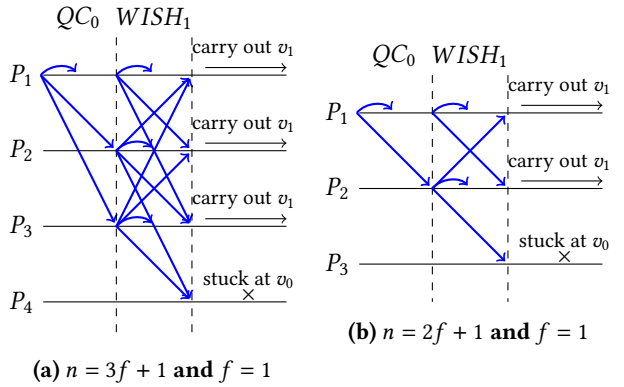
Two algorithms proposed by Naor et al. [34], the flagship Cogsworth and a broadcast-based algorithm from the appendix, have much lower timeout requirements than LP22 and Lumiere. We will use the same design principles to design our view synchronization protocol. Neither algorithm can be used directly under the $2f + 1$ paradigm because of an *echoing* mechanism, which is explained in detail in Section 5.2. We therefore design a new protocol from scratch.

5.2 The Catch-up Problem

We begin by ensuring that at least a quorum of nodes wishes to advance before any node moves to the next view, so that the new view is able to make progress. To achieve this, every node wishing to advance to view v broadcasts a $WISH_v$ message, where *broadcast* denotes sending the message to all nodes in the network, including the sender itself. In practice, sending a message to oneself amounts to invoking the message-handling routine locally with $sender = self$. Advancing to view v immediately upon receiving a quorum of $WISH_v$ messages is not sufficient with $2f + 1$ replicas. A

Byzantine leader can send the QC to only a subset of nodes, just enough to form a quorum, causing those nodes to wish to enter the next view. If the f Byzantine nodes then direct their own wishes exclusively to this subset, the remaining honest nodes are left behind with no means of catching up. This becomes a problem if the f Byzantine nodes stop participating in the consensus. The honest nodes get split into two groups, one ahead and one behind, neither able to reach a quorum on its own. The protocol stalls and the liveness property is broken. Figure 4 illustrates this attack for both the $3f + 1$ and $2f + 1$ settings.

Naor et al. [34] resolve this efficiently under $3f + 1$ by applying principles from the Bracha reliable broadcast protocol [2]. An honest node that receives $f + 1$ wishes for view v without having sent its own $WISH_v$ *echoes* the wish by broadcasting one itself. This guarantees that even if the f Byzantine nodes drag $f + 1$ honest nodes into the next view to form a quorum of $2f + 1$, the remaining honest nodes still receive at least $f + 1$ wishes, generate their own, and accumulate enough to advance together.



(a) $n = 3f + 1$ **and** $f = 1$ **(b)** $n = 2f + 1$ **and** $f = 1$

Figure 4: Byzantine leader P_1 of view 0, sending the last step of consensus (QC_0) and wishes for view 1 ($WISH_1$) to subset of nodes, just enough for a quorum, making the rest get left behind with no way to catch up

The same principle cannot be applied directly to $2f + 1$, because the Byzantine nodes need to bring along only one honest node to form a consensus quorum of $f + 1$. In the $3f + 1$ case, echoing after receiving $f + 1$ wishes guaranteed that at least one of the senders was honest and therefore that the wish to advance was well-founded. In the $2f + 1$ case, the analogous rule would require echoing after just a single wish, namely the one sent by the honest node dragged forward. Such a wish, however, could equally have originated from a Byzantine node attempting to advance prematurely.

5.3 Catching Up Using View Certificates

We need a mechanism by which the lone forward node proves that it received enough wishes to advance, and broadcasts this proof so that the remaining nodes can follow. We call this proof a *view certificate* (VC) and construct it as follows. Each node signs its wish under the PKI before sending it out, where a wish is represented by a short string identifying the message type and the target view (e.g. `"WISH5"`). When a node has collected enough wishes to advance, it combines the signed wishes from distinct nodes into a single VC message, broadcasts the VC, and moves to the next view. Upon receiving a VC for a view higher than its current one, a node verifies the signatures and, if they are valid, advances to that view. In this way, even nodes that did not directly receive enough wishes are guaranteed to advance, since at least one correct node must have collected a quorum, formed a VC, and broadcast it.

However, allowing a node to immediately advance to a view upon receiving a corresponding valid VC introduces a new problem. Byzantine nodes can mount an attack similar to the one in Figure 4, except that they now exchange wishes only among themselves. In this scenario, the single honest node that receives the QC wishes to advance, and the Byzantine nodes combine its wish with their own f wishes to form a VC, which they then deliver only to a subset of honest nodes, leaving the rest behind. Figure 5 illustrates this attack.

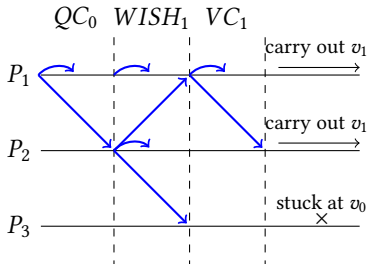


Figure 5: Byzantine leader P_1 of view 0, sending the last step of consensus (QC_0) only one node, withholding $WISH_1$ and becoming the only node able to form VC_1 and sending it to one node, just enough for a quorum, making the rest get left behind without a way to catch up

A simple fix is to require honest nodes to rebroadcast any VC they receive. More generally, combining this with the earlier broadcast rule, a node must broadcast a VC whenever it changes view, whether by constructing the VC itself or by forwarding one received from another node. This already yields a correct view synchronizer with optimal worst-case asymptotic latency. The adversary can incur a maximum

latency of $O(f \cdot \Delta)$, by forcing nodes to time out f times in succession with each timeout bounded by a constant multiple of Δ.

5.4 Babette

The broadcast-based algorithm described above is highly inefficient in terms of message complexity. Each view change requires roughly $3n^2$ view-sync messages in the worst-case, but this number can be reduced substantially. Since Byzantine nodes always need at least one honest node to advance with them, and since that honest node must broadcast a valid VC to everyone in order to do so, there is no need for nodes to send their wishes to all other nodes. Once a single honest node moves, the rest follow. It therefore suffices for nodes to send their wishes only to the leader of the next view, who is then responsible for assembling the VC. While this concentrates more responsibility in the leader, no leader action can cause honest nodes to desynchronize. Under this scheme, each view change incurs only $n^2 + 2n$ view-sync messages. We refer to this leader-based algorithm as *Babette*[2]. The pseudocode is given in Algorithm 1 and a visualization in Figure 6. A proof that *Babette* achieves view synchronization is provided in Appendix B.

An important and subtle distinction between the two algorithms lies in how view transitions occur. In the broadcast-based algorithm, nodes always advance to the next view, regardless of whether the leader is responsive. In *Babette*, a node advances only upon receiving a valid VC from the next leader. This distinction is significant for timeout configuration. In *Babette*, once our assumptions about Δ hold (i.e., after GST), the absence of a VC within a short time window is itself evidence that the leader is faulty, whereas in the broadcast-based algorithm a node must wait out the full consensus window before concluding that the view should be skipped. We discuss timeouts in detail in the next section.

