# OptiLog: Assigning Roles in Byzantine Consensus

**Hanish Gogada**
University of Stavanger
Norway
hanish.gogada@uis.no

**Christian Berger**
Friedrich-Alexander-Universität
Erlangen-Nürnberg
Germany
christian.g.berger@fau.de

**Leander Jehl**
University of Stavanger
Norway
leander.jehl@uis.no

**Hans P. Reiser**
Reykjavik University
Iceland

**Hein Meling**
University of Stavanger
Norway

## Abstract

Byzantine Fault-Tolerant (BFT) protocols play an important role in blockchains. As the deployment of such systems extends to wide-area networks, the scalability of BFT protocols becomes a critical concern. Optimizations that assign specific roles to individual replicas can significantly improve the performance of BFT systems. However, such role assignment is highly sensitive to faults, potentially undermining the optimizations' effectiveness.

To address these challenges, we present OptiLog, a logging framework for collecting and analyzing measurements that help to assign roles in globally distributed systems, despite the presence of faults. OptiLog presents local measurements in global data structures, to enable consistent decisions and hold replicas accountable if they do not perform according to their reported measurements.

We demonstrate OptiLog's flexibility by applying it to two BFT protocols: (1) Aware, a highly optimized PBFT-like protocol, and (2) Kauri, a tree-based protocol designed for large-scale deployments. OptiLog detects and excludes replicas that misbehave during consensus and thus enables the system to operate in an optimized, low-latency configuration, even under adverse conditions. Experiments show that for tree overlays deployed across 73 worldwide cities, trees found by OptiLog display 39% lower latency than Kauri.

**CCS Concepts:** • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **General and reference** → *Performance*; • **Computing methodologies** → Distributed algorithms.

*Keywords:* Byzantine Fault Tolerance, Consensus, Reliability, Performance Optimization, Adaptive Systems, Reconfiguration Framework, Low Latency, Fault Detection

## 1 Introduction

Byzantine Fault-Tolerant (BFT) consensus protocols enable distributed systems that can tolerate arbitrary failures, making them a critical component for blockchain systems that distribute trust among participants that validate transactions and maintain the ledger. Therefore, to reduce reliance on a few participants and bolster security, scaling BFT protocols beyond their traditional capacity [19, 45] has become crucial.

Numerous studies have tackled the scalability challenges using various optimization strategies effective under specific favorable conditions [18, 29, 31, 51, 58]. However, in adverse scenarios, these optimizations become less effective or even defunct, necessitating a switch [9, 11] to a more resilient but less effective protocol, like PBFT [19], to make progress. It is, however, non-trivial to detect whether or not the operating conditions are favorable or adversarial. Existing work utilizes inefficient techniques based on randomness [29, 47, 51] or predefined assignments [58], requiring repeated trial-and-error to find a working system configuration. These systems often ignore the actual operating conditions, such as the latency between replicas and prior misbehavior, resulting in poor performance. Many systems [12, 16, 26, 49, 53] that consider the operating conditions rely only on local measurements to select a system configuration. This is problematic because local measurements taken at different replicas may be inconsistent, making it difficult to derive a global configuration. Moreover, delegating configuration decisions to a single replica, e.g., the leader, based on its own measurements, erodes accountability because other replicas cannot verify that the decision was based on actual measurements.

This paper advocates for a holistic, measurement-based approach to accurately identify the current operating conditions, promoting aggressive use of efficient protocols and reducing fallback to less efficient ones. We accomplish this through a shared append-only log of measurements.
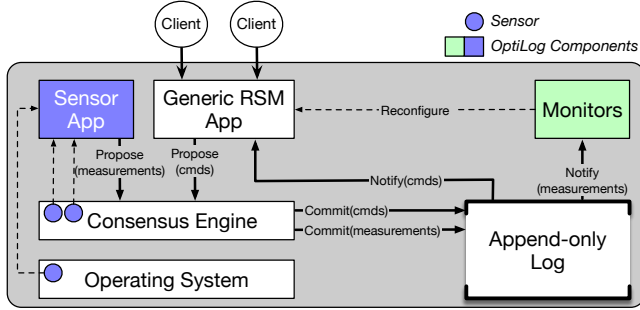
**Figure 1.** OptiLog's component architecture.

We describe OptiLog, an integrated shared log that records measurements and derives metrics for system configuration. OptiLog extends a generic Replicated State Machine (RSM) with sensors to capture *measurements* and monitors for evaluating *metrics*. Replicas instrumented with sensors append measurements to the log, while corresponding monitors collate these measurements to derive *efficient configurations*. OptiLog enables replicas to make consistent configuration decisions based on the same information. The log also provides accountability, allowing all replicas to verify decisions and recognize faulty behaviors. Fig. 1 illustrates OptiLog's architecture; see §4 for a detailed description.

Some optimization techniques can be costly to (deterministically) evaluate for large configuration sizes. Thus, OptiLog allows heuristic optimization techniques to be used, where the potentially non-deterministic search outcomes are logged, enabling replicas to consistently derive a global ranking of configurations. OptiLog also supports collaborative optimization techniques, where the search space is partitioned and distributed across the RSM replicas.

Our second contribution is a pipeline of sensors and monitors for selecting functional, low-latency configurations. Our pipeline allows replicas to predict a configuration's latency based on individual link latencies, and to use simulated annealing [40] to search for low-latency configurations. The main novelty of our pipeline is a suspicion mechanism that detects timing failures and takes suspicions into account during configuration search. In particular, our suspicion mechanism detects replicas that underperform relative to their self-reported latencies. This allows OptiLog to find efficient configurations even in the presence of performance attacks [7, 21].

To showcase OptiLog, we integrate our pipeline into two BFT protocols: Aware [13] and Kauri [51]. Aware and Kauri are based on PBFT [19] and HotStuff [63], respectively, and represent optimizations of different classes of BFT protocols, each posing distinct challenges. This demonstrates the flexibility and generality of OptiLog.

**OptiAware.** Aware is a modern variant of PBFT and assumes a clique topology with multiple all-to-all communication phases, where replicas respond after collecting messages

from a quorum. Each replica assigns weights to other replicas based on the measured latency between them. The all-to-all message pattern allows any configuration with a functioning leader to remain operational, despite backup failures. In PBFT however, distinguishing benign from intentional delays is hard, making it tricky to set a detection threshold that avoids false positives but still catches real delays. While Aware improves on PBFT's latency, it remains vulnerable to performance attacks. Our third contribution, OptiAware, applies OptiLog's pipeline to Aware, yielding a system that can detect the culprits of such attacks and mitigate their impact by excluding them from consideration when assigning high voting weights and the leader role.

**OptiTree.** For our fourth contribution, we focus on BFT systems [42, 43, 51] that organize their RSM replicas in a tree topology for improved scalability. A tree topology reduces communication overhead by limiting interactions to parent and child replicas. However, this structure is vulnerable to failures, which can fragment the network. To counter this, tree-based protocols require fallback and reconfiguration mechanisms. Kauri [51], for example, builds multiple randomized trees to prevent targeted attacks, switching to a new tree when failures occur. However, this approach can cause considerable delays in reaching consensus due to the number of reconfigurations needed to find a working tree. Moreover, blindly constructing trees based on randomness can result in latency-imbalanced trees that degrade the RSM's performance. Ideally, trees should be constructed with reliable replicas at the core and less reliable ones at the leaves.

OptiTree applies OptiLog's sensor and monitor pipeline to find working and performant trees. We implemented two variants of OptiTree and compared their throughput under various latency conditions. Our experiments show that trees selected by OptiTree outperform Kauri's throughput by up to 2.5× and HotStuff by 2.9×. Further, OptiTree can reconfigure the RSM until a working tree is found, while recording the measurements in the log. Our approach guarantees to find a working tree within at most $2f$ reconfigurations for certain tree configurations, where $f$ is the maximum number of faulty replicas.

In summary, the main contributions of this paper are:

1. OptiLog, a logging framework for systematic measurement collection and configuration optimization for RSMs.
2. OptiLog enables RSM optimizations to compute functional low-latency configurations in the presence of faults.
3. OptiAware uses OptiLog to detect-and-mitigate performance attacks and maintain optimal latency.
4. OptiTree uses OptiLog to generate low-latency working trees with linear reconfigurations.
5. Trees selected by OptiTree boost Kauri's throughput by 67.5% in a simulated Stellar network [5].

## 2 System Model

We consider an RSM that accepts client commands as input and produces output to clients. The RSM may record certain events in an append-only log. We assume an RSM with $n \geq 3f + 1$ replicas, where up to $f$ can be Byzantine faulty, meaning they may exhibit arbitrary, potentially adversarial behavior. Replicas that faithfully follow the protocol and do not crash are considered *correct*, while others are deemed *faulty*. In every view, at least a quorum of $q = n - f$ correct replicas is assumed to be available.

We assume a *partially synchronous* network [24], where periods of instability may cause unpredictable delays. However, after a *global stabilization time* (GST), latencies become bounded. We also assume there exists a known parameter $\delta$ and an *actual* latency $L_a(A, B)$ between correct replicas $A$ and $B$, such that after GST, the round-trip time of any message between $A$ and $B$ is in the interval $[L_a(A, B), \delta \cdot L_a(A, B)]$.

Note that the partial synchrony assumption only pertains to liveness; the RSM's safety is always preserved. A faulty replica may attempt to disrupt the RSM's performance by recording incorrect measurements in the log. However, an adversary cannot delay messages, including measurements, between correct replicas. Thus, faulty replicas cannot interfere with measurement exchanges between correct replicas.

Let $\Pi$ denote the set of $n$ replicas running the RSM. A *configuration* is an assignment of roles to replicas, which may also encode topology information, e.g., a tree configuration. While describing a replica's position in a tree/graph, it may also be referred to as a *node*. An RSM may *reconfigure* from one configuration to another, e.g., to activate a better configuration or recover from a faulty configuration.

## 3 Motivation

> *If you can't measure it, you can't improve it.*
> —Lord Kelvin

This section outlines several challenges with existing systems that motivate OptiLog's design.

**Performance tuning.** Optimizing the performance of RSM-based protocols necessitates precise tuning to the conditions of their deployment environment. Performance parameters—such as the replication degree, batch size, concurrency, shard count, and network timeouts—must be carefully calibrated to balance efficiency and reliability. For instance, RSMs operating over wide-area networks with inherently higher and less predictable latencies require more conservative timeouts compared to those on local area networks [27, 60]. For leader-based protocols, choosing a short timeout may lead to consecutive leader changes [47], while a longer timeout results in slower recovery from failures. A single leader often becomes a performance bottleneck [41, 59], and rotating the leader-role [63] can result in significant performance degradation when some replicas are faulty [30, 47]. To the best of

our knowledge, real-world RSM deployments lack real-time feedback mechanisms to optimize their configuration.

**Limitations of local-only measurements.** Local measurements [12, 49, 53], where individual replicas capture and use measurements without sharing them with other replicas, suffer from inherent limitations. Such local-only observations can lead to discrepancies, as replicas may draw inconsistent conclusions about their operating environment. Furthermore, replicas cannot be held accountable for configuration decisions based on local measurements, since such decisions cannot be verified by other replicas.

**Measurement-based role assignment.** A wide range of strategies exists for RSM optimization [15], including pipelining [31, 34], committee selection [33, 37, 64], and tree-based communication topologies [42, 51]. These optimizations rely on special roles, like coordinators [34], collectors [31], committee members [64] and internal nodes [51]. These roles are assigned in a randomized [29, 51] or predefined fashion [8, 48, 58], requiring trial-and-error to find a working assignment. This precludes informed decisions and learning from past configuration failures.

Performing end-to-end measurements to evaluate and select configurations [25] becomes impractical as the configuration search space grows. For example, when assigning voting weights [57] or positions in a tree overlay [51], the number of possible configurations is exponential in the number of replicas, making it infeasible to perform end-to-end measurements for all configurations.

Aware [13] selects low-latency configurations based on per-link latency measurements and a scoring function. While this approach is applicable to large configuration search spaces, Aware lacks a mechanism to hold misbehaving replicas accountable when configurations do not perform as expected. In particular, if probe messages exhibit different latencies than actual protocol messages, Aware is unable to select a suitable configuration, as we show in §7.1.

Some systems [21, 25] use measurements to monitor configuration performance, while others rely on observations of past failures [61, 68]. However, these systems focus on leader selection and hold only the leader accountable for failed or degraded configurations. This limitation makes them unsuitable for complex configurations, such as tree topologies [51], which may fail despite a correct leader.

**Collaborative optimization.** Optimizing RSM configurations by exploring the search space on a single replica creates a performance bottleneck. Throughput can be improved by partitioning the search space and distributing the partitions across replicas via scatter-gather [54] or map-reduce [22, 67] techniques. Yet, these distribution techniques remain surprisingly underutilized for optimizing RSM configurations.

**Non-deterministic configuration optimization.** Finding an effective configuration can be framed as a combinatorial optimization problem. Due to the vastness of possible

configurations, finding an optimal solution is often unfeasible. This makes heuristic approaches, including machine learning, more viable than deterministic methods. However, due to the inherent non-determinism of these heuristic solutions, existing systems [13] do not support non-deterministic configuration optimization.