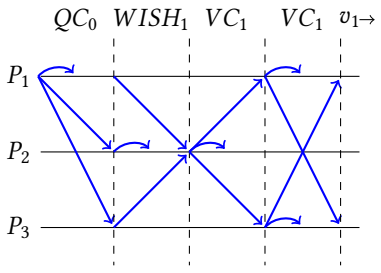


Figure 6: One round of view synchronization between v_0 and v_1 of *Babette* visualized with $n = 3$, $f_a = 0$, the QC_0 comes from the consensus layer

[2]The name is drawn from Disney's *Beauty and the Beast*, where Babette, a feather duster, is Lumiere's love interest.

Algorithm 1 Babette - Instructions for node a

upon completion of view $v - 1$ OR timeout for $v - 1$:
  send $WISH_v$ to leader of v

upon receiving $WISH_v$ as leader of v:
  if $f + 1$ distinct and verified $WISH_v$ received:
    if $v >$ current view:
      start view v
    form VC_v
    broadcast VC_v

upon receiving VC_v from b:
  if VC_v signatures valid AND $v >$ current view:
    start view v
    send VC_v to **all** except b

5.5 Latency and Timeouts of Babette

The work of Naor et al. [34] defines a parameter α specifying how often the underlying consensus protocol must invoke `wish_to_advance()` for view synchronization to remain valid. This α has a lower bound which is determined by the maximum time interval within which any two honest nodes can enter the same view under a given view synchronization protocol. For the broadcast-based algorithm this bound is 2δ; for Cogsworth it is 4δ. For *Babette*, a δ bound can be derived as follows.

The first honest node to enter view v is either the leader of v, if honest, at the moment of receiving its $(f + 1)^{\text{st}}$ wish, or, if the leader is Byzantine, the first honest node to receive the corresponding VC_v. Let t denote this moment. In either case, the entering node broadcasts VC_v, and all remaining honest nodes receive it and enter view v by time $t + \delta$. Hence honest nodes enter the same view at most δ apart, giving $\alpha = \delta$ for *Babette*.

Setting the timeout to Δ alone does not suffice, since nodes must also be allowed sufficient time to complete consensus within the view. The time required by the consensus layer depends directly on the number of communication steps of the underlying protocol, which we denote by x. This yields a total per-view timeout window of $x\delta + \delta$. In practice, then, a node times out after $(x + 1)\Delta$ from the moment it first enters the view.

For *Babette*, since honest nodes cannot enter view v more than Δ apart (shown earlier), they also time out of view v at most Δ apart. The last honest node sends its $(f + 1)^{\text{st}}$ wish to the leader of $v + 1$ at time $t = \Delta$; the leader receives it by 2Δ and broadcasts VC_{v+1}, which reaches all honest nodes by 3Δ. Hence if a node has timed out of view v and has not received VC_{v+1} within a further 3Δ, it can safely conclude that the leader of $v + 1$ is also faulty and wish to advance directly to

$v + 2$. Only upon successfully entering a view should a node restart its $(x + 1)\Delta$ timer.

The same short timeout applies to the transition between two successful views. If view v completes at time t (with QC_v or an equivalent received), then view $v + 1$ should begin by $t + 3\Delta$; otherwise, the node times out. These timeout bounds are summarized in Table 1. For LP22 and Lumiere we only have an overall timeout window of Γ while for *Babette* and Broadcast we split the Γ into two components one for a required timeout within a view and one for between views.

Table 1: Required timeout settings for each $2f + 1$ view synchronization protocol using an underlying consensus protocol with x communication steps

Protocol	Γ	
	Intra-view	Inter-view
Babette	$(x + 1)\Delta$	3Δ
Broadcast	$(x + 1)\Delta$	N/A*
LP22	$(x + 3)\Delta$	
Lumiere	$2(x + 2)\Delta$	

* An inter-view timeout value for the broadcast-based protocol is not required. It always successfully moves to the next view.

5.6 The Hidden Advantage

All BVS protocols discussed for the $2f + 1$ setting share the same worst-case asymptotic latency of $O(n\Delta)$. The constant factors hidden by this notation, however, differ substantially. Lumiere's view timer is fixed at $2(x + 2)\Delta$, and LP22 inherits the late-epoch waiting issue discussed in Section 4.2. *Babette*, by contrast, operates with a per-view timeout of $(x+1)\Delta$ and a short inter-view timeout of 3Δ, both of which are lower than the required timeout window Γ in Lumiere and LP22.

The practical consequence is that whenever Byzantine leaders force timeouts, *Babette* spends substantially less time stalled before advancing past a faulty leader than either Lumiere or LP22, with the gap widening as the number of consecutive faulty leaders grows. In fault-free executions the three protocols should behave similarly, since no timeouts are triggered and progress is driven by successful *QC*s. We quantify this fault-dependent reduction in stall time, and its effect on end-to-end consensus throughput, empirically in the following sections.

6 Implementation

6.1 Existing Codebase

To implement and evaluate our $2f + 1$ view synchronization algorithms, we require an underlying $2f + 1$ consensus protocol to attach them to. We first considered the most recent work in this area by Decouchant et al. [12], but its custom rejoining mechanism is tightly coupled to its view synchronizer. Since our synchronizer is designed for general $2f + 1$ consensus, integrating it would require substantial modifications to the protocol. We therefore adopt OneShot [11] as the base protocol. OneShot is a general-purpose, streamlined $2f + 1$ design with linear communication and strong empirical performance, which provides a cleaner and more representative foundation for evaluating our synchronizer.

We build on the existing codebase originally developed by Vincent Rahli for Damysus [37], which also contains an implementation of OneShot. The codebase is written in C++ and follows a process-per-replica deployment model. Each node is initialized with a static membership and configuration, and operates as an independent replica executing the protocol logic. Replicas communicate via authenticated point-to-point messages over the Salticidae [13] event-driven networking stack. Protocol handlers manage view progression, proposal and vote exchange, timeout handling, and leader rotation.

OneShot's TEE is instantiated with Intel SGX [8], a widely deployed enclave technology with mature SDK support and remote attestation. This makes it a practical choice for running trusted protocol logic on real hardware. The codebase can be built in either SGX hardware mode or SGX simulation mode. Simulation mode preserves the full enclave interface, including the ECALL/OCALL flow and the trusted/untrusted split, so the enclave logic runs exactly as it would on SGX hardware, just without the hardware-backed isolation guarantees.

Cryptographic operations rely on OpenSSL [20] for both hashing and digital signatures, using libcrypto on the untrusted side and SGXSSL inside the enclave. Keys are provisioned before execution: each replica loads its own key pair from a set of pre-generated key files, using the provisioning flow provided with the codebase.

6.2 Adding View Synchronization

The stock OneShot configuration includes no view synchronization mechanism. Each node simply enters the next view immediately after executing the request from the previous one. We extend OneShot by inserting the necessary synchronization logic between the completion of one view and the start of the next.

Each of the four protocols we evaluate — LP22, Lumiere, the broadcast-based algorithm, and *Babette* — is implemented on a separate branch of the codebase, rather than combined behind a runtime selector. Merging all four into a single branch would require pervasive conditional logic that scales poorly with the number of synchronizers and obscures the differences between them. Keeping the branches separate preserves the clarity of each implementation and simplifies maintenance.

Correctness was validated through aggregated logs collected across all nodes. Each node logs every message send and receive, every timeout, and every clock advance, allowing the full protocol trace to be reconstructed post-hoc and any deviation from expected behavior to be traced to its source. Experiments are orchestrated by a Python script that starts and terminates the replica processes, tracks per-run statistics, executes the configured number of repetitions, and writes aggregated results to disk alongside the raw logs.

The full implementation is available on GitHub [32].

7 Methodology

7.1 Environment

We conducted our experiments on the DAS-5 [1] (Distributed ASCI Supercomputer 5), a six-cluster wide-area research infrastructure developed within ASCI [17] as a shared platform for parallel and distributed computing research. Its dedicated wide-area interconnect and controlled multi-node environment make it well suited for evaluating distributed protocols. DAS-5 does not provide SGX-capable hardware, so all experiments are run in SGX simulation mode.

DAS-5's per-node allocation model is well suited to our use case, since we can run each replica on a dedicated node, closely mirroring how an SMR system would be deployed in practice. All experiments are conducted on the VU cluster, the largest of the six DAS-5 clusters, which allows us to scale to the largest possible network sizes. Of its nodes, typically 40–50 are available at any given time, and this defines the upper bound on the network sizes we evaluate. Detailed node specifications are available on the DAS-5 website [18]. We did not use the newer DAS-6 [19], as its clusters are considerably smaller and its design is oriented toward machine learning rather than distributed systems research.

7.2 Measured metrics

We adopt the same experimental configuration used by Damysus [10], OneShot [11], and Pallas [12], whose codebase we extend. Each block contains 400 transactions with 0 B payloads, and a single client drives the workload. **Throughput** is measured in Kops/s (kilo-operations per second), the number of transactions decided per second, expressed in thousands.

Note that 0 B payloads do not mean empty blocks. Each transaction still carries 40 B of metadata (client id, transaction id, and previous block hash), so a block of 400 transactions amounts to 15.6 KB on the wire. Using 0 B payloads isolates the protocol-level overhead of consensus and view synchronization without confounding measurements with application-layer transaction-processing costs.

The specific block size of 400 transactions is not load-bearing for our evaluation. What matters is that it is held constant across all measured configurations, so that observed performance differences are cleanly attributable to the view-synchronizer changes rather than to varying batching behavior. For the same reason, a single client is sufficient. Our goal is to compare synchronizer variants under identical conditions, and additional clients would not provide further insight into synchronizer behavior.

Latency is reported in milliseconds (ms) and measures the average time to complete a view. Each replica starts a timer at the beginning of execution and stops it once the final request is executed. The elapsed time is averaged across all honest nodes and then divided by the number of successful views. Lower latency means better performance.

Note that throughput and latency are tightly coupled by construction in view-based BFT SMR, since each view produces exactly one committed block. We report both because they answer different practical questions. Throughput reflects command rate, latency reflects commit time, but the two should be understood as complementary views of the same underlying behavior.

Each honest node tracks the number of view synchronization messages it sends over the course of a run, namely wishes, *VC*s, and *EC*s. The reported **view sync messages** metric is the average per-node message count divided by the number of successfully completed views, giving the mean synchronization overhead incurred per honest node per successful view.

Other metrics that we measured, that can be seen in Appendix C, are:

- **Handle** - average time in milliseconds spent on handling messages after reception per honest node
- **Sign dur.** - average time in milliseconds spent on signing view sync messages per honest node
- **Verif. dur.** - average time in milliseconds spent verifying view sync messages per honest node
- **Sign num.** - average amount of signatures generated for view sync messages per honest node
- **Verif. num.** - average amount of view sync message signatures verified per honest node

7.3 Setting Delta and Timeouts

Choosing a value for Δ involves a tradeoff between performance and tolerance for network delays. A smaller Δ is desirable for performance, since each failure can trigger several Δ-length waiting periods. Setting Δ too low, however, causes premature timeouts, which abort in-progress consensus rounds and skip potentially correct leaders. This wastes work already done and forces the system to expend additional effort on resynchronization. Although the partial synchrony model guarantees eventual progress even when Δ is temporarily violated, entering this recovery regime is

costly, and staying within the synchronous bound is preferable. In practice, Δ is therefore set conservatively.

We calibrate Δ from end-to-end RTT measurements that include transport and library overhead. Taking the maximum RTT observed across all node pairs on the DAS-5 and halving it yields a conservative one-way estimate. The maximum RTT measured in our experiments was 9138.023 µs, giving a base value of $\Delta_{\text{base}} \approx 4.569$ ms. We add a safety margin of 0.431 ms to round this up to $\Delta = 5$ ms.

Our chosen underlying consensus protocol, OneShot [11], has three execution paths, each with a different number of steps. Since the timeout must accommodate any of them, we set x to the maximum, which is $x = 8$ under the catch-up execution. Our final timeout settings are shown in Table 2. Again, for LP22 and Lumiere we only have an overall timeout window of Γ while for *Babette* and Broadcast we split the Γ into two components one for a required timeout within a view and one for between views.

Table 2: Required timeout settings for each of our measured protocols using OneShot consensus

Protocol	Γ	
	Intra-view	Inter-view
Babette	$9\Delta = 45$ ms	$3\Delta = 15$ ms
Broadcast	$9\Delta = 45$ ms	N/A
LP22	$12\Delta = 60$ ms	
Lumiere	$20\Delta = 100$ ms	

8 Results

8.1 All Nodes Correct

Because OneShot is streamlined, every consensus decision is immediately followed by a view change, so any view synchronizer we layer on top incurs its cost on every decision rather than only during recovery. This makes the failure-free case a meaningful benchmark: it isolates the steady-state overhead of view synchronization from any adversarial behavior. We compare our four adapted synchronizers against an unsynchronized OneShot baseline, in which nodes advance to the next view immediately after completing the previous one with no synchronization messages exchanged. Each network size is run for 30 views with 50 repetitions, giving 1500 view measurements per data point for both throughput and latency.

Figure 7 shows how the three metrics evolve with network size. Throughput and latency both degrade as n grows, primarily because the quorum sizes required at each step scale with n, and all four synchronizers degrade at broadly similar rates. The notable exception is Lumiere, which outperforms our other synchronizers by a clear margin and, in smaller networks, even outperforms the baseline. This suggests that view synchronization can be beneficial even in the fully cooperative case. Aligning view entry across replicas appears to more than offset the cost of the synchronization messages, likely by preventing the timing skew that arises when each replica advances independently.

The message-count subplot shows the expected split between the two design families. LP22 and Lumiere, which are optimized for message complexity, keep the per-node per-view count essentially flat as n grows. Our two all-to-all designs scale linearly with n, but the leader-based structure of *Babette* pays off clearly. It sends roughly half as many messages per node as the broadcast-based variant at every network size.

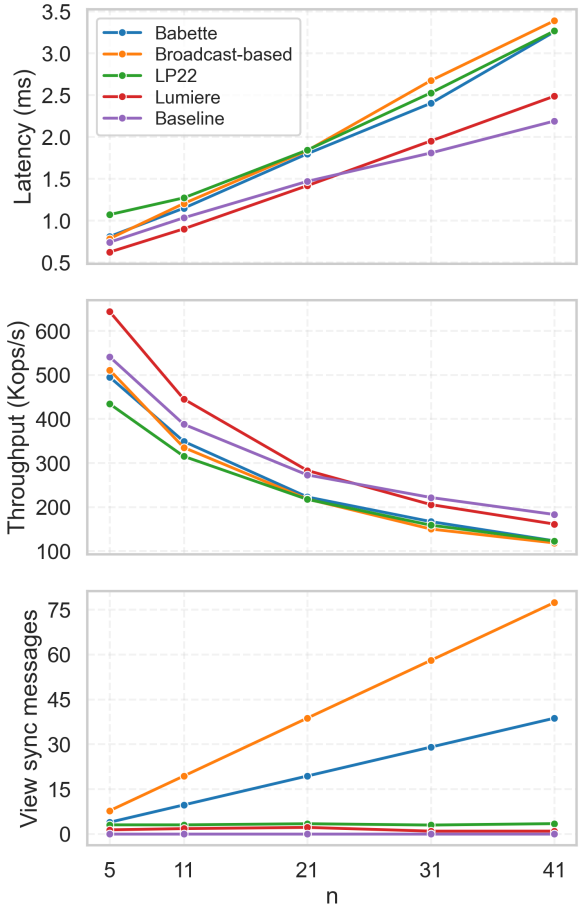


Figure 7: Latency, throughput, and number of view synchronization messages sent per node per view (top to bottom) with regards to the number of nodes in the network n for $f_a = 0$