## 4 OptiLog Architecture Overview

This section gives an overview of OptiLog's architecture, as shown in Fig. 1. OptiLog provides measurement and monitoring capabilities to optimize the performance and scalability of generic RSM applications. The RSM uses a *consensus engine* to consistently replicate client commands to an *append-only log*. Optimizing the RSM requires metrics on both its state and environment. To achieve this, OptiLog augments each replica with *sensors* and *monitors*. These components use the log to share information about the system's configuration and sensor measurements with all replicas.

Sensors collect various measurements and pass them to the *sensor app*, which uses the consensus engine to disseminate authenticated measurements to all replicas. These measurements can come from different sources, such as the replica's operating environment or sensors integrated into the consensus engine. The monitors act as counterparts to sensors, receiving notifications about specific measurements and use them to compute various metrics.

Fig. 2 shows how monitor $M_1$ ($M_2$) collect measurements from sensor $S_1$ ($S_2$) at replicas $A$, $B$, $C$, and $D$. This allows OptiLog replicas to make operational decisions based on deterministic information rather than local-only sensor data. As shown in Fig. 2, monitor $M_2$ makes reconfiguration decisions for the RSM based on globally recorded and agreed-upon measurements. For example, one of our monitors generates a latency matrix that provides insights into the latencies between all participating replicas, as discussed in §4.2.1.

### 4.1 Sensors and Monitors

This section presents OptiLog's mechanisms for collecting and processing measurements, namely sensors and monitors. Each sensor-monitor pair has a one-to-one relationship, as shown with $S_1$–$M_1$ and $S_2$–$M_2$ in Fig. 2.

A *sensor* abstracts the capture of local-only measurements related to a replica's operational context. Sensors can be integrated into system components, such as the consensus engine, or can query the operating system for metrics such as CPU load. A sensor can also use information from local monitors, as exemplified by the dashed arrow from $M_1$ to $S_1$ in Fig. 2. A sensor's output is assumed to be non-deterministic and may vary across replicas. Sensors may also perform non-deterministic computations, e.g., to support heuristic and distributed optimization algorithms. We refer to both sensor output and compute results as *measurements*. These measurements are recorded in the log, providing a consistent record
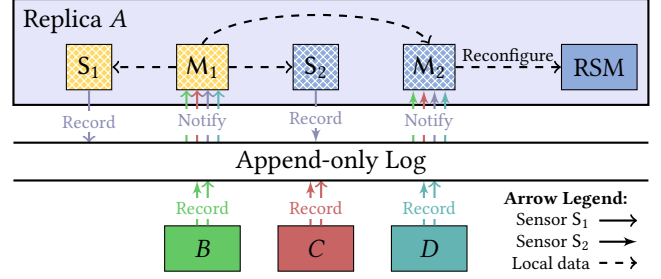


**Figure 2.** Replicas ($A$,$B$,$C$,$D$) with sensors and monitors.

of the system's operation. The recorded measurements are later accessed by monitors, as discussed below.

In a Byzantine environment, sensors at up to $f$ faulty replicas may report incorrect measurements. Consequently, any monitor using these measurements must account for potential inaccuracies. Yet retaining a record of incorrect measurements can be invaluable for forensic analysis.

The *monitor* is the core abstraction linking sensor measurements from individual replicas with system-wide operations, enabling dynamic adjustments to the RSM. When a sensor records a measurement, the corresponding monitor updates its data structures through the following steps:

1. **Data Collection:** The monitor collects measurements from its associated sensor via the log; it may also use data from other local monitors.
2. **Deterministic Computation:** The monitor processes consistent measurements by performing a deterministic computation, producing consistent output; referred to as metrics.

Since monitors on each replica operate on the same ordered set of measurements, they maintain consistent data structures across the system. This consistency provides a consolidated, global view of the measurements, allowing replicas to coordinate system-wide operations reliably, such as activating a new tree configuration with OptiTree (§6). Table 1 summarizes the key properties of sensors and monitors.

**Table 1.** Summary of sensor and monitor properties.

| Property | Sensors | Monitors |
|---|---|---|
| **Input** | System & local monitors | Log & local monitors |
| **Computation** | Non-deterministic | Deterministic |
| **Output** | Variable across replicas | Consistent across replicas |

In the following sections we present the sensors and monitors we implemented for OptiLog, and Fig. 3 shows how these interact to ensure robust configurations.
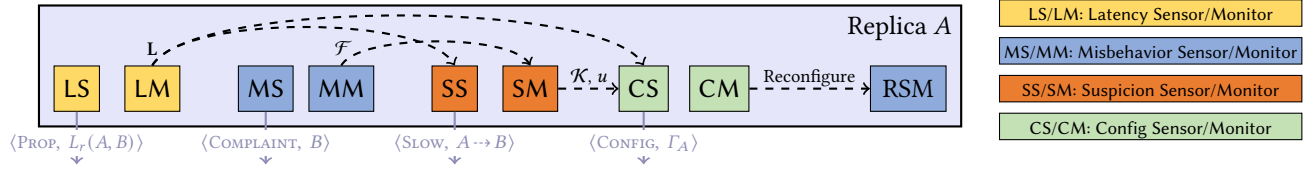
**Figure 3.** Connections between Replica $A$'s sensors and monitors.

## 4.2 Low-Latency Role Assignment

In this section, we present a pipeline of four sensors and monitors suitable for assigning roles for low-latency configurations under Byzantine failures. The pipeline comprises: a LatencySensor to measure link latencies, a SuspicionSensor to suspect replicas whose protocol messages are significantly delayed compared to their reported latencies, a Misbehavior-Sensor to report on provable misbehavior, and a Config-Sensor to search for an efficient configuration.

While Aware [13] uses latency measurements to optimize configurations, our novelty is in combining latency measurements with suspicions, both to detect protocol latencies that deviate from expectations and to incorporate suspicions into the configuration search. Additionally, through the use of the generic sensor and monitor abstraction, we can design reusable components, as we show by integrating them into different BFT systems in §5 and §6. Applying our sensors to different BFT protocols and topologies requires certain functionality to be provided by the underlying protocol, which we highlight in the description of each sensor.

### 4.2.1 Latency Monitoring.
We now discuss our approach to measuring latencies between replicas. For protocols with direct message exchanges, like HotStuff [63], the latency sensor measures the *round-trip latency* of the protocol messages themselves. For protocols where replicas do not issue direct replies, the sensor instead relies on dedicated probe messages. Each replica compiles its measurements into a *latency vector*, which it submits to the consensus engine. Any replica that fails to reply is marked as $\infty$ in the latency vector. We assume that the consensus engine prevents the censoring of latency vectors—for example, by forcing a leader change upon message omission [19] or rotating leaders frequently [63].

Each replica's LatencyMonitor maintains a *latency matrix* $\mathbf{L}$, representing inter-replica latencies. When a measurement is committed, the monitor updates the sender's row with the new latency vector. Symmetry is preserved by setting $\mathbf{L}_{A,B} = \mathbf{L}_{B,A} = \max(L_r(A, B), L_r(B, A))$, where $L_r(A, B)$ is the *recorded* latency between $A$ and $B$.

While measuring latencies of protocol messages may reduce overhead and yield precise estimates, probe messages can fill the latency matrix faster. In HotStuff, for instance, a rotating leader needs $2n$ views to fill the matrix when it relies solely on protocol messages. While measurements using protocol messages must be implemented within the protocol, the LatencySensor, which uses probe messages, does not require any functionality from the underlying protocol.

We measure latency periodically. When measurement is performed after GST, the recorded latency between correct replicas is within a factor $\delta$ of the actual latency. No assumption is made about recorded latencies between replicas where at least one replica is faulty. However, our suspicion sensor, presented below, is designed to detect faulty replicas that under-report their latencies. Moreover, in the examples we consider, faulty replicas that inflate the latencies of adjacent links are less likely to be assigned critical roles.

### 4.2.2 Misbehavior Monitoring.
A significant challenge for scalable RSMs is to stay optimized despite faulty replicas disrupting the configuration. OptiLog provides two monitors for detecting faulty replicas: one precise (this section) and one for suspicions (§4.2.3).

Precise detection is possible using the *proof-of-misbehavior* technique [7, 23, 35, 45, 55]. The MisbehaviorSensor, integrated into the consensus engine, observes the replicas' protocol-specific behavior and raises a *complaint* when detecting a provable violation. Complaints are signed and proposed via the sensor app and logged. Detected misbehaviors include invalid threshold signatures, proposals, votes, complaints, and equivocation. An equivocation is identified when replicas detect the leader sending different messages despite the protocol dictating it should send identical messages. What entails a misbehavior depends on the protocol or even the implementation. Therefore, the detection functionality for the MisbehaviorSensor must be provided by the underlying protocol.

When a complaint proposal is received, each replica's MisbehaviorMonitor verifies it. If valid, the monitor updates its *list of provably faulty replicas* $\mathcal{F}$, which is used to exclude replicas from special protocol roles, such as leader.

### 4.2.3 Suspicion Monitoring.
Gathering enough evidence for a proof-of-misbehavior complaint is often infeasible. We therefore add suspicion monitoring to detect timing and omission faults. These suspicions are then used to build a candidate set that excludes the suspected replicas.

Our SuspicionSensor applies to leader-driven RSM protocols where proposals are committed through repeated message exchanges. We refer to each proposal-commit cycle as a round. The SuspicionSensor assumes replicas have access to synchronized clocks.
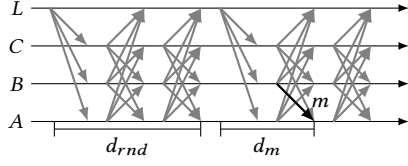
**Figure 4.** PBFT message pattern showing expected round duration $d_{rnd}$ and message delay $d_m$.

The SuspicionSensor's main goal is to detect replicas that fail to meet their claimed latencies as reported in **L**. Detecting these deviations is challenging, since protocol messages often do not follow a request-reply pattern. For example, consider the message pattern in PBFT [19], illustrated in Fig. 4. For replica $A$ to determine whether the highlighted message $m$ is delayed, it must know when replica $B$ should have sent $m$.

To solve this issue, we let the leader timestamp its proposal when starting a round and use this timestamp as a reference point for timeouts on individual messages. Thus, in our example, replica $A$ would expect message $m$ to arrive a certain time after the proposal timestamp. This timeout is shown as $d_m$ in Fig. 4.

Additionally, the SuspicionSensor checks that leaders do not delay timestamping. It does so by verifying that consecutive proposal timestamps fall within the expected round duration, denoted $d_{rnd}$.

The SuspicionSensor requires the underlying protocol to provide functionality to compute the estimated round duration $d_{rnd}$ for a configuration. The protocol must also provide $d_m$ for each message.

The conditions for raising a suspicion are summarized in the table below. Here, $A \dashrightarrow B$ denotes that $A$ suspects $B$.

| Condition for Suspicion | Suspicion |
|---|---|
| (a) Proposal timestamps not within $\delta \cdot d_{rnd}$ | $\langle \text{Slow},\ A \dashrightarrow L \rangle$ |
| (b) No message $m$ from $B$, $\delta \cdot d_m$ after round start | $\langle \text{Slow},\ A \dashrightarrow B \rangle$ |
| (c) False suspicion from $B$ on $A$ | $\langle \text{False},\ A \dashrightarrow B \rangle$ |

First, $A \dashrightarrow L$ if two consecutive proposal timestamps from the leader differ by more than $\delta \cdot d_{rnd}$. As discussed above, this ensures that rounds are started at the correct interval. Second, $A \dashrightarrow B$ if a message $m$ from $B$ to $A$ does not arrive within $\delta \cdot d_m$ after the leader's proposal timestamp. This mechanism also raises suspicions against the leader if it creates the proposal on time but delays sending it, thereby delaying round initialization. The timeouts $d_m$ and $d_{rnd}$ are adjusted by the factor $\delta$ to account for variations in latency. Finally, if a $\langle \_,\ B \dashrightarrow A \rangle$ is raised, then $A$ reciprocates by raising a $\langle \text{False},\ A \dashrightarrow B \rangle$. The last condition helps to distinguish suspicions raised against crashed replicas, which cannot reciprocate, from suspicions against misbehaving replicas.

In complex message patterns, a single delayed message may delay subsequent messages and trigger multiple suspicions. Such causally related suspicions are filtered in the SuspicionMonitor. To enable this filtering, a suspicion includes the message type and round that caused it. For each round, the SuspicionMonitor retains only the first suspicion, discarding suspicions from later protocol phases. Additionally, if the leader raises suspicion for a message in round $i$, suspicions against a delayed proposal timestamp in round $i + 1$ are also filtered.

The SuspicionSensor guarantees that if a protocol does not finish a round within $\delta \cdot d_{rnd}$, a suspicion will be raised. Additionally, if $d_{rnd}$ and $d_m$ are configured correctly, then after GST, any suspicion between correct processes will be filtered out. Thus, the SuspicionSensor ensures that, after GST, a configuration either meets expected latencies or raises a useful suspicion. Appendix C formalizes the properties that $d_{rnd}$ and $d_m$ must satisfy and provides proofs of these guarantees.

***Candidate Selection.*** The SuspicionMonitor processes suspicions to produce a *candidate set* $\mathcal{K}$ of replicas considered correct, along with an estimate $u$ of the number of misbehaving replicas. To this end, the SuspicionMonitor first removes provably faulty replicas $\mathcal{F}$, reported by the MisbehaviorMonitor. Next, the SuspicionMonitor distinguishes between crash suspicions and misbehavior suspicions to produce a good estimate $u$.

The SuspicionMonitor must ensure that faulty replicas causing suspicions are removed from the candidate set and that the number of misbehaving replicas is not underestimated. Failure to do so may allow faulty replicas to repeatedly interrupt the system, by rendering a configuration dysfunctional. On the other hand, the SuspicionMonitor must provide a sufficiently large candidate set to ensure all critical roles can be assigned to candidates. Further, a larger candidate set and a lower estimate of misbehaving replicas permit more choices when assigning roles and may thus result in a more efficient configuration. We design the SuspicionMonitor to always provide $n - f$ candidates and to ensure that if repeated suspicions are raised against faulty replicas after GST, eventually none of these replicas are included in the candidate set, as we prove in Appendix C.

Condition (c) allows us to distinguish suspicions against crashed replicas from those caused by misbehaving replicas. A crashed replica $B$ will be suspected by others through $\langle \text{Slow},\ A \dashrightarrow B \rangle$. If a faulty replica $B$ raises $\langle \text{Slow},\ B \dashrightarrow A \rangle$ against a correct replica $A$, then $A$ will reciprocate with $\langle \text{False},\ A \dashrightarrow B \rangle$. Thus, a replica suspected of being slow without raising a counter-suspicion is *considered crashed.* Conversely, if two replicas suspect each other, it remains unclear which one is faulty. The SuspicionMonitor keeps separate data structures for these two cases: a set $C$ for crashed

replicas and a *suspicion graph* $\mathcal{G}$ for other suspicions. Formally, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an undirected graph where the vertices $\mathcal{V} = \{\Pi \setminus \mathcal{F} \setminus C\}$ represent possible candidate replicas, and an edge $(A, B) \in \mathcal{E}$ indicates a two-way suspicion, $A \leftrightarrow B$.

When a suspicion $\langle \text{Slow}, \ A \dashrightarrow B \rangle$ is raised against a correct replica $B$, it may take several views before a reciprocation $\langle \text{False}, \ B \dashrightarrow A \rangle$ is logged, especially if faulty replicas attempt to censor such suspicions. We therefore treat every new suspicion between two replicas in $\mathcal{V}$ as a two-way suspicion by adding the corresponding edge to $\mathcal{G}$. If no reciprocation is received within $f + 1$ leader changes (views), the edge is treated as a one-way suspicion, and $B$ is added to $C$.

The SuspicionMonitor computes a maximum independent set from the vertices in $\mathcal{G}$ and uses it as the candidate set $\mathcal{K}$. This computation must be deterministic, to ensure that all correct replicas reach the same conclusion. Additionally, the monitor outputs the estimated number of misbehaving replicas, $u = |\mathcal{V}| - |\mathcal{K}|$.

Before GST, unstable latencies may cause suspicions, even between correct replicas. To prevent these suspicions from persisting indefinitely, the SuspicionMonitor employs two mechanisms. First, if the system remains stable with no new suspicions for $w$ views, the monitor begins removing old suspicions in the order they appear in the log. Second, if too many suspicions are received, it also starts discarding old suspicions. Too many suspicions occur when $\mathcal{G}$ no longer contains an independent set of size $n - f$.

### 4.2.4 Configuration Monitoring.

OptiLog's monitoring architecture is also used to trigger reconfigurations. The ConfigSensor searches for a new configuration and proposes it via the log. The ConfigMonitor then selects among the proposed configurations and initiates reconfiguration.

This separation has several benefits. First, while the monitor's reconfiguration decision must be deterministic and taken by all replicas, the sensor's search for a better configuration can be performed by a small subset of replicas. Second, when the search space is large, the sensor can use more efficient non-deterministic search algorithms. Lastly, the search space can be divided among the replicas.

The ConfigSensor utilizes measurements from the other monitors to find a new configuration. We say that a configuration is valid if all special roles are assigned to replicas from the candidate set $\mathcal{K}$. A search can be triggered periodically, or by an updated $\mathcal{K}$ that invalidates the current configuration. The remaining configurations are ranked using a $\text{score}(\cdot)$ function, and the best-scoring configuration, $\Gamma$, is proposed to the log. To achieve low-latency configurations, $\text{score}(\cdot)$ predicts the round duration $d_{rnd}$ using the latency matrix $\mathbf{L}$ from the LatencyMonitor together with the estimate of misbehaving replicas $u$ from the SuspicionMonitor. By incorporating $u$, $\text{score}(\cdot)$ can account for potential message delays caused by misbehaving replicas. Thus, $\text{score}(\cdot)$ can assume that $u$ replicas with normal roles will not respond.

Hence, by using $u$, $\text{score}(\cdot)$ can adapt to the actual number of faults in the system, yielding significantly faster configurations compared to assuming the worst-case of $f$ faults.

When assigning multiple roles, it quickly becomes intractable to compute the score for all configurations. In such cases, we use a heuristic optimization technique based on simulated annealing [40]. We initiate the search with a random configuration, ensuring that special roles are only selected from $\mathcal{K}$. Simulated annealing uses a *mutate* function to randomly swap two replicas in the configuration. Our mutate function ensures that replicas with special roles are only swapped with other replicas from $\mathcal{K}$.

Simulated annealing then compares the score of the mutated configuration with that of the best configuration found so far and probabilistically selects the new configuration for further mutation. While simulated annealing allows exploration beyond local minima, it does not guarantee a global minimum. The search ends when the *search timer* expires or when simulated annealing converges. At this point, the ConfigSensor logs the best configuration found.

We note that our search procedure is not deterministic. Indeed, when different replicas start the search with different initial configurations, it is more likely that one of them finds the best configuration.

When the ConfigMonitor receives a configuration proposal $\Gamma$, it first checks that the proposal represents a *valid configuration*. The monitor also checks if the current configuration has become invalid, e.g., due to an updated $\mathcal{K}$ from the SuspicionMonitor. If the current configuration is invalid, reconfiguration is needed. However, the monitor waits for at least $f + 1$ replicas to propose new configurations before selecting the one with the best score. This prevents reconfiguring to a potentially suboptimal configuration proposed by a faulty replica. The monitor can trigger a reconfiguration even if the current configuration is still valid but has been active for a while. In this case, the monitor requires the new configuration to have a significantly better score to avoid frequent reconfigurations that could disrupt the system. We note that a replica can validate the score of configurations proposed by others and independently decide which configuration to accept only because measurements are recorded in the log and metrics are consistent across replicas.

## 5 OptiAware

We first study how OptiLog can be applied to BFT-SMaRt [16], a modern PBFT-like replication library that features many optimizations. These optimizations are often realized through protocol extensions such as Wheat [57] and Aware [13], which we use for our study. Both protocols assign higher voting weights to some replicas. These weighted votes enable faster consensus, especially when $n > 3f + 1$, as fewer replies are needed to make progress in favorable conditions.

Aware extends Wheat with infrastructure for measuring round-trip latencies using probe messages. It also adds a reconfiguration framework that reassigns the leader role and weights to minimize a configuration's latency. However, Aware does not include infrastructure to detect misbehaving replicas, for example replicas that respond quickly to probes but handle protocol messages slowly.

OptiAware applies OptiLog to Aware. OptiAware's main novelty is the addition of misbehavior and suspicion monitoring. Further, when searching for a configuration, OptiAware avoids replicas not in the candidate set.

As mentioned in §4.2.3 and §4.2.4, the main functionality necessary to apply OptiLog to a protocol are

1. provide a $\text{score}(\cdot)$ function for configuration search
2. provide a procedure to estimate $d_{rnd}$ and $d_m$.

Aware already provides a $\text{score}(\cdot)$ function that computes the round duration $d_{rnd}$ from the latency matrix $\mathbf{L}$. We use that function in OptiAware. The $\text{score}(\cdot)$ function computes the estimated time for the arrival of messages in a round, ignoring the slowest messages, when a quorum is formed. For the duration estimate $d_m$ we use the same approach as $\text{score}(\cdot)$, but only consider messages causally preceding the sending of a message $m$ in a round.

By integrating OptiLog into Aware, we can achieve an adaptive BFT protocol that can optimize its configuration to achieve low latency in a WAN (which is what Aware does) *and* react to malicious delays and omissions carried out by Byzantine replicas that try to slow down the system.

## 6 OptiTree

This section introduces OptiTree, a variant of Kauri [51], utilizing OptiLog to construct correct, low-latency trees.

### 6.1 Preliminaries for OptiTree

**6.1.1 A Brief Primer on Kauri.** This section gives an overview of Kauri and its reconfiguration approach. To improve scalability, Kauri replaces HotStuff's star topology with a tree topology (Fig. 5), where leader $R$ disseminates proposals top-down and aggregates votes bottom-up. We refer to replicas with a specific role or placement in the tree as nodes. In this structure, Kauri's *internal nodes*, $\mathcal{I} = \{R, I_1, \ldots, I_b\}$, manage only $b$ child nodes, where $b$ is the tree's branch factor. As a result, the internal nodes experience a much lower load than HotStuff's leader, which must interact with $n - 1$ replicas. Let $\mathcal{M} = \mathcal{I} \setminus \{R\}$ denote the *intermediate nodes*, and $\mathcal{L} = \Pi \setminus \mathcal{I} = \{T_1, \ldots, T_{b^2}\}$ the *leaf nodes*.

Kauri exploits the parallelism of its tree topology through pipelining. Specifically, the root can initiate multiple consensus instances concurrently, resulting in higher throughput than a comparable star topology.

However, it is hard to construct trees in the presence of faults. That is, the internal nodes of a tree can control both the dissemination and aggregation stages of the protocol. In a
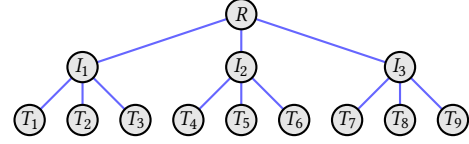


**Figure 5.** Tree with $n = 13$ replicas and branch factor $b = 3$.

star topology, assuming an honest leader, protocol execution may succeed despite some faulty votes. This may not be the case for a tree topology. To address this concern, Kauri forms a correct tree using a reconfiguration algorithm based on $t$-Bounded Conformity [51]. This method involves dividing the replicas into $t$ disjoint bins, assuming that $f < t$, where $t = n/i$ and $i$ is the number of internal nodes. Kauri then constructs trees using these bins, with replicas in each bin serving as internal nodes and the rest as leaf nodes. If $f < t$, there is always at least one fault-free bin, ensuring at least one tree with correct internal nodes. If Kauri is unable to establish a correct tree within $t$ reconfiguration trials, it reverts to using a star topology. Neiheiser et al. [51] show that a three-level tree can guarantee liveness if the internal nodes are correct. Their reconfiguration algorithm exploits this guarantee by iterating through trees constructed from disjoint sets of internal nodes.

**6.1.2 Kauri's Challenges.** In a wide-area environment, Kauri may face several challenges: 1) A single faulty replica may exclude many other replicas from being considered as internal nodes. Moreover, if latencies are not uniform, the performance of successive trees may degrade significantly. We show this in §7.5. 2) Kauri reconfigures only when a tree fails to collect $q$ replies. For example, if a low-latency subtree fails, Kauri may wait for a higher-latency subtree to respond, resulting in degraded performance. 3) Kauri's reconfiguration approach only supports $O(\sqrt{n})$ reconfigurations.

To address Challenge 1, OptiTree uses the suspicion sensor of OptiLog to eliminate faulty replicas from the candidate set individually or paired with at most one correct replica. For Challenge 2, the estimate on faulty nodes $u$ provided by the SuspicionMonitor in OptiLog allows OptiTree to successively adjust the failure threshold of the tree. Further, we prove that in any scenario with up to $f$ faults, OptiTree uses at most $2f$ reconfigurations to find a correct tree, addressing Challenge 3.

### 6.2 Overview of OptiTree

OptiTree finds correct, low-latency trees for large-scale RSM deployments. A *correct tree* is one where the internal nodes are correct, ensuring that the tree can collect a quorum of votes. OptiTree assigns the roles of root, intermediate node, and leaf node connected to a specific internal node. As observed in Kauri [51], a tree remains functional as long as all internal nodes (root and intermediate) are correct. We

therefore only consider the roles of internal nodes as special, ensuring that OptiLog assigns these roles to candidate replicas considered correct ($\mathcal{K}$).

However, an arbitrary correct tree may be suboptimal, and so OptiTree additionally aims to minimize the tree's latency based on the replicas' recorded latencies. We define *tree latency* as the time it takes for the root to disseminate the proposal and aggregate a quorum of votes. A tree fails when it cannot collect a quorum of votes within a timeout. In the following, we explain how OptiLog's sensors and monitors are adjusted to match the characteristics of a tree topology.

In §6.3, we show how to provide the necessary functionality to apply OptiLog to Kauri. In §6.4, we describe a new variant of the SuspicionMonitor that finds a working tree in fewer iterations. This shows how the OptiLog architecture can further be adjusted to a specific use case.