8.2 A Single Failing Node

The primary purpose of a view synchronizer is to tolerate malicious or unresponsive leaders, so it is essential to measure performance in the presence of such failures. The first scenario we consider is a single unresponsive node, dead from the beginning and not participating in the protocol. Every n^{th} view is therefore expected to fail and timeout, and unlike the failure-free case the number of views run per configuration matters, because the frequency with which the faulty node is elected leader depends on it. We fix the number of failed views to be constant across network sizes and protocols. For each n, we run enough views for the faulty node to be leader exactly 8 times, and stop execution at the next correct view. This gives a total of $v = 8(n-1)+1$ successful views per run at network size n.

Figure 8 shows the resulting throughput, latency, and message counts. *Babette* achieves higher throughput and lower latency than the other synchronizers under the same failure pattern, with the largest advantage at smaller n where a single timed out view represents a much larger fraction of the total views and thus has a larger effect on the latency. Two opposing trends are visible in this plot. As n increases, the proportion of failed views decreases, leading to improved performance. At the same time, performance deteriorates due to increasing quorum sizes—the same effect observed in Figure 7. The faster performance degradation of *Babette*, and to a lesser extent Broadcast, is a direct consequence of the decreasing proportion of failed views. As the proportion of failed views shrinks, the advantage of smaller timeouts diminishes. Beyond $n = 51$ we expect the trend to continue and view synchronization overhead to be dominated by the quorum-size costs already visible at $n = 51$.

The message-count subplot closely mirrors the failure-free case in Figure 7. Per-node per-view counts grow linearly with n for our broadcast-based synchronizer and *Babette*, and remain essentially flat for LP22 and Lumiere. The single failing node adds a small constant overhead from the timeout-triggered synchronization on each failed view, but does not change the asymptotic behavior.

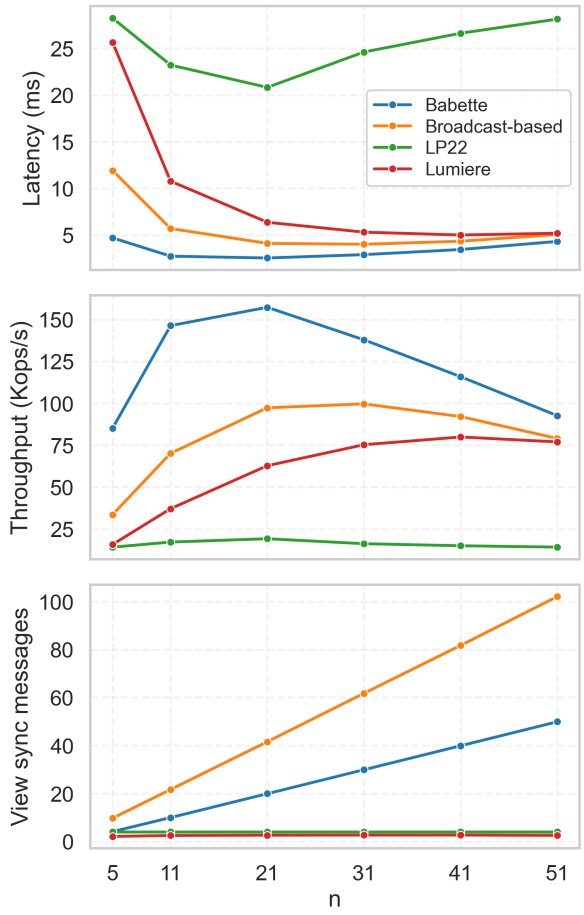


Figure 8: Latency, throughput, and number of view synchronization messages sent per node per view (top to bottom) with regards to the number of nodes in the network n for $f_a = 1$

8.3 Maximum Failing Nodes

Our main goal was to show experimentally that our synchronizers achieve their claimed worst-case latency behavior, and this scenario is the one designed to expose it. All f faulty nodes are dead from the start and positioned consecutively in the leader rotation, which maximizes the number of back-to-back timeouts the synchronizer must absorb. Unlike the single-failure scenario, the fraction of failed views is now held roughly constant as n grows, since f scales with n. Babette's advantage therefore does not dilute with scale in the way it does when the absolute number of failures is held fixed.

Figure 9 shows the resulting metrics. *Babette* maintains a substantial throughput and latency advantage over the other synchronizers across every network size, and this advantage is more pronounced than in the previous scenarios because the timeout wait now scales with f rather than being a one-off cost. All four protocols degrade only slightly as n grows, and they do so at nearly the same rate, because the timeout cost dominates end-to-end performance in this regime and the number of consecutive timeouts is the same across protocols. What differs is the per-timeout cost itself, and this is where our synchronizers, and *Babette* in particular, win. Although all synchronizers incur $O(n\Delta)$ latency asymptotically, our synchronizers achieve smaller constants, and in

this regime those constants translate almost directly into absolute latency and, correspondingly, into higher throughput.

In the message-count subplot the gap between the broadcast-based synchronizer and *Babette* is even larger. In the $f = 0$ and $f = 1$ scenarios we saved messages by not always having to send a *VC*, since some nodes received a *VC* from another node before creating their own. Here, however, all nodes receive the required number of wishes at almost exactly the same time, right after the slowest wish arrives, so nearly every node ends up forming and sending its own *VC*.

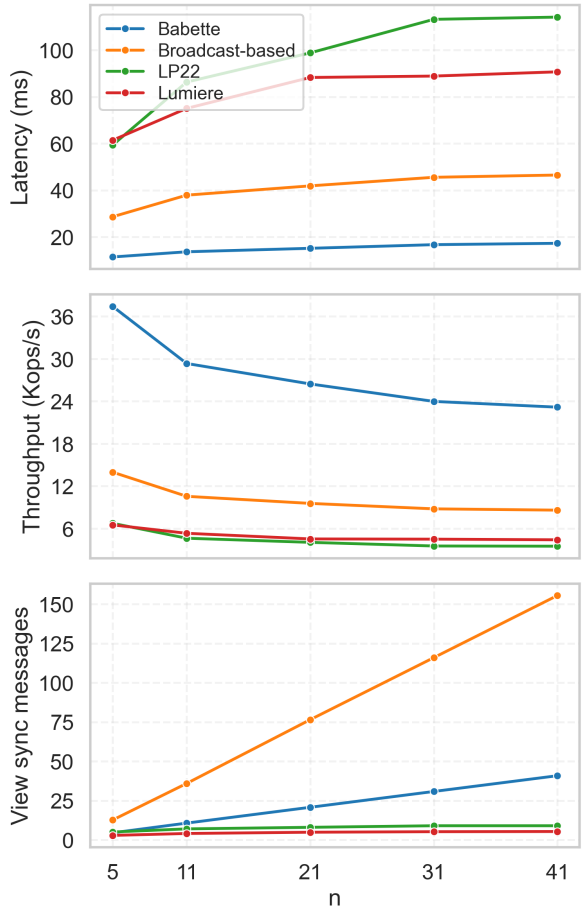


Figure 9: Latency, throughput, and number of view synchronization messages sent per node per view (top to bottom) with regards to the number of nodes in the network n for $f_a = f$