## 6.3 OptiLog for Tree Topologies

To apply OptiLog to assign roles for a tree topology, we must implement the functionality highlighted in §4.2, namely a score($\cdot$) function for configuration search and a procedure to estimate $d_{rnd}$ and $d_m$. Further, for OptiTree we also augment the MisbehaviorMonitor with a special rule, as described below.

Our score($\cdot$) function is defined in Definition 1. It computes the minimum latency required to collect votes from $k = q + u$ nodes in a tree, where $u$ is the estimate of faulty nodes provided by the SuspicionMonitor. This estimate ensures that the tree can meet the predicted latency despite up to $u$ unresponsive nodes.

**Definition 1.** For a tree $\tau$, the score($k, \tau$) is the minimum latency required to collect votes from $k = q + u$ nodes.

This score is computed using the link latencies in the tree, derived from $\mathbf{L}$, taking into account that intermediate nodes first collect votes from their children before forwarding them to the root.

The *aggregation latency* $L_{agg}(I)$, for intermediate node $I$, is the maximum latency from $I$ to any of its children, Ch($I$):

$$L_{agg}(I) = \max_{V \in \text{Ch}(I)} \mathbf{L}_{I,V}$$

Let $\mathcal{M}_k$ represent subsets of all intermediate nodes such that the subtrees rooted at these nodes contain a total of at least $k$ nodes:

$$\mathcal{M}_k = \{M \subset \mathcal{M} \mid \sum_{I \in M} |\text{Ch}(I)| + 1 \geq k\}$$

Thus, the time to collect $k$ votes is the minimum time to collect aggregates from subtrees in a set from $\mathcal{M}_{k-1}$, since the root $R$'s vote is added separately.

$$\text{score}(k, \tau) = \min_{M \in \mathcal{M}_{k-1}} \left( \max_{I \in M} \left( L_{agg}(I) + \mathbf{L}_{I,R} \right) \right)$$

The score($\cdot$) function is also used to estimate the round duration $d_{rnd}$. For message duration estimate $d_m$ we sum the latency of causally preceding messages in a round.

OptiTree also provides an additional rule for misbehavior monitoring of invalid vote aggregation. In a tree configuration, intermediate nodes are responsible for aggregating votes from their children and forwarding them to the root. However, if an intermediate node does not receive a vote from a child node, the aggregate vote must include a suspicion for the missing vote. That is, the aggregate must include $b + 1$ votes or suspicions. Failure to meet this requirement allows the root to use the aggregation vote as *proof-of-misbehavior* against the intermediate node.

We note that, as an additional optimization, the suspicion sensor can be simplified in OptiTree. For example, for suspecting leaf nodes, intermediate nodes can rely on measuring round trip times, rather than relying on the delay relative to proposal timestamps. Additionally, due to the added misbehavior rule, intermediate nodes need to either include a leaf's vote on time or suspect that leaf. Therefore, suspicion monitoring for condition (b) in §4.2.3 can be omitted at leaf nodes.
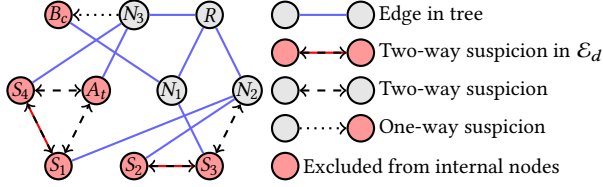
## 6.4 Candidates for Trees

OptiTree assigns the roles of internal nodes to replicas in the candidate set $\mathcal{K}$. However, since a configuration has only $b + 1 \approx \sqrt{n}$ many internal nodes, OptiTree does not require that the candidate set contains $n - f$ replicas, as is the case in OptiLog. In the following, we describe an updated SuspicionMonitor that computes a smaller candidate set but guarantees that the replicas in that candidate set are correct after fewer reconfigurations (at most $2f$) compared to the approach in §4.2.3. This shows the flexibility of OptiLog's architecture.

When computing a maximum independent set for candidate selection (§4.2.3), a single intermediate node suspected by a leaf may cause the candidate set to update and require a reconfiguration. Faulty replicas may force $\Omega(f^2)$ reconfigurations before they are excluded from the candidate set [39].

We introduce two additional sets, $\mathcal{E}_d$ and $\mathcal{T}$, derived from the suspicion graph $\mathcal{G}$. $\mathcal{E}_d$ is a maximal set of disjoint edges in $\mathcal{G}$. For every edge in $\mathcal{E}_d$, at least one of the adjacent vertices is a faulty replica. We therefore exclude both replicas from $\mathcal{K}$. Whenever an edge is added to $\mathcal{G}$, the SuspicionMonitor checks if $\mathcal{E}_d$ is still maximal. This check includes possibly removing one edge from $\mathcal{E}_d$ and adding two new ones. The cost of maintaining $\mathcal{E}_d$ is $O(e^2)$, where $e$ is the number of edges in $\mathcal{G}$. We also determine $\mathcal{T}$, the set of vertices that are not adjacent to an edge in $\mathcal{E}_d$ but part of a triangle in $\mathcal{G}$ with an edge in $\mathcal{E}_d$. We also exclude replicas in $\mathcal{T}$ from $\mathcal{K}$.

Finally, we return $\mathcal{K}$ containing replicas (vertices) in $\mathcal{G}$ that are not in $\mathcal{T}$ or adjacent to an edge in $\mathcal{E}_d$. We also return the estimated number of faulty replicas $u = |\mathcal{E}_d| + |\mathcal{T}|$.

**Figure 6.** Example of suspicions leading to removal of nodes as candidates for internal tree positions.

Fig. 6 illustrates a suspicion graph $\mathcal{G}$ and the resulting tree configuration with excluded nodes. In this example, $\mathcal{E}_d = \{(S_1, S_4), (S_2, S_3)\}$, as adding any other suspicion would violate the disjoint property of $\mathcal{E}_d$, which is already maximal. The set $\mathcal{T} = \{A_t\}$ includes nodes that form a triangle with one edge in $\mathcal{E}_d$ and two others in $\mathcal{G}$. As $B_c$ has a one-way suspicion, it is added to $C$. Since all replicas from $\mathcal{T}$, $\mathcal{E}_d$, and $C$ are excluded from $\Pi$, we get $\mathcal{K} = \{N_1, N_2, N_3, R\}$.

We provide proofs of OptiTree's correctness in Appendix D. We prove that after GST, at most $2f$ reconfigurations are needed to form a correct tree. Especially, if $t < f$ replicas are faulty, at most $2t$ reconfigurations are needed.

## 7 Evaluation

In this section, we demonstrate OptiLog's usefulness for BFT protocols by evaluating the use cases: OptiAware and OptiTree. For OptiAware, we show that using OptiLog, particularly its suspicion monitoring, prevents Aware from performance degradation, for which we evaluate its runtime behavior under attack (§7.1). We also assess the scalability of handling suspicions in OptiAware (§7.2).

For OptiTree, we show that integrating OptiLog sensors and monitors yields better tree configurations, improving overall system performance. After describing OptiTree's implementation and experimental setup (§7.3), we evaluate its performance against Kauri and HotStuff as baselines (§7.4). We then examine the robustness of OptiTree configurations under malicious replicas (§7.5, §7.6) and analyze how simulated annealing search time impacts overall tree latency (§7.7). Finally, we assess the overhead introduced by Opti-Log (§7.8).

Artifacts used in the evaluation of OptiTree and Opti-Aware, including experimental procedures and execution scripts, are available at https://zenodo.org/records/17169372.

### 7.1 OptiAware Runtime Behavior

We implemented OptiAware inside the Aware framework [13] by integrating OptiLog's monitors. Since we now enriched Aware with the capability to *detect suspicions* (such as malicious replicas delaying messages), we are interested in what happens when an attack occurs. For this purpose, we evaluate OptiAware's runtime behavior under the known *Pre-Prepare delay attack* [7, 21], in which a Byzantine leader intentionally
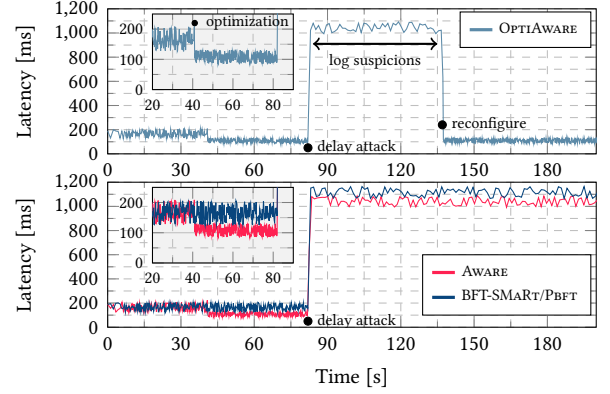


**Figure 7.** Runtime attack: OptiAware vs. baselines.

delays a proposal. We are interested in its ability to detect delayed protocol messages through suspicions and reassign the leader role to maintain optimal performance.

***Setup.*** We deploy one client and one replica in each of 21 European cities using the Phantom network simulator [38]. Clients continuously issue requests to the replicas. Our experiment compares the protocols BFT-SMaRt, Aware, and OptiAware. We measure end-to-end request latency as observed by the clients. During execution, a Byzantine leader launches a Pre-Prepare delay attack at approximately $t = 80\,\mathrm{s}$ to degrade system performance.

***Observations.*** Fig. 7 shows latency measurements from a representative client in Nuremberg. Initially, for $t < 40\,\mathrm{s}$, all protocols exhibit comparable latency. At $t = 40\,\mathrm{s}$, Aware and OptiAware optimize their configuration (i.e., assigning the leader role and voting weights to replicas) as explained in §5. This optimization yields a significant 35% reduction in latency compared to BFT-SMaRt, which remains static. The observed improvement aligns with prior findings on adaptive BFT protocols [13, 14], which also demonstrate performance gains under different experimental setups.

At $t = 82\,\mathrm{s}$, the Byzantine leader initiates the Pre-Prepare delay attack, delaying proposals to inflate client-observed latency. All protocols experience inflated latency. However, OptiAware using OptiLog's SuspicionSensor can detect the delay attack and reassigns the leader role to exclude the malicious replica. Consequently, at $t = 137\,\mathrm{s}$, OptiAware reconfigures to a new optimal configuration, thus restoring low latency, while BFT-SMaRt and Aware remain degraded.

### 7.2 OptiAware Scalability

While the overhead of logging suspicions can be kept low using an efficient encoding scheme (see §7.8), our primary concern here is evaluating the scalability of OptiAware's suspicion handling. To this end, we measure the overhead of computing the maximum independent set in the resulting suspicion graph. This step is necessary to identify candidate
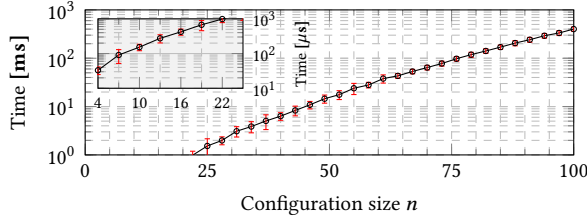
**Figure 8.** Suspicion graph computation time for OptiAware.

replicas for important consensus roles to be assigned during a reconfiguration.

**Setup.** We generate random suspicion graphs for small ($n = 4$) to medium-sized ($n = 100$) system configurations relevant for PBFT. Each graph encodes pairwise suspicions between replicas with edges representing mutual distrust. For each value of $n$, we generate 100 random graphs to account for structural variance. We compute the maximum independent set of each suspicion graph using a heuristic variant of the Bron-Kerbosch algorithm [17], which detects cliques on the inverted graph.

**Observations.** Fig. 8 plots the average time to compute the candidate set for increasing configuration sizes on a VM with two Intel Xeon Gold 6210U cores. Execution times below 1 ms for smaller systems ($n < 25$) are shown in the plot's inset. While the computation time grows rapidly with $n$, it remains below 1 s for all configurations up to $n = 100$, which we consider acceptable for typical PBFT-scale deployments. This suggests the candidate selection process described in §4.2.2 can be efficiently used for reconfigurations.

### 7.3 OptiTree Implementation & Experimental Setup

We implemented OptiLog and OptiTree in an open-source BFT framework based on HotStuff [3]. We also implemented Kauri [51] in the same framework for comparison.

Our primary focus is evaluating our optimizations' performance in deployment scenarios where replicas are distributed globally. To emulate such a distribution of replicas, we integrated a module mimicking network latency between replicas. Each replica is assigned a *location* from a predefined set of cities, and messages are artificially delayed based on the recipient's location. Our network emulator includes 220 worldwide locations, obtained from WonderProxy [6], with intercontinental delays ranging from 150 to 250 ms, in addition to the actual network delay of 1 ms.

Our deployment cluster consists of 30 machines, each with 32GB of RAM and 12-core Intel Xeon processors, connected via a 10Gbps TOR switch. We deployed 4 replicas per machine for all evaluations. We used 4 clients to send requests to replicas, and replicas batch these requests into block proposals. We do not emulate latency between clients and replicas; their communication is instantaneous. Several

existing studies [13, 25, 44] examine client placement and its impact on BFT protocol performance. We did not address this aspect, as it is orthogonal to our work.

We use the Chained HotStuff [63] protocol for consensus. In all experiments, a fixed leader sends proposals and collects a quorum of votes to commit the proposal. We use blocks of 1000 proposals, each without transaction payload. *Throughput* is the number of requests that replicas commit, while *consensus latency* is the time between sending a block proposal and committing the block. Unless specified, any reference to latency refers to consensus latency.

In Kauri, the leader (root) sends proposals in a tree structure, and responses are aggregated up the tree. We implemented pipelining in Kauri by running multiple protocol instances concurrently. We used 3 instances for all experiments with pipelining. OptiTree's main advantage is that it is able to form trees with lower latency. Since pipelining has little effect on latency, we run most experiments without pipelining. In all experiments, trees have a height of 3, and the configuration size $n$ determines the branching factor $b = (\sqrt{4n - 3} - 1)/2$. Taller trees tend to incur overly high latencies. Trees are ranked using the $\text{score}(k, \tau)$ function from Definition 1, with $k = 2f + 1$ as the default.

### 7.4 OptiTree Baseline Comparison

This section evaluates OptiTree compared to Kauri and HotStuff by measuring throughput and latency. This evaluation shows the potential of applying OptiLog to Kauri. We conducted experiments with 21, 43, and 73 replicas, each with 4 clients. Replicas were distributed across cities based on configuration size: 21 cities in Europe, 43 cities across Europe and North America, and 73 cities worldwide. We used the same cities as in Fig. 7.

For experiments with OptiTree, we initially run simulated annealing for one second to search for a low-latency tree configuration. For Kauri, tree configurations are randomly selected. We configure the timers as described in §6.4 with $\delta = 1$. For comparison, Kauri's intermediate node timers are also set based on the recorded latencies $\mathbf{L}$, rather than using a fixed timer [51]. Each experiment runs for 120 seconds, with throughput and latency recorded every second. We report the average throughput and latency over the 120 seconds, with error bars representing the 95% confidence interval.

Fig. 9 shows the throughput and latency results. As expected, pipelined Kauri outperformed HotStuff, while OptiTree achieved significantly better throughput than both protocols. HotStuff-rr refers to HotStuff with round-robin leader selection, while HotStuff-fixed uses a random fixed leader. The tree overlay in Kauri and OptiTree trades reduced latency for higher throughput, but OptiTree mitigates this to a large extend. With 73 replicas distributed worldwide, OptiTree increased throughput by 159% and reduced latency
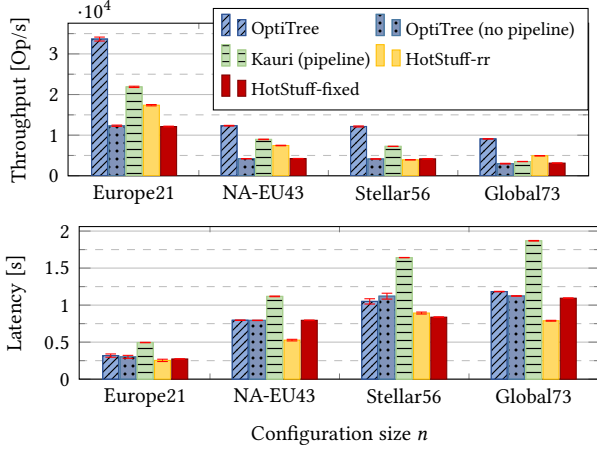
**Figure 9.** Throughput and latency of HotStuff, Kauri, and OptiTree with different geographic distributions.



**Figure 10.** Tree latency (score) with increasing number of reconfigurations. Replicas are randomly distributed across the world.

by 39%. Given that practical deployments [1, 2, 4, 29] often involve hundreds of replicas distributed across diverse locations, OptiTree provides a distinct advantage.

We repeated the experiment on a simulated Stellar [5] network. Stellar provides a list of validators and their location. At the time of the experiment, the Stellar network had 56 validators distributed worldwide. We mapped these validator locations to the closest locations in our network emulator. In this experiment, OptiTree increased throughput by 67.5% and reduced latency by 36% compared to Kauri.

### 7.5 Suspicions' Impact on OptiTree's Latency

In this section we analyze how suspicions and failures affect the latency of OptiTree's configurations. We also examine how faulty replicas may exploit the suspicion mechanism by raising false suspicions. Faulty replicas can exploit the tree formation process by pre-computing the optimal tree based on the recorded latencies and then raise suspicions against the correct internal nodes. This strategy increases tree latency by targeted elimination of a few correct replicas, forming a suboptimal tree. The severity of such an attack depends on the latency distribution of the replicas.

We simulated the attack to assess its impact on OptiTree. Starting with the assumption that all replicas are non-faulty, we first compute a tree using simulated annealing. We then simulate the attack by randomly selecting an internal node to raise suspicion against the root, removing both from the candidate set. The next tree is computed with the remaining replicas as candidates and an increased $u$. This attack is repeated $f$ times to analyze how OptiTree's performance degrades with increasing faults in the configuration. We compare these results with the random trees formed in Kauri, which can only reconfigure up to $\sqrt{n}$ times before reverting to a star topology.
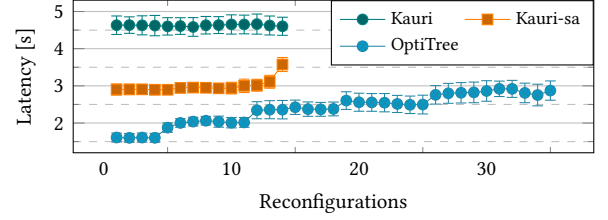
Fig. 10 shows the results for a configuration with 211 replicas, with each data point representing the average of 1000 simulation runs with a 95% confidence interval. OptiTree reconfigures if more than $u$ votes are missing, while Kauri reconfigures if $f$ votes are missing. We therefore report the score for collecting $q + u$ replicas in OptiTree and $q + f$ replicas in Kauri. The increased latency observed in OptiTree is due to the successive rise in $u$ and the reduced availability of candidates. A separate investigation (see Appendix B.1) showed that at 32 suspicions the reduced number of candidates causes 65% of the latency increase, while the rest is due to a larger factor $u$ in the score function. The stepwise pattern reflects the higher latency required to collect votes from additional subtrees.

We observe that Kauri can benefit from simulated annealing to improve tree formation, so we implemented a variant called *Kauri-sa*. However, this variant does not benefit from the increasing $u$ and must account for up to $f$ failures. In Kauri-sa, all internal nodes are excluded from the candidate set after each reconfiguration and a new tree is formed with the remaining replicas as candidate internal nodes. The increased latency towards the last reconfiguration in Kauri-sa is due to the reduced number of available candidates. In contrast, even after 35 suspicions have been recorded, OptiTree retains enough candidates to construct a tree with latency comparable to the initial tree formed by Kauri-sa.

To summarize, OptiTree's increased resilience compared to Kauri-sa is due to OptiLog's suspicion monitoring, which provides both the candidate set and the estimate $u$.

### 7.6 Tolerance to Malicious Delays

As discussed in §4.2.3, OptiLog uses the $\delta$ parameter to adjust timers based on recorded latencies. While this mechanism is designed to tolerate latency variations, faulty replicas may artificially inflate their latency by a factor $\delta$ without raising suspicion. We conducted experiments to assess the impact of such an attack on OptiTree's performance.

In this experiment, we ran OptiTree without pipelining. The experiment involved 21 replicas forming a tree with branch factor 4. We varied the number of faulty replicas from 1 to 4, randomly selecting them from the intermediate
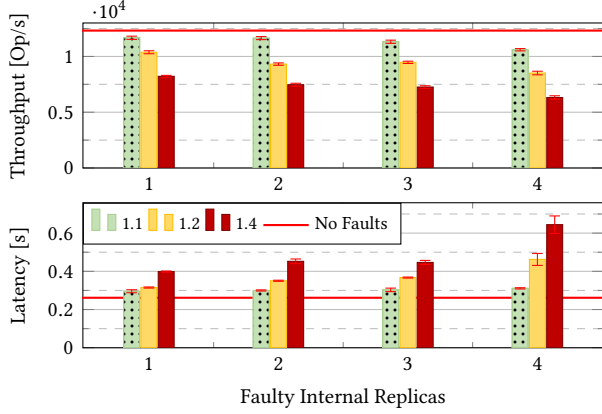
**Figure 11.** Throughput and latency of OptiTree in Europe21 with varying delays imposed by faulty internal nodes.
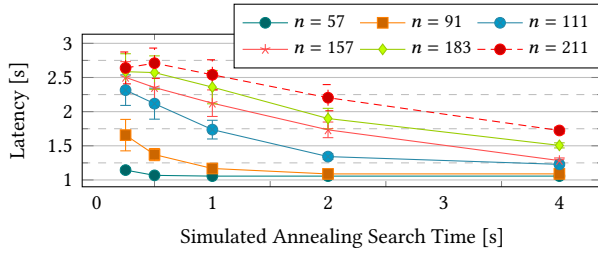


**Figure 12.** Tree latency improves with longer search time.

nodes in the tree. These faulty replicas delay requests to leaf nodes and aggregation messages to the root by adjusting the $\delta$ parameter. We tested $\delta$ values of 1.1, 1.2, and 1.4.

Fig. 11 shows the impact of faulty replicas on throughput and latency. We see that a larger delay allows attackers to reduce performance significantly, reducing throughput by almost 49%. We note that, while a large $\delta$ enables this attack, a small $\delta$ can easily trigger false suspicions.

The above experiments use simulated network latencies. These simulations do not include sudden latency spikes that may occur in real deployments. While OptiLog tolerates spikes within the $\delta$ parameter, our results highlight the trade-off in choosing $\delta$. Fig. 10 illustrates how suspicions, that may be triggered by latency spikes, affect configuration latency. Fig. 11 shows how faulty replicas can exploit a larger $\delta$ to slow down the system. A smaller $\delta$ increases sensitivity to latency variations, causing more suspicions and reconfigurations, whereas a larger $\delta$ tolerates benign variations but allows faulty replicas to increase latency.

OptiLog enables selecting an optimal $\delta$ through historical analysis of recorded latencies. By systematically analyzing past latency data, OptiLog could determine $\delta$ values best suited for varying network conditions, thereby enhancing system resilience and performance. A comprehensive evaluation of this approach is left for future work.
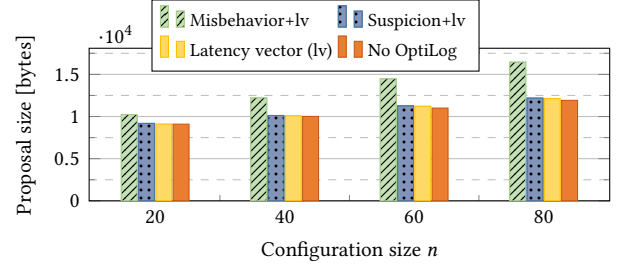


**Figure 13.** Proposal size including different measurements.

### 7.7 Impact of Simulated Annealing on Tree Latency

In this section, we analyze the impact of OptiLog's use of simulated annealing (SA) on tree latency in OptiTree. SA stops when the configured *search time* expires, or the cooling temperature reaches a threshold. We conducted experiments to evaluate how different search times impact the resulting tree's latency. We vary the tree size from 21 to 211 replicas, and the search time is varied from 250 ms to 4 seconds, doubling with each step. Each experiment was run 1000 times, and we report the average tree latency with error bars showing the 95% confidence interval.

Fig. 12 shows how SA improves latency with increasing search time for different configuration sizes. For smaller tree sizes (21 to 57 replicas), SA found no improvements beyond one second of search time. For larger tree sizes, the latency improves with increased search time. For a tree with 211 replicas, a 4-second search resulted in a tree with 35% lower latency than a 250 ms search.

Since SA is a non-deterministic optimization technique, the variance in latency provides insights into its consistency. As the search time increases, this variance decreases, indicating that longer search times improve the likelihood of consistently finding better tree configurations. In addition, we also evaluated SA's impact on throughput during reconfigurations, as detailed in Appendix B.2.

### 7.8 OptiLog Overhead

This section evaluates the increase in proposal size due to OptiLog's sensors. Fig. 13 shows the average proposal size when measurements from different sensors are included, for 20, 40, 60, and 80 replicas distributed across 10 locations. With 80 replicas, including both latency measurements and suspicions increases the proposal size by about 270 bytes compared to the baseline. Adding proofs of misbehavior further increases the proposal size by approximately 4.5 KB. The MisbehaviorSensor includes proofs of misbehavior such as quorum certificates or signatures. Unlike latency measurements, reports from the MisbehaviorSensor and Suspicion-Sensor are infrequent, since complaints are raised at most once per replica. With all sensors active, throughput and latency measurements showed no significant differences in

WAN settings, while in a data center environment throughput was reduced by less than 1% and CPU usage increased by 20%.