9 Discussion

Our results make it clear that higher required timeouts translate directly into higher latency and lower throughput. Under maximum failures, *Babette* achieves around 5.5× lower latency and correspondingly higher throughput on average than Lumiere across all tested network sizes. Whether Lumiere's performance sacrifice is worth its message-complexity advantage is therefore debatable. In very fast networks where failures are rare, lower message complexity is preferable, as it saves bandwidth and reduces the risk of network congestion. In slower networks where Δ must be larger and failures are semi-frequent, the latency cost of view synchronization outweighs the benefit of sending fewer messages. Of our three scenarios, the relative performance of the protocols shown in Figure 9 is the pattern most likely to hold at deployment scale. Since it is the only scenario in which the failure rate remains constant with n and is non-zero, it is also the only one in which our synchronizers' advantage over other protocols neither dilutes nor amplifies as networks grow, which is the behavior any deployment with a non-vanishing failure rate should exhibit.

9.1 Consequences of findings for $3f + 1$ view synchronization

We evaluated existing BVS protocols and designed our own specifically for the $2f + 1$ setting. The significance of our findings, however, extends beyond reduced-replica BFT. The same principle of timeout durations dominating performance applies to classical $3f + 1$ view synchronization as well, and the specific latency-versus-message-complexity trade-off we observed against Lumiere should carry over largely unchanged.

The most promising $3f + 1$ candidate is Cogsworth [34], which we ruled out for $2f + 1$ because its echo-based catch-up rule relies on the $3f + 1$ quorum structure (as explained in Section 5.2). Under $3f + 1$ this obstacle disappears. If Cogsworth is implemented with the same distinction between intra-view and inter-view timeouts that we adopt for *Babette*, it should benefit from the same tight timeout bounds our results demonstrate. It would also not require the additional *VC* rebroadcast phase that *Babette* needs under $2f + 1$. Cogsworth should therefore beat Lumiere on latency for the same reasons *Babette* does in our experiments, while also improving on *Babette*'s message complexity. *Babette* itself is technically usable under $3f + 1$ by replacing the $f + 1$ quorum threshold with $2f + 1$, but Cogsworth is the more natural choice in that setting.

10 Reflection

10.1 Relation to the Broader CS Field

Recent BFT view synchronization research [27–29] has been driven primarily by the pursuit of asymptotically optimal message complexity, culminating in protocols that match the $\Omega(n^2)$ Dolev–Reischuk lower bound. Our work pushes

against this trend. We show empirically that the constants hidden behind $O(n\Delta)$ latency bounds can dominate practical performance to the point that a message-optimal protocol is substantially slower than a non-message-optimal one under realistic fault conditions. This reflects a wider tension in distributed systems and, more generally, in algorithms research. Asymptotic optimality on one metric does not imply good engineered performance, especially when a second metric (here, latency) is sensitive to the same design choices that minimize the first. We also push against the long-standing tendency of BVS work to rely exclusively on asymptotic analysis without empirical evaluation, which leaves the constant factors that dominate real-world performance invisible to protocol designers and system operators.

Our work also engages with the reduced-replica $2f + 1$ BFT setting, which is part of a broader CS trend of leveraging trusted hardware to relax classical theoretical bounds. The $2f + 1$ literature [10–12, 33] has so far treated view synchronization as a secondary concern, and our work contributes a systematic study of synchronizer design and performance specifically tailored to this setting.

Our findings also carry over, in a more limited form, to modern DAG-based BFT protocols such as Autobahn [21] and hybrid leader-based/leaderless designs such as Morpheus [30]. Although these protocols decouple data dissemination from consensus, their consensus layer remains leader-based and partially synchronous, with view changes still governed by the same conservative timeout constants our work targets. When a leader fails, consensus stalls until the view synchronizer times out and installs a new leader, and larger timeouts extend this stall proportionally. Client requests submitted during the stall are not executed until consensus recovers, so their end-to-end latency is at least as large as the timeout-driven recovery time. Latency-optimal view synchronization directly shortens this recovery time.

10.2 Implications, Limitations, and Stakeholders

The most direct implication of our findings is for the design and deployment of BFT systems in settings where leader failures are non-negligible, such as permissioned blockchains and consortium-operated distributed ledgers. In such systems, view synchronization is on the common path, and our results suggest that the prevailing emphasis on message-complexity optimality may be the wrong default. The relevant stakeholders are protocol designers, who gain an explicit articulation of the latency–message-complexity trade-off that asymptotic analysis tends to obscure, and system operators, who gain concrete guidance: message-optimal synchronizers remain preferable in bandwidth-constrained networks with rare faults, while latency-optimized synchronizers like *Babette* are preferable when Δ is large or faults are frequent.

Our evaluation has limitations that bound the strength of these claims. Experiments run on a single cluster with low and homogeneous inter-node latencies, which does not capture geo-distributed deployments where Δ is realistically larger and the cost of conservative timeouts more severe – if anything, this strengthens our qualitative conclusion but leaves the precise crossover points open. Network size is capped at roughly 50 nodes by DAS-5 availability, limiting claims about very large deployments. Finally, our implementation inherits the practical limitations of Intel SGX, including known side-channel exposures; the strength of the underlying $2f + 1$ assumption depends on the integrity of the TEE in deployment, which is an active area of research in its own right.

11 Use of Generative AI

This section acknowledges how we used generative AI tools during the development of this thesis. We used multiple generative AI tools throughout this work, specifically OpenAI's GPT models, Anthropic's Claude, DeepSeek, and Google's Gemini. The tools were mainly used to support literature research, implementation, and writing.

At the start of the project, they helped with surveying literature and understanding the broader subject of BFT and view synchronization. We used generative AI to clarify concepts and topics that were initially unfamiliar and to summarize potentially relevant papers.

During implementation, they supported our understanding of the existing codebase, as well as debugging, log analysis, and the generation/adaptation of Python scripts for collecting and presenting results. In the writing process, we used them mainly for LaTeX assistance and for improving language, academic style, grammar, and sentence clarity. We also used them at the end to help with summary of the paper and writing the conclusion and abstract.

All generated responses were critically reviewed and verified by us, and full responsibility for the content, correctness, interpretation, and conclusions of this thesis remains entirely our own.

12 Conclusion and Future Work

View synchronization is a performance-critical component of modern streamlined BFT protocols, yet prior work on $2f + 1$ consensus has largely treated it as a secondary concern, and prior work on $3f + 1$ view synchronization has focused on minimizing asymptotic message complexity at the expense of latency. This thesis addressed both gaps. We adapted the message-optimal LP22 and Lumiere synchronizers to the $2f+$

1 setting, designed two new latency-optimized synchronizers tailored to it – a broadcast-based protocol and the leader-based *Babette* – and evaluated all four empirically on the DAS-5 cluster across network sizes up to $n = 51$ and fault scenarios ranging from no failures to the maximum tolerated f.

Our central finding is that the latency cost of conservative timeout settings, which message-optimal protocols require for correctness, dominates the message-cost savings they offer. As shown in our evaluation, *Babette* achieves around 5.5× lower latency and correspondingly higher throughput than Lumiere under maximum failures across all tested network sizes, despite using asymptotically more messages per view. This is also the regime whose relative performance is expected to generalize to deployments with non-vanishing failure rates. The practical recommendation is therefore regime-dependent. Message-optimal synchronizers remain preferable in bandwidth-constrained networks where faults are rare and where the marginal cost of data transmission, monetary or resource-based, is non-negligible, while latency-optimized synchronizers like *Babette* are preferable when failures are expected and recurring. More broadly, our results suggest that asymptotic message-complexity optimality is not the right design target because timeout constants dominate end-to-end performance.

Several directions remain open. The latency cost of message-optimal protocols could be mitigated through dynamic Δ tuning based on observed network conditions, or through leader-rotation schedules that avoid revisiting recently faulty leaders. A hybrid synchronizer that switches between message-optimal and latency-optimal modes based on the current fault rate is a natural next step. Empirical confirmation that our findings also extend to the classical $3f + 1$ setting would strengthen the case that the latency advantage of protocols with tight timeout bounds is not specific to reduced-replica BFT. Finally, larger-scale experiments beyond the ∼50-node DAS-5 limit, ideally on geo-distributed infrastructure where Δ is realistically large, would help establish where the crossover points between synchronizer regimes actually lie in production-like conditions.

References

[1] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer* 49, 5 (May 2016), 54–63. doi:10.1109/MC.2016.127

[2] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (Nov. 1987), 130–143. doi:10.1016/0890-5401(87)90054-X

[3] Ethan Buchman. 2016. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. https://www.semanticscholar.org/paper/Tendermint%3A-Byzantine-Fault-Tolerance-in-the-Age-of-Buchman/07ebd46dc1f020cfbad17c9285ba372c040695d5

[4] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance

[5] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: making adversaries stick to their word. *ACM SIGOPS Operating Systems Review* 41, 6 (Oct. 2007), 189–204. doi:10.1145/1323293.1294280

[6] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. 2022. Byzantine Consensus is Θ(n^2): The Dolev-Reischuk Bound is Tight even in Partial Synchrony! [Extended Version]. doi:10.48550/ARXIV.2208.09262 Version Number: 2.

[7] Allen Clement, Edmund Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. 153–168.

[8] Intel Corporation. 2015. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. See SGX-related chapters; accessed 2026-06-18.

[9] M. Correia, N.F. Neves, and P. Verissimo. 2004. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* 174–183. doi:10.1109/RELDIS.2004.1353018

[10] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. 2022. DAMYSUS: streamlined BFT consensus leveraging trusted components. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3492321.3519568

[11] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. 2024. OneShot: View-Adapting Streamlined BFT Protocols with Trusted Execution Environments. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1022–1033. doi:10.1109/IPDPS57955.2024.00095 ISSN: 1530-2075.

[12] Jerémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. 2026. Pallas and Aegis: Rollback Resilience in TEE-Aided Blockchain Consensus. In *Proceedings 2026 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. doi:10.14722/ndss.2026.242443

[13] The Salticidae Developers. 2018. Salticidae: A Fast Network Library. https://github.com/Determinant/salticidae. GitHub repository, accessed 2026-06-18.

[14] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on information exchange for Byzantine agreement. *J. ACM* 32, 1 (Jan. 1985), 191–204. doi:10.1145/2455.214112

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

[16] Michael J. Fischer, View Profile, Nancy A. Lynch, View Profile, Michael S. Paterson, and View Profile. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (April 1985), 374–382. doi:10.1145/3149.214121

[17] Advanced School for Computing and Imaging (ASCI). 1985. ASCI School. https://asci.school/. Accessed 2026-06-18.

[18] Advanced School for Computing and Imaging (ASCI). 2012. DAS-5: The Distributed ASCI Supercomputer 5. https://www.cs.vu.nl/das5/. Accessed 2026-06-18.

[19] Advanced School for Computing and Imaging (ASCI). 2021. DAS-6: The Distributed ASCI Supercomputer 6. https://www.cs.vu.nl/das6/. Accessed 2026-06-18.

[20] OpenSSL Software Foundation. 1998. OpenSSL. https://www.openssl.org/. Accessed 2026-06-18.

[21] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Autobahn: Seamless high speed BFT. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 1–23. doi:10.1145/3694715.3695942

[22] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: a Scalable and Decentralized Trust Infrastructure. doi:10.48550/arXiv.1804.01626 arXiv:1804.01626 [cs].

[23] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 295–308. doi:10.1145/2168836.2168866

[24] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems* 27, 4 (Dec. 2009), 1–39. doi:10.1145/1658357.1658358

[25] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. doi:10.1145/357172.357176

[26] Leslie Lamport and View Profile. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. doi:10.1145/359545.359563

[27] Andrew Lewis-Pye. 2022. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model. doi:10.48550/arXiv.2201.01107 arXiv:2201.01107 [cs].

[28] Andrew Lewis-Pye and Ittai Abraham. 2023. Fever: Optimal Responsive View Synchronisation. doi:10.48550/arXiv.2301.09881 arXiv:2301.09881 [cs].

[29] Andrew Lewis-Pye, Dahlia Malkhi, Oded Naor, and Kartik Nayak. 2024. Lumiere: Making Optimal BFT for Partial Synchrony Practical. doi:10.48550/arXiv.2311.08091 arXiv:2311.08091 [cs].

[30] Andrew Lewis-Pye and Ehud Shapiro. 2026. Morpheus Consensus: Excelling on trails and autobahns. *LIPIcs, Volume 361, OPODIS 2025* 361 (2026), 35:1–35:20. doi:10.4230/LIPIcs.OPODIS.2025.35 arXiv:2502.08465 [cs].

[31] Daniel Lihotsky. 2026. *Archived experimental log data underlying figures in "Latency-Optimal View Synchronization for 2f+1 BFT Consensus in the Presence of Faults".* doi:10.5281/zenodo.21225679

[32] Daniel Lihotsky. 2026. View Synchronizers: Implementations of LP22, Lumiere, Broadcast-based, and Babette for OneShot. https://github.com/danielliho/view-synchronizers. GitHub repository.

[33] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. 2019. Scalable Byzantine Consensus via Hardware-assisted Secret Sharing. *IEEE Trans. Comput.* 68, 1 (Jan. 2019), 139–151. doi:10.1109/TC.2018.2860009 arXiv:1612.04997 [cs].

[34] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2020. Cogsworth: Byzantine View Synchronization. doi:10.48550/arXiv.1909.05204 arXiv:1909.05204 [cs].

[35] Oded Naor and Idit Keidar. 2020. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR. doi:10.48550/arXiv.2002.07539 arXiv:2002.07539 [cs].

[36] M. Pease, View Profile, R. Shostak, View Profile, L. Lamport, and View Profile. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (April 1980), 228–234. doi:10.1145/322186.322188

[37] Vincent Rahli. 2022. Damysus: Streamlined BFT consensus leveraging trusted components (code repository). https://github.com/vrahli/damysus. GitHub repository, accessed 2026-06-18.