## 8  Related Work

Several prior works [50, 56, 65] use accountability as a tool to enhance security by supporting forensics to identify misbehaving replicas. PeerReview [36] introduced a comprehensive method to establish accountability in distributed systems by auditing messages from other replicas. However, due to the extensive message exchange among replicas, this approach may be too costly for RSMs. Polygraph [20] introduced accountable Byzantine agreement by logging protocol messages to justify the decisions made by replicas. IA-CCF [55] improves on Polygraph by providing individual accountability through the use of *ledgers* and *receipts*. It relies on trusted execution environments to secure replicas. Given a set of conflicting receipts and a ledger, any third party can generate proof-of-misbehavior against at least $n/3$ replicas. These systems are designed for forensic analysis and do not utilize detected misbehavior to select a suitable configuration. OptiLog instead focuses on utilizing detected misbehavior or suspicions for informed configuration decisions. While proof-of-misbehavior is a useful part of OptiLog, we show that raising suspicions is sufficient to find performant configurations.

Several systems have optimized configuration latency by selecting leaders based on location [25, 46, 48] or by ranking configurations to identify the optimal one [13, 53, 57]. However, these approaches lack mechanisms to suspect or hold misbehaving replicas accountable for failed or degraded configurations. Systems such as Archer [25] and Aardvark [21] hold only the leader responsible. In contrast, OptiLog's SuspicionSensor detects underperforming replicas, not only leaders, enabling its use in complex configurations such as tree topologies [51], which may fail despite a correct leader. RBFT [10] monitors performance through redundant leader executions, allowing it to detect slow leaders and replace misbehaving ones that deliberately degrade performance. OptiLog achieves similar detection without redundant executions, making it more practical for tree-based BFT protocols where the configuration space is vast.

Abstract [9] introduced a switching mechanism to enable more efficient protocols under favorable conditions and to revert to more resilient protocols when conditions deteriorate. Similarly, Chenyuan et al. [62] propose a system to collect and replicate configuration metrics and use them in a reinforcement learning-based mechanism to switch between protocols. Adapt [11] presents a quality-control system that assesses the suitability of a configuration and protocol based on the current environment. These systems treat the underlying protocols as black boxes. Protocol switching is a useful fallback mechanism that complements intra-protocol

optimizations, such as the role-assignment optimizations enabled by OptiLog.

There are several techniques [28, 32, 52] for estimating latency among Internet-deployed nodes. Some approaches [28] use landmark nodes to estimate latency, while others [32] rely on DNS name servers. GNP [52] estimates unknown path latencies by embedding nodes in a geometric space and using Euclidean distance as the predictor. Variants of these approaches incorporate machine learning, quality-of-service protocols, and application-specific mechanisms. However, latency estimation is challenging in the presence of faulty nodes and attacks [66]. Therefore, OptiLog employs direct latency measurements on all links.

## 9  Conclusion

RSM optimizations are difficult to maintain in dynamically changing environments, especially in the presence of faulty replicas and potentially manipulated measurements.

OptiLog's sensors and monitors tackle this problem by combining collection of local measurement and the execution of non-deterministic computations, with consistent presentations and processing of this information. Moreover, OptiLog selects candidates for special roles and holds faulty replicas accountable to the reported measurements. For instance, by using OptiLog's sensors and monitors, OptiAware can detect-and-mitigate performance attacks and maintain optimal latency. Furthermore, we showcase how OptiTree leverages OptiLog to find more performant configurations and to stay optimized despite failures. In a global deployment, OptiTree can select tree configurations with up to 39% lower latency and achieve 2.5× of Kauri's throughput. In particular, the estimate on actual faults, provided by OptiLog, enables OptiTree to balance performance and robustness.

## References

[1] 2024. Avalanche Network. https://subnets.avax.network/validators/dashboard/.

[2] 2024. Cosmos. https://www.mintscan.io/cosmos/validators.

[3] 2024. Relab HotStuff. https://github.com/relab/hotstuff.

[4] 2024. Solana. https://solanabeach.io/validators.

[5] 2024. stellar. https://stellarbeat.io/.

[6] 2024. WonderProxy. https://wonderproxy.com/.

[7] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2011. Prime: Byzantine Replication under Attack. *IEEE Transactions on Dependable and Secure Computing* 8, 4 (2011), 564–577. https://doi.org/10.1109/TDSC.2010.70

[8] James W. Anderson, Hein Meling, Alexander Rasmussen, Amin Vahdat, and Keith Marzullo. 2017. Local Recovery for High Availability in Strongly Consistent Cloud Services. *IEEE Transactions on Dependable and Secure Computing* 14, 2 (2017), 172–184. https://doi.org/10.1109/TDSC.2015.2443781

[9] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems* 32, 4, Article 12 (jan 2015), 45 pages. https://doi.org/10.1145/2658994

[10] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. RBFT: Redundant Byzantine Fault Tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*. 297–306. https://doi.org/10.1109/ICDCS.2013.53

[11] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. 2015. Making BFT Protocols Really Adaptive. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 904–913. https://doi.org/10.1109/IPDPS.2015.21

[12] Omar Mohammed Bakr and Idit Keidar. 2008. *On the Performance of Quorum Replication on the Internet.* Technical Report UCB/EECS-2008-141. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-141.html

[13] Christian Berger, Hans P. Reiser, Joao Sousa, and Alysson Bessani. 2022. AWARE: Adaptive Wide-Area Replication for Fast and Resilient Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2022), 1605–1620. https://doi.org/10.1109/tdsc.2020.3030605

[14] Christian Berger, Lívio Rodrigues, Hans P Reiser, Vinicius Cogo, and Alysson Bessani. 2024. Chasing Lightspeed Consensus: Fast Wide-Area Byzantine Replication with Mercury. In *Proceedings of the 25th International Middleware Conference*. 158–171.

[15] Christian Berger, Signe Schwarz-Rüsch, Arne Vogel, Kai Bleeke, Leander Jehl, Hans P. Reiser, and Rüdiger Kapitza. 2023. SoK: Scalability Techniques for BFT Consensus. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–18. https://doi.org/10.1109/ICBC56567.2023.10174976

[16] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*. IEEE, 355–362. https://doi.org/10.1109/DSN.2014.43

[17] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (Sept. 1973), 575–577. https://doi.org/10.1145/362342.362367

[18] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *CoRR* abs/1807.04938 (2018). arXiv:1807.04938 http://arxiv.org/abs/1807.04938

[19] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*. USENIX Association, 173–186.

[20] Pierre Civit, Seth Gilbert, and Vincent Gramoli. 2021. Polygraph: Accountable Byzantine Agreement. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 403–413. https://doi.org/10.1109/ICDCS51616.2021.00046

[21] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, 153–168.

[22] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*. USENIX Association, 137–150.

[23] Sisi Duan, Karl N. Levitt, Hein Meling, Sean Peisert, and Haibin Zhang. 2014. ByzID: Byzantine Fault Tolerance from Intrusion Detection. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 253–264. https://doi.org/10.1109/SRDS.2014.28

[24] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (apr 1988), 288–323. https://doi.org/10.1145/42282.42283

[25] Michael Eischer and Tobias Distler. 2018. Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 140–145. https://doi.org/10.1109/DSN-W.2018.00053

[26] Michael Eischer, Benedikt Straßner, and Tobias Distler. 2020. Low-Latency Geo-Replicated State Machines with Guaranteed Writes (*PaPoC'20*). ACM, Article 13, 9 pages. https://doi.org/10.1145/3380787.3393686

[27] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-Machine Replication for Planet-Scale Systems. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. ACM, Article 24, 15 pages. https://doi.org/10.1145/3342195.3387543

[28] P. Francis, S. Jamin, Cheng Jin, Yixin Jin, D. Raz, Y. Shavitt, and L. Zhang. 2001. IDMaps: a global Internet host distance estimation service. *IEEE/ACM Transactions on Networking* 9, 5 (2001), 525–540. https://doi.org/10.1109/90.958323

[29] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'17)*. ACM, 51–68. https://doi.org/10.1145/3132747.3132757

[30] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. 2023. BeeGees: Stayin' Alive in Chained BFT. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing (PODC'23)*. ACM, 233–243. https://doi.org/10.1145/3583668.3594572

[31] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19)*. 568–580. https://doi.org/10.1109/DSN.2019.00063

[32] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. 2002. King: Estimating Latency between Arbitrary Internet End Hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment (IMW'02)*. ACM, 5–18. https://doi.org/10.1145/637201.637203

[33] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster Asynchronous BFT Protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20)*. ACM, 803–818. https://doi.org/10.1145/3372297.3417262

[34] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1392–1403. https://doi.org/10.1109/ICDE51399.2021.00124

[35] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2006. The Case for Byzantine Fault Detection. In *Proceedings of the 2nd Conference on Hot Topics in System Dependability - Volume 2 (HOTDEP'06)*. USENIX Association, 5.

[36] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, 175–188. https://doi.org/10.1145/1294261.1294279

[37] Mohammad M. Jalalzai, Costas Busch, and Golden G. Richard. 2019. Proteus: A Scalable BFT Consensus Protocol for Blockchains. In *2019 IEEE International Conference on Blockchain (Blockchain)*. 308–313. https://doi.org/10.1109/Blockchain.2019.00048

[38] Rob Jansen, Jim Newsome, and Ryan Wails. 2022. Co-opting Linux Processes for High-Performance Network Simulation. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC'22)*. 327–350.

[39] Leander Jehl. 2019. Quorum Selection for Byzantine Fault Tolerance. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2168–2177. https://doi.org/10.1109/ICDCS.2019.00213

[40] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220, 4598 (1983), 671–680. https://doi.org/10.1126/science.220.4598.671

[41] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. ACM, Article 25, 17 pages. https://doi.org/10.1145/3342195.3387545

[42] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Security Symposium (SEC'16)*. USENIX Association, 279–296.

[43] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. 583–598. https://doi.org/10.1109/SP.2018.000-5

[44] Johannes Köstler, Hans P. Reiser, Franz J. Hauck, and Gerhard Habiger. 2023. Fluidity: Location-Awareness in Replicated State Machines. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC'23)*. ACM, 192–201. https://doi.org/10.1145/3555776.3577763

[45] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems* 27, 4, Article 7 (jan 2010), 39 pages. https://doi.org/10.1145/1658357.1658358

[46] Shengyun Liu and Marko Vukolić. 2017. Leader Set Selection for Low-Latency Geo-Replicated State Machine. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2017), 1933–1946. https://doi.org/10.1109/TPDS.2016.2634148

[47] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. 2023. Flexible Advancement in Asynchronous BFT Consensus. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'23)*. ACM, 264–280. https://doi.org/10.1145/3600006.3613164

[48] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 369–384.

[49] Michael G. Merideth, Florian Oprea, and Michael K. Reiter. 2009. When and How to Change Quorums on Wide Area Networks. In *2009 IEEE 28th International Symposium on Reliable Distributed Systems (SRDS)*. 12–21. https://doi.org/10.1109/SRDS.2009.35

[50] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. 2007. Ensuring Content Integrity for Untrusted Peer-to-Peer Content Distribution Networks. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*. USENIX Association.

[51] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. 2021. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'21)*. ACM, 35–48. https://doi.org/10.1145/3477132.3483584

[52] T.S.E. Ng and Hui Zhang. 2002. Predicting Internet network distance with coordinates-based approaches. In *Proceedings Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 1. 170–179. https://doi.org/10.1109/INFCOM.2002.1019258

[53] Shota Numakura, Junya Nakamura, and Ren Ohmura. 2019. Evaluation and Ranking of Replica Deployments in Geographic State Machine Replication. In *2019 IEEE 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW)*. 37–42. https://doi.org/10.1109/SRDSW49218.2019.00014

[54] Peter Pacheco. 1996. *Parallel Programming with MPI*. Morgan Kaufmann.

[55] Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cedric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, Julien Maffre, Olga Vrousgou, Christoph M. Wintersteiger, Manuel Costa, and Mark Russinovich. 2022. IA-CCF: Individual Accountability for Permissioned Ledgers. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*. USENIX Association, 467–491.

[56] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. 2021. BFT Protocol Forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. ACM, 1722–1743. https://doi.org/10.1145/3460120.3484566

[57] João Sousa and Alysson Bessani. 2015. Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. In *2015 IEEE 34th International Symposium on Reliable Distributed Systems (SRDS)*. 146–155. https://doi.org/10.1109/SRDS.2015.40

[58] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. 2022. [Solution] Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *Journal of Systems Research* 2, 1 (2022). https://doi.org/10.5070/SR32159278

[59] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State Machine Replication Scalability Made Simple. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys'22)*. ACM, 17–33. https://doi.org/10.1145/3492321.3519579

[60] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. 2023. QuePaxa: Escaping the Tyranny of Timeouts in Consensus. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'23)*. ACM, 281–297. https://doi.org/10.1145/3600006.3613150

[61] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *2009 IEEE 28th International Symposium on Reliable Distributed Systems (SRDS)*. 135–144. https://doi.org/10.1109/SRDS.2009.36

[62] Chenyuan Wu, Haoyun Qin, Mohammad Javad Amiri, Boon Thau Loo, Dahlia Malkhi, and Ryan Marcus. 2024. Towards Truly Adaptive Byzantine Fault-Tolerant Consensus. *SIGOPS Oper. Syst. Rev.* 58, 1 (Aug. 2024), 15–22. https://doi.org/10.1145/3689051.3689055

[63] 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)*. ACM, 347–356. https://doi.org/10.1145/3293611.3331591

[64] Bin Yu, Joseph Liu, Surya Nepal, Jiangshan Yu, and Paul Rimba. 2019. Proof-of-QoS: QoS Based Blockchain Consensus Protocol. *Computers & Security* 87, C (nov 2019). https://doi.org/10.1016/j.cose.2019.101580

[65] Aydan R. Yumerefendi and Jeffrey S. Chase. 2004. Trust but Verify: Accountability for Network Services. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop (EW'11)*. ACM, 37–es.

| Notation | Description |
|---|---|
| $n$ | total number of replicas in the configuration |
| $q$ | number of replicas in a quorum |
| $f$ | number of failures tolerated |
| $t$ | actual faults in the configuration |
| $u$ | estimate on number of non-crash faults in the configuration |
| $b$ | branching factor of the tree |
| $i$ | the number of internal nodes |
| $w$ | window length for sliding window mechanism |
| $k$ | number of nodes from which the tree's root collects votes |
| $A,B,C,D$ | replicas |
| $R$ | root node of a tree |
| $I$ | intermediate node of a tree |
| $T$ | leaf node of a tree |
| $L$ | leader node |
| $A_t$ | a node in a triangle |
| $B_c$ | a crashed node |
| $S_1$ | sensor type 1 |
| $M_1$ | monitor type 1 |
| $\Pi$ | the set of all $n$ replicas |
| $\Gamma$ | a configuration proposal |
| $C$ | set of crashed replicas |
| $\mathcal{F}$ | set of provably faulty replicas |
| $\mathcal{K}$ | set of replicas that are candidates for special roles |
| $\mathcal{G}$ | suspicion graph |
| $\mathcal{V}$ | set of vertices in $\mathcal{G}$ |
| $\mathcal{E}$ | set of edges in $\mathcal{G}$ |
| $Q$ | quorum of nodes in a configuration |
| $\tau$ | a tree |
| $\mathcal{M}$ | set of all intermediate nodes in a tree |
| $\mathcal{I}$ | set of all internal nodes in a tree |
| $\mathcal{L}$ | set of all leaf nodes in a tree |
| $IS$ | independent set of nodes |
| $\mathcal{E}_d$ | maximal set of disjoint nodes with two-way suspicions |
| $\mathcal{T}$ | set of nodes that form a triangle with two nodes in $\mathcal{E}_d$ |
| $A \dashrightarrow B$ | replica $A$ suspects replica $B$ |
| $d_{rnd}$ | expected round duration |
| $d_m$ | expected delay from round start to message arrival |
| $L_r(A,B)$ | recorded latency between replicas $A$ and $B$ |
| $L_a(A,B)$ | actual latency between replicas $A$ and $B$ |
| $L_{agg}(I)$ | time to aggregate votes at intermediate node $I$ |
| $score(\cdot)$ | general score function |
| $score(u,\tau)$ | the minimum latency to collect votes from $u + q$ nodes |
| $\mathbf{L}$ | latency matrix |
| $\mathbf{L}_{A,B}$ | latency matrix element between replicas $A$ and $B$ |
| $\delta$ | multiplier used in timer calculations |
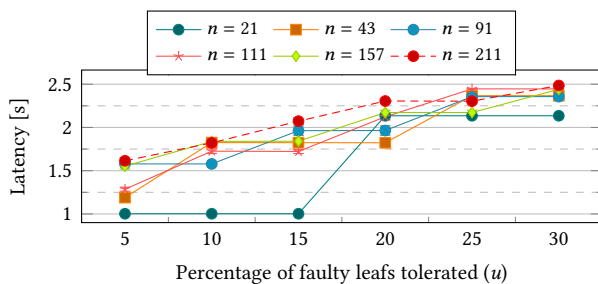


**Figure 14.** Tree latency (score) degradation as the number of faulty nodes increase. Replicas are randomly distributed across the world.

https://doi.org/10.1145/1133572.1133585

[66] David Zage and Cristina Nita-Rotaru. 2010. Robust Decentralized Virtual Coordinate Systems in Adversarial Environments. *ACM Trans. Inf. Syst. Secur.* 13, 4, Article 38 (Dec. 2010), 34 pages. https://doi.org/10.1145/1880022.1880032

[67] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, 15–28.

[68] Gengrui Zhang and Hans-Arno Jacobsen. 2022. ESCAPE to Precaution against Leader Failures. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 625–635. https://doi.org/10.1109/ICDCS54860.2022.00066

## A  Notation

## B  Additional Results

### B.1  Tree Robustness and Configuration Latency

When searching for a low-latency tree, one has to choose between optimizing the latency for fault-free cases, or optimizing the latency in presence of faults, resulting in a lower bound for fault-free cases. OptiTree uses the parameter $u$ to create increasingly robust trees, optimizing for the latency when $u$ leafs become unresponsive. Recall that $u$ is the estimated number of non-crash faults provided by the SuspicionMonitor. We achieve this by using $score(k, \tau)$ (see §6.3) in simulated annealing.

We evaluated the cost of this overprovisioning, by varying $u$ from 5% to 30% of the tree size. Fig. 14 shows the best score when applying simulated annealing with increasing $u$ in the score function. We use latency data for 220 cities from WonderProxy [6]. With a tree size of 211, latency increases by 54% when $u$ is 30% of the tree size. We note that OptiTree overprovisions only in the presence of adversarial omission and timing faults, not for pure crash faults.

### B.2  OptiTree Reconfiguration

We now evaluate OptiTree's reconfiguration mechanism. We consider a scenario with 21 Europe-based replicas, where the root replica becomes faulty and triggers a reconfiguration
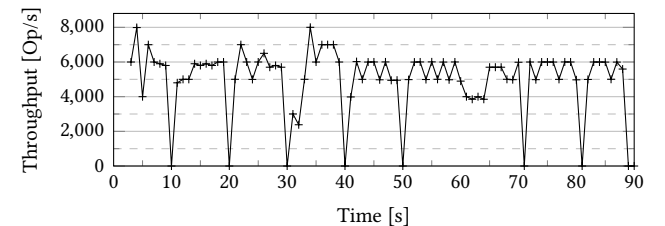


**Figure 15.** Reconfiguration experiment with 21 replicas. The root replica fails every 10 seconds, triggering a simulated annealing search.

every 10 seconds. To make reconfigurations more visible, we do not process any payload, when recording suspicions and searching for a new tree. Fig. 15 shows the throughput observed by a non-faulty replica. We see that after one second used for simulated annealing, throughput is recovered. This experiment shows one of the main tradeoffs in OptiTree. OptiTree achieves better configurations but incurs higher reconfiguration latency than Kauri, when recovering from failed configurations. In scenarios where all replicas are equidistant with each other, alternatives like Kauri may perform comparably or better.

## C  OptiLog Correctness

In this section, we prove correctness of the suspicions monitoring in OptiLog, as described in §4.2.3. We proof the following guarantees:

C1  There are always at least $n - f$ candidates available in $\mathcal{K}$. (Lemma 1)

C2  If some correct replica does not commit a new proposal on average, every $\delta \cdot d_{rnd}$, a suspicion is raised. (Lemma 2)

C3  After GST, any suspicion between two correct replicas is filtered in the SuspicionMonitor. (Lemma 3)

These guarantees ensure that eventually (after GST) the system performs at a steady pace. This pace can be optimized by searching for a role assignment minimizing $d_{rnd}$, within the candidate set, as is done by our configuration sensor .

***C1 – Sufficient Candidates.*** Lemma 1 ensures C1. This is necessary to ensure that the system can always form a new configuration.

**Lemma 1.** There are always at least $n - f$ candidates available in $\mathcal{K}$.

*Proof.* As described in §4.2.3, the SuspicionMonitor selects the candidate set $\mathcal{K}$ as a maximum independent set in the suspicion graph $\mathcal{G}$. Note that, if all suspicions are caused by faulty replicas, then the set of correct replicas is an independent set of size $n - f$. If no independent set of size $n - f$ can be found, old suspicions are removed from $\mathcal{G}$.  □

***C2 – Suspicion are raised.*** Claims C2 (Lemma 2) expresses that suspicions are raised, if the system does not make progress as expected.

Table 2 repeats the conditions for the SuspicionSensor to raise suspicions, as described in §4.2.3. We repeat it here for better readability.

**Table 2.** Suspicion conditions in OptiLog

| Condition for Suspicion | Suspicion |
|---|---|
| (a)  Proposal timestamps not within $\delta \cdot d_{rnd}$ | $\langle \textsc{Slow},\ A \dashrightarrow L \rangle$ |
| (b)  No message $m$ from $B$, $\delta \cdot d_m$ after round start | $\langle \textsc{Slow},\ A \dashrightarrow B \rangle$ |
| (c)  False suspicion from $B$ on $A$ | $\langle \textsc{False},\ A \dashrightarrow B \rangle$ |

**Lemma 2.** If some correct replica does not complete a round, every $\delta \cdot d_{rnd}$, a suspicion is raised.

*Proof.* Let $A$ be a correct replica, that commits a proposal in round $r_i$ after receiving messages $m_1, ... m_k$ and let $t_i^p$ be the proposal timestamp of round $r_i$. If any message $m \in \{m_1, ... m_k\}$ is not received by $A$ at most $\delta \cdot d_m$ after $t_i^p$, then the SuspicionSensor raises a suspicion following condition (b). Additionally, the SuspicionSensor raises a suspicion (b) if consecutive proposal timestamps are not within $\delta \cdot d_{rnd}$, i.e. if $t_{i+1}^p - t_i^p > \delta \cdot d_{rnd}$. Thus, if no suspicions are raised by $A$ proposal timestamps appear at least every $\delta \cdot d_{rnd}$ and rounds are finished timely after the proposal timestamp.  □

We note that Claim C2 follows from Lemma 2 only for protocols that commit a proposal in one round. This is the case for PBFT based protocols. In HotStuff and derived protocols, like Kauri and OptiTree a proposal is not committed in one round. Instead in steady operation in every round, the proposal of a previous round is committed. Thus the claim holds also here.

***C3 – Filtered false suspicions.*** Lemma 3 proves C3, ensuring that suspicions which are not filtered after GST are useful.

Clearly, to avoid false suspicions between correct replicas requires that timeout durations $d_m$ and $d_{rnd}$ are set correctly. The timeout requirements listed as TR1-TR3 below give precise requirements for these timeouts. Below we argue how these requirements are satisfied in OptiAware and OptiTree.

TR1  Let $m$ be a message sent from the leader $L$ to replica $A$ directly after creating the proposal. Then $d_m$ is configured correctly if: $d_m = L_r(L, A)$.

TR2  Let $m$ be a message sent from replica $A$ to replica $B$. Then there exist an earlier message $m'$ sent to $A$, such that $d_m = d_{m'} + L_r(A, B)$.

TR3  There exists a message $m$ sent to the leader $L$, such that $d_{rnd} = d_m$.

**Example C.1. Timeouts in OptiAware:** In OptiAware, the timeout $d_m$ and $d_{rnd}$ are set according to the timeout requirements. Below we discuss how to set timeouts for individual messages and the round timeout. We use message types from the Aware (BFT-Smart) protocol, but also mention equivalent types from PBFT.

- For a Propose message (Preprepare in PBFT) $m_p$, $d_{m_p} = L_r(L, A)$ and thus TR1 holds.
- Write messages in Aware (Prepare in PBFT) are sent after receiving a Propose message. For a Write message $m_w$ from $A$ to $B$, $d_{m_w} = L_r(L, A) + L_r(A, B)$. Thus, TR2 holds for $m_w$ with $m' = m_p$.

- Accept messages in Aware (Commit in PBFT) are sent after receiving a quorum of Write messages. Thus, for an Accept message $m_a$ sent by $A$ to $B$, the following equation holds, where *quorums* is the set of all quorums of Write messages sent to $A$:

$$d_m = \min_{q \in quorums} \left( \max_{m_w \in q} d_{m_w} \right) + L_r(A, B)$$

Thus, TR2 holds for $m_a$ with $m'$ being the write message with the slowest expected duration, in the fastest quorum of write message sent to $A$.

- Finally, in Aware, the round is concluded when the leader $L$ receives a quorum of Accept messages. Thus the round duration $d_{rnd}$ can be computed as follows:, where *quorums* is the set of all quorums of Accept messages sent to $L$:

$$d_{rnd} = \min_{q \in quorums} \left( \max_{m_a \in q} d_{m_a} \right)$$

Thus, TR3 holds with $m'$ being the accept message with the slowest expected duration, in the fastest quorum of accept messages sent to $L$.

We note that the $d_{rnd}$ developed above is the same as the result of the score function defined by Aware. These timeouts are used in OptiAware.

**Lemma 3.** If TR1 - TR3 hold, then after GST, any suspicion between two correct replicas is filtered in the SuspicionMonitor.