[38] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *Comput. Surveys* 22, 4 (Dec. 1990), 299–319. doi:10.1145/98163.98167

[39] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (Jan. 2013), 16–30. doi:10.1109/TC.2011.221

[40] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. doi:10.1145/3293611.3331591

A Broadcast-based View Synchronizer Pseudocode

Algorithm 2 Instructions for node a

upon completion of view $v-1$ OR timeout for $v-1$:
  broadcast $WISH_v$

upon receiving $WISH_v$:
  if $f+1$ distinct and verified $WISH_v$ received:
    if $v >$ current view:
      start view v
    form VC_v
    broadcast VC_v

upon receiving VC_v from b:
  if VC_v signatures valid AND $v >$ current view:
    start view v
    send VC_v to **all** except b

B Proof of *Babette* achieving view synchronization

Claim 1. *The* Babette *algorithm eventually brings all honest nodes into the same view for a sufficiently long time for consensus to resume.*

Proof. Let honest node a enter view v with a correct leader at time t after GST (Given our assumptions this scenario has to occur eventually). If a moved to v by receiving $f+1$ distinct and correctly signed $WISH_v$ messages, it formed and set a correct VC_v, which reaches all honest nodes within $t+\Delta$, which also move to view v. If a moved to v by receiving a correct VC_v from b, it resends the VC_a to all other nodes which reaches all honest nodes by $t+\Delta$. Therefore all honest nodes enter view v maximum Δ apart. Since nodes timeout and wish to enter $v+1$ only after it is certain consensus had enough time to complete and account for the delay of entry, nodes must have been in view v sufficiently long enough for consensus to complete. □

Note: The same proof is valid for our Broadcast-based algorithm.

C Full Results

All measurements were rounded to three decimal points with standard deviation reported in parentheses next to the mean. The full experimental logs and raw results are publicly available on Zenodo [31].

C.1 All Nodes Correct

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	494.954(9.555)	0.808(0.016)	6.532(0.14)	3.867(0)	1.746(0.044)	0.198(0.013)	29(0)	87(0)
Broadcast	510.942(10.776)	0.783(0.017)	6.387(0.217)	7.733(0.002)	1.646(0.054)	0.206(0.016)	29(0)	87.024(0.097)
LP22	434.004(21.459)	1.07(1.076)	7.764(0.207)	3.004(0.028)	2.448(0.078)	0.256(0.017)	37.908(0.456)	113.74(1.373)
Lumiere	645.099(21.884)	0.621(0.021)	3.917(0.201)	1.586(0.022)	1.001(0.043)	0.07(0.005)	13.896(0.163)	18(0)
Baseline	540.857(21.162)	0.741(0.029)	4.468(0.207)	0(0)	0(0)	0(0)	0(0)	0(0)

Table 3: $n = 5$, $f_a = 0$, **views** = 30, **repetitions** = 50

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	348.695(4.537)	1.147(0.015)	6.479(0.172)	9.67(0.025)	1.601(0.025)	0.19(0.007)	28.989(0.077)	174(0)
Broadcast	334.862(8.279)	1.203(0.074)	6.926(0.216)	19.335(0.018)	1.62(0.042)	0.307(0.022)	28.992(0.053)	174.089(0.197)
LP22	314.866(6.247)	1.271(0.025)	8.65(0.229)	3(0)	2.353(0.046)	0.356(0.017)	33(0)	198(0)
Lumiere	444.721(17.76)	0.901(0.034)	4.647(0.223)	2.023(0.026)	1.08(0.043)	0.062(0.005)	13.887(0.079)	18.556(0.077)
Baseline	387.624(9.07)	1.032(0.024)	5.115(0.179)	0(0)	0(0)	0(0)	0(0)	0(0)

Table 4: $n = 11$, $f_a = 0$, **views** = 30, **repetitions** = 50

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	222.177(3.056)	1.801(0.025)	10.778(0.342)	19.333(0)	2.24(0.051)	0.467(0.019)	29(0)	319(0)
Broadcast	217.911(4.593)	1.836(0.039)	8.518(0.368)	38.694(0.026)	1.671(0.04)	0.518(0.022)	29(0)	319.471(0.459)
LP22	217.127(2.372)	1.842(0.02)	10.906(0.324)	3.381(0)	2.463(0.033)	0.53(0.018)	31(0)	341(0)
Lumiere	282.68(9.704)	1.417(0.049)	6.238(0.385)	2.149(0.021)	1.134(0.01)	0.05(0.004)	13.986(0.032)	17.862(0.166)
Baseline	272.576(5.09)	1.468(0.028)	6.368(0.458)	0(0)	0(0)	0(0)	0(0)	0(0)

Table 5: $n = 21$, $f_a = 0$, **views** = 30, **repetitions** = 10

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	166.575(2.427)	2.402(0.033)	7.484(0.072)	29.001(0.006)	1.581(0.02)	0.277(0.014)	28.999(0.006)	464(0)
Broadcast	149.697(2.628)	2.673(0.045)	7.197(0.092)	58(0)	1.531(0.022)	0.611(0.022)	29(0)	464.038(0.102)
LP22	158.404(3.088)	2.526(0.047)	11.736(0.519)	2.935(0)	2.376(0.089)	0.607(0.052)	30(0)	480(0)
Lumiere	205.211(6.921)	1.951(0.063)	7.032(0.404)	0.903(0)	1.127(0.027)	0.022(0.002)	14(0)	7.226(0)
Baseline	221.427(2.828)	1.807(0.023)	7.053(0.101)	0(0)	0(0)	0(0)	0(0)	0(0)

Table 6: $n = 31$, $f_a = 0$, **views** = 30, **repetitions** = 50

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	122.829(1.332)	3.257(0.035)	13.043(0.539)	38.667(0)	2.103(0.028)	0.82(0.024)	29(0)	609(0)
Broadcast	117.861(1.468)	3.394(0.042)	9.716(0.834)	77.344(0.016)	1.837(0.022)	1.016(0.022)	29(0)	609.171(0.256)
LP22	122.473(1.678)	3.267(0.045)	13.547(0.727)	3.415(0)	2.399(0.022)	0.75(0.017)	30(0)	630(0)
Lumiere	160.927(3.783)	2.487(0.058)	8.151(0.666)	0.911(0)	1.157(0.02)	0.019(0.001)	14(0)	7.171(0)
Baseline	182.707(1.32)	2.189(0.016)	7.843(0.475)	0(0)	0(0)	0(0)	0(0)	0(0)

Table 7: $n = 41$, $f_a = 0$, **views** = 30, **repetitions** = 10

C.2 A Single Failing Node

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	85.114(0.562)	4.7(0.031)	9.298(0.356)	4.119(0.004)	3.07(0.079)	0.296(0.02)	39.9(0.125)	96(0)
Broadcast	33.543(0.078)	11.925(0.028)	8.83(0.361)	9.7(0.01)	2.806(0.126)	0.382(0.031)	40(0)	120.09(0.33)
LP22	14.159(0.016)	28.251(0.032)	10.372(0.321)	4(0)	3.645(0.106)	0.339(0.017)	52.97(0.083)	135(0)
Lumiere	16.043(0.012)	24.933(0.018)	4.896(0.135)	2.121(0.013)	1.414(0.051)	0.101(0.008)	19.938(0.111)	26.25(0)

Table 8: $n = 5$, $f_a = 1$, **views** = 33, **repetitions** = 25

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	147.36(1.783)	2.715(0.032)	22.651(1.046)	9.975(0)	6.221(0.252)	0.859(0.068)	87.938(0.065)	480(0)
Broadcast	70.161(0.231)	5.701(0.019)	20.81(0.44)	21.665(0.067)	5.317(0.162)	1.077(0.066)	87.392(0.531)	528.625(0.625)
LP22	17.225(0.011)	23.222(0.015)	24.804(0.309)	4(0)	7.168(0.141)	0.927(0.023)	101.984(0.037)	564(0)
Lumiere	37.12(0.054)	10.776(0.016)	12.749(0.214)	2.519(0.02)	3.246(0.065)	0.181(0.01)	43.732(0.163)	61.92(0.245)