*Proof.* The SuspicionMonitor filters suspicions as explained in §4.2.3: *For each round, the SuspicionMonitor retains only the first suspicion, discarding suspicions from later protocol phases. Additionally, if the leader raises suspicion for a message in round $i$, suspicions against a delayed proposal timestamp in round $i + 1$ are also filtered.*
Assume that the latency between correct replicas $A$ and $B$ has been measured and recorded in the latency matrix after GST. According to our system model in §2, after GST, the latency between $A$ and $B$ lies in the interval $[L_a(A, B), \delta \cdot L_a(A, B)]$. Especially, for the recorded latency $\delta \cdot L_a(A, B) < \delta \cdot L_r(A, B)$ holds. Thus, any message $m$ sent from $A$ to $B$ after GST, will have a latency, less than $\delta \cdot L_r(A, B)$. It follows from TR1 that if $A$ is the leader, a message $m$ sent directly after the proposal timestamp, will be received within $\delta \cdot L_r(L, B) = \delta \cdot d_m$. For other message, due to TR2, the expected message duration for $d_m$ is set to $L_r(A, B)$ after another message $m'$. Thus, if a suspicion $B \dashrightarrow A$ is raised, based on a delayed message $m$, $A$ will also have raised a suspicion $A \dashrightarrow C$ based on some message $m'$ enabling sending $m$. Thus, suspicion $B \dashrightarrow A$ will be filtered.

If a correct leader $L$ is suspected based on Condition (a) in Table 2, the leader did not start a new round on time. According to TR3 this means $L$ did not receive a quorum of messages in time. Thus, $L$ will have raised a suspicion in the previous round and the suspicion against $L$ will be filtered.

Finally, a correct replica $A$ will only raise $A \dashrightarrow B$ based on Condition (c) in Table 2 if a suspicion $B \dashrightarrow A$ was already registered. If $B$ is correct this suspicion $B \dashrightarrow A$ will be filtered and $A \dashrightarrow B$ will not be raised. □

## D OptiTree Correctness

In this section, we prove correctness properties for OptiTree. We note that OptiTree uses a different version of the SuspicionMonitor and computes the candidate set differently, as explained in §6.4. We therefore need to prove C1 again as CT1. Similarly, to show CT3, we have to show that the timeout conditions TR1-TR3 hold for OptiTree. OptiTree provides the following guarantees:

CT1 There are always enough candidates available to form a tree. (Theorem D.1)

CT2 If a tree configuration fails and the system reverts to a star topology, sufficient suspicions will be recorded to update the candidate set and invalidate the failed tree configuration. (Lemma 4 & 5)

CT3 After GST, any suspicion between two correct replicas is filtered in the SuspicionMonitor. (Lemma 6)

CT4 After GST, at most $2f$ reconfigurations are needed to form a correct tree. Specifically, if $t < f$ replicas are faulty, at most $2t$ reconfigurations are needed. (Theorem D.2)

***CT1 – Sufficient Candidates.*** Faulty replicas can cast suspicions on non-faulty replicas to exclude them from being selected as internal nodes. However, OptiTree can continue to form and reconfigure to new trees if there are enough candidate replicas to select the internal nodes. The process for selecting candidate internal nodes is detailed in §6.4. It involves constructing $\mathcal{E}_d$ from the suspicion graph $\mathcal{G}$. Theorem D.1 shows that enough candidate replicas are always available to form a tree.

**Theorem D.1.** *There are always enough candidate replicas available for selecting the internal nodes to form a tree.*

*Proof.* A tree configuration of size $n \geq 3f + 1$ with a branch factor $b \leq \sqrt{n}$ requires at most $\sqrt{n} + 1$ replicas as internal nodes. As explained in §4.2.3, the graph $\mathcal{G}$ always contains an independent set ($IS$) of at least $n - f$ replicas. Then, we construct $\mathcal{E}_d$ and $\mathcal{T}$ from $\mathcal{G}$, where $\mathcal{E}_d$ is a maximum set of disjoint edges and $\mathcal{T}$ is a set of nodes that form a triangle in $\mathcal{G}$ with an edge in $\mathcal{E}_d$. For any edge in $\mathcal{E}_d$, at most one of the two vertices is in $IS$. Since $\mathcal{E}_d$ is disjoint, each vertex belongs to at most one edge in $\mathcal{E}_d$. Moreover, for any edge in $\mathcal{E}_d$ that forms a triangle in $\mathcal{G}$, only one of the three vertices forming the triangle can be in $IS$.

Due to the maximality of $\mathcal{E}_d$, any edge in $\mathcal{E}_d$ can be part of at most one triangle involving a replica in $\mathcal{T}$. Therefore, if at most $f$ replicas are outside $IS$, the number of vertices in the independent set adjacent to edges in $\mathcal{E}_d$ or in $\mathcal{T}$ is at

most $f$. Consequently, the remaining replicas in $IS$ (at least $f + 1$) are available as candidates for internal nodes.

For $n \geq 13$, we have $\sqrt{n} < f$, implying that for configuration sizes of 13 and above, there are always enough candidate replicas available for selecting internal nodes.  □

*CT2 – Suspicions are raised.* It follows from Lemma 2, that suspicions are raised if a tree configuration does not collect votes in time. The following Lemmas show additional properties on which suspicions are raised. This ensures that the candidate set is updated and the failed tree configuration is no longer valid. Additionally, we use the properties about what suspicions are raised in the proof of Theorem D.2 to show that after enough trees have failed, faulty replicas are excluded from the candidate set.

**Lemma 4.** When a tree fails, either (1) one suspicion between internal nodes is raised, or (2) $u + 1$ leaves, not in $C$ get suspected.

*Proof.* A tree fails, if the replica at the root fails, or cannot collect $n - f$ votes in time. If the root fails, it is suspected by other internal nodes, matching (1) from the Lemma. Otherwise, the root waits for aggregates from internal nodes, which together aggregate more than $n - f + u$ votes. Thus, there are two possibilities. Either at least $u + 1$ leaves are missing and suspected by their parents, matching (2), or one internal node is either missing or did send a faulty aggregate, and is suspected by the root, matching (1).  □

**Lemma 5.** If a tree failed, either: (i) $|\mathcal{E}_d|$ increases, or (ii) $|\mathcal{T}|$ increases and $|\mathcal{E}_d^n|$ stays unchanged.

*Proof.* All suspicions are added to the graph $\mathcal{G}$. In case (1) of Lemma 2, the new suspicion can be added to $\mathcal{E}_d$ and $\mathcal{T}$ remains unchanged. In case (2), $u + 1$ edges are added to $\mathcal{G}$. These new edges include $u+1$ different leaves from the failed tree. These leaves, can be already adjacent to an edge in $\mathcal{E}_d$ or part of $\mathcal{T}$. There are three cases: a) one leaf was not part of $\mathcal{E}_d$ or $\mathcal{T}$. The new edge can be added to $\mathcal{E}_d$ giving case (i) of the Lemma. b) one leaf was part of $\mathcal{T}$. The new edge can be added to $\mathcal{E}_d$ and $|\mathcal{T}|$ is reduced by 1, giving case (i) of the Lemma. (3) all leaves are part of $\mathcal{E}_d$, but since $|\mathcal{E}_d| \leq u$, at least two suspected leaves are already connected in $\mathcal{E}_d$. This allows to either replace one edge in $\mathcal{E}_d$ by two new ones, or add a new node to $\mathcal{T}$.  □

*CT3 – Eventually no false suspicions.* Lemma 6 proves C3, ensuring that suspicions which are not filtered after GST are useful.

**Lemma 6.** After GST, any suspicion between two correct replicas is filtered in the SuspicionMonitor.

*Proof.* We need to show that the timeout requirements TR1 - TR3 hold for OptiTree. The Lemma then follows from Lemma 3.

We show that TR1 - TR3 hold by showing how the timeouts (or durations) for the different messages are set. The different messages in the tree are:

- Propose messages from the root $R$ to the intermediate nodes.
- Forwarded Propose messages from intermediate nodes to the leaf nodes.
- Vote messages from the leaf nodes to the intermediate nodes.
- Aggregated Vote messages from the intermediate nodes to the root.

The message timeout $d_{m_p}$ for a Propose message from the root is set to the latency of the used link, ensuring TR1.

Forwarded Propose messages are sent directly after receiving the Propose message and Vote messages are sent directly after receiving a forwarded Propose. The message timeout for these messages is set by simply adding the link latency to the timeout of the previous message. Thus TR2 holds, since for a Forwarded Propose message, $m'$ is the Propose message, and for a Vote message, $m'$ is the Forwarded Propose message.

An intermediate nodes $I$ sends the Aggregated Vote message after receiving Vote messages from all its children, or after waiting for the timeout for all these Vote messages. Thus, the timeout for the Aggregated Vote message is set according to TR2, where $m'$ is the Vote message with largest timeout.

Finally, we note that $d_{rnd}$ is set according to TR3., where *quorums* is the set $\mathcal{M}_k$ defined in the score function from Definition 1 in §6.3.

We note that in OptiTree, we actually omit raising suspicions on Forwarded Propose messages. The reason for this is that, if the Vote message is delayed and the Leaf node is suspected by the intermediate node, then the leaf node will anyway reciprocate the suspicion. The intermediate node cannot omit the suspicion, if the Vote is delayed, without itself being suspected by the root. Additionally, when calculating $d_m$ for Vote messages, we set the timeout based on when the Forwarded Propose was sent, rather than the Proposal timestamp.  □

*CT4 – Limiting the number of reconfigurations.* To prove our main theorem, about the number of reconfigurations needed to find a correct tree, we consider new subsets $\mathcal{E}_d^n \subset \mathcal{E}_d$, $\mathcal{T}^n \subset \mathcal{T}$, and $C^n \subset C$ which are added to the respective sets after GST. Lemma 7 shows that the size of these data-structures reflects the number of faulty replicas removed from the candidate set. Lemma 5 shows that for any two failed trees, the size of these data-structures increases by at least one. Lemmas 9 and 10 show that other mechanisms do not reduce the size. We denote the total size of these structures as $t^n = |C^n| + |\mathcal{E}_d^n| + |\mathcal{T}^n|$. Remember that $t \leq f$ is the number of actually faulty replicas in the system.

**Lemma 7.** If $t^n = t$, then a working configuration is found.

*Proof.* After GST, no two correct replicas raise suspicions on each other and all suspicions on correct replicas are reciprocated. Therefore, if a replica was added to $C^n$ it is indeed faulty. Further, if a new edge was added to $\mathcal{G}$, then at least one of the adjacent replicas is faulty. Thus, for every edge in $\mathcal{E}_d^n$, at least one of the adjacent nodes is faulty, and for each triangle build from a replica in $\mathcal{T}^n$ and an edge in $\mathcal{E}_d^n$, at least two of the adjacent replicas are faulty. Finally, for a triangle in $\mathcal{G}$ containing a replica in $\mathcal{T}^n$ and an edge in $\mathcal{E}_d$ but not in $\mathcal{E}_d^n$, at least one of the adjacent replicas is faulty. Thus $t^n$ many faulty replicas have been removed from the candidate set for internal nodes. □

**Lemma 8.** If a tree failed after GST, either: (i) $|\mathcal{E}_d^n|$ increases, or (ii) $|\mathcal{T}^n|$ increases and $|\mathcal{E}_d^n|$ stays unchanged. Also, either $t'$ increases, or $t'$ stays constant and $\mathcal{T}^n$ decreases.

This Lemma is a variation of Lemma 5 and the proof follows the same schema.

*Proof.* All suspicions are added to the graph $\mathcal{G}$. In case (1) of Lemma 2, the new suspicion can be added to $\mathcal{E}_d^n$ and $\mathcal{T}$ remains unchanged, increasing $t'$. In case (2), $u+1$ edges are added to $\mathcal{G}$. These new edges include $u+1$ different leaves from the failed tree. These leaves, can be already adjacent to an edge in $\mathcal{E}_d$ or part of $\mathcal{T}$. There are three cases: a) one leaf was not part of $\mathcal{E}_d$ or $\mathcal{T}$. The new edge can be added to $\mathcal{E}_d^n$ giving case (i) of the Lemma. b) one leaf was part of $\mathcal{T}$. The new edge can be added to $\mathcal{E}_d^n$ and $|\mathcal{T}|$, possibly also $|\mathcal{T}^n|$ is reduced by 1, giving case (ii) of the Lemma. In this case, $t'$ stays constant. (3) all leaves are part of $\mathcal{E}_d$, but since $|\mathcal{E}_d| \leq u$, at least two suspected leaves are already connected in $\mathcal{E}_d$. This allows to either replace one edge in $\mathcal{E}_d$ by two in $\mathcal{E}_d^n$, or add a new node to $\mathcal{T}^n$. Also here $t'$ increases. □

If a new suspicion is raised between two nodes in $G$, it is first treated as a two way suspicion and added to $\mathcal{G}$. If no

reciprocation happens during the next $f+1$ views, then the suspected replica is added to $C$. The next Lemma proves that these changes do not decrease $t'$.

**Lemma 9.** If nodes are added to $C$ due to not reciprocated suspicions, this does not change $t'$, and does not increase $\mathcal{T}^n$.

*Proof.* Clearly, if an edge is removed from $\mathcal{E}_d^n$, then a replica is added to $C^n$. In some cases the removal removing an edge from $\mathcal{E}_d$ causes that a replica is removed from $\mathcal{T}^n$ and instead, a new edge is added to $\mathcal{E}_d^n$. Thus $t'$ does not change. □

All suspicions added after GST include at least one faulty replica. Thus these suspicions leave an independent set of $n-f$ correct replicas. However, suspicions made before GST may be successively removed after GST.

**Lemma