Table 9: $n = 11$, $f_a = 1$, **views** = 81, **repetitions** = 25

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	157.266(2.878)	2.544(0.044)	56.889(3.037)	19.925(0)	12.416(0.472)	2.452(0.277)	167.95(0.043)	1760(0)
Broadcast	97.316(0.756)	4.111(0.031)	45.811(1.051)	41.577(0.039)	9.041(0.194)	2.892(0.126)	166.492(0.234)	1850.376(1.89)
LP22	19.263(0.026)	20.765(0.028)	58.762(2.068)	4(0)	13.908(0.367)	2.819(0.227)	182.978(0.033)	1925(0)
Lumiere	62.782(0.487)	6.372(0.048)	32.67(1.882)	2.64(0.008)	6.259(0.312)	0.272(0.014)	83.934(0.066)	117.414(0.553)

Table 10: $n = 21$, $f_a = 1$, **views** = 161, **repetitions** = 25

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	138.427(0.898)	2.89(0.019)	89.434(0.99)	29.909(0)	18.226(0.235)	5.44(0.11)	247.99(0.021)	3840(0)
Broadcast	99.648(0.428)	4.014(0.017)	69.347(0.688)	61.758(0.043)	15.131(0.105)	6.618(0.172)	247.543(0.138)	3977.556(5.542)
LP22	16.25(0.018)	24.615(0.027)	101.902(4.75)	4(0)	20.438(0.721)	5.156(0.446)	262.986(0.019)	4080(0)
Lumiere	75.298(0.369)	5.312(0.024)	58.744(0.551)	2.677(0.004)	9.346(0.092)	0.378(0.01)	123.968(0.031)	172.309(0.149)

Table 11: $n = 31$, $f_a = 1$, **views** = 241, **repetitions** = 25

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	115.859(0.78)	3.453(0.023)	121.87(0.957)	39.902(0.002)	21.671(0.174)	9.031(0.203)	327.974(0.019)	6720(0)
Broadcast	92.086(0.502)	4.344(0.024)	96.602(0.766)	81.715(0.046)	18.604(0.17)	10.047(0.214)	326.856(0.227)	6902.89(5.647)
LP22	15.02(0.011)	26.631(0.019)	133.496(4.844)	4(0)	25.895(0.936)	8.134(0.503)	342.978(0.026)	7035(0)
Lumiere	79.902(0.634)	5.006(0.039)	77.904(3.634)	2.686(0.014)	12.512(0.473)	0.503(0.024)	163.9(0.116)	227.451(0.314)

Table 12: $n = 41$, $f_a = 1$, **views** = 321, **repetitions** = 25

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	92.603(0.493)	4.32(0.023)	169.296(2.104)	49.902(0.005)	27.149(0.476)	13.316(0.245)	407.944(0.041)	10400(0)
Broadcast	79.009(0.979)	5.063(0.063)	133.984(3.368)	102.073(0.164)	23.26(0.519)	14.524(0.807)	406.616(0.869)	10661.144(13.477)
LP22	14.212(0.007)	28.145(0.014)	185.269(6.351)	4(0)	32.134(1.07)	11.263(0.629)	422.974(0.049)	10790(0)
Lumiere	76.972(0.422)	5.197(0.028)	105.091(0.591)	2.563(0.032)	15.273(0.075)	0.548(0.008)	202.832(0.254)	282.984(0.478)

Table 13: $n = 51$, $f_a = 1$, **views** = 401, **repetitions** = 10

C.3 Maximum Failing Nodes

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	37.432(5.482)	11.405(5.114)	10.353(0.427)	4.485(0)	4.4(0.143)	0.326(0.019)	52(0)	96(0)
Broadcast	13.979(0.02)	28.614(0.04)	11.347(0.359)	12.612(0.02)	4.236(0.139)	0.571(0.029)	52.027(0.133)	156.08(0.4)
LP22	6.746(0.005)	59.295(0.04)	12.13(0.403)	5(0)	5.016(0.179)	0.434(0.026)	69(0)	147(0)
Lumiere	6.518(0.002)	61.372(0.022)	5.604(0.251)	2.848(0)	1.867(0.087)	0.141(0.011)	26(0)	36(0)

Table 14: $n = 5$, $f_a = 2$, **views** = 33, **repetitions** = 25

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	29.346(0.055)	13.631(0.026)	26.158(0.4)	10.679(0)	12.166(0.246)	1.017(0.033)	145(0)	480(0)
Broadcast	10.561(0.009)	37.876(0.033)	29.381(0.904)	35.924(0.06)	10.611(0.294)	2.227(0.155)	145.927(0.472)	870.4(1)
LP22	4.636(0.001)	86.289(0.016)	28.721(0.472)	7(0)	12.512(0.23)	1.199(0.048)	169(0)	624(0)
Lumiere	5.333(0.002)	75.012(0.026)	13.919(0.3)	4.074(0)	5.053(0.138)	0.296(0.022)	70(0)	100(0)

Table 15: $n = 11$, $f_a = 5$, **views** = 81, **repetitions** = 25

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	26.452(0.163)	15.122(0.092)	61.444(4.008)	20.745(0)	25.88(1.606)	2.545(0.359)	300(0)	1760(0)
Broadcast	9.558(0.009)	41.848(0.041)	78.038(2.855)	76.51(0.09)	21.099(0.628)	6.917(0.331)	300.975(0.419)	3307.04(6.477)
LP22	4.049(0.001)	98.784(0.014)	65.051(0.916)	8(0)	25.402(0.352)	3.066(0.123)	327(0)	2057(0)
Lumiere	4.533(0.001)	88.244(0.023)	33.901(0.734)	4.875(0)	11.115(0.274)	0.525(0.032)	150(0)	217(0)

Table 16: $n = 21$, $f_a = 10$, **views** = 161, **repetitions** = 25

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	23.981(0.023)	16.68(0.016)	98.291(1.013)	30.809(0)	41.712(0.253)	5.646(0.191)	465(0)	3840(0)
Broadcast	8.788(0.015)	45.518(0.078)	114.549(5.003)	115.976(0.077)	35.632(1.645)	12.697(0.657)	466.188(0.344)	7448.04(9.37)
LP22	3.536(0.001)	113.134(0.019)	101.483(2.178)	9(0)	37.978(0.714)	6.15(0.208)	494(0)	4304(0)
Lumiere	4.502(0.001)	88.848(0.025)	53.989(1.861)	5.211(0)	16.52(0.449)	0.665(0.024)	225(0)	337.04(0.2)

Table 17: $n = 31$, $f_a = 15$, **views** = 241, **repetitions** = 25

Algorithm	Throughput (Kops/s)	Latency (ms)	Handle (ms)	View sync. mes. (#)	Sign dur. (ms)	Verif. dur. (ms)	Sign num. (#)	Verif. num. (#)
Babette	23.168(0.082)	17.265(0.06)	152.321(6.799)	40.831(0)	60.384(3.162)	9.821(0.833)	700(0)	7560(0)
Broadcast	8.605(0.008)	46.485(0.044)	181.261(2.747)	155.491(0.076)	53.232(0.773)	25.138(0.82)	703.07(0.513)	14705.08(8.817)
LP22	3.507(0.054)	114.088(1.89)	160.122(3.002)	9(0)	57.3(0.838)	9.757(0.316)	733(0)	8253.36(1.8)
Lumiere	4.412(0.001)	90.659(0.025)	90.417(2.47)	5.328(0)	24.956(0.401)	1.008(0.034)	340(0)	508(0)

Table 18: $n = 41$, $f_a = 20$, **views** = 361, **repetitions** = 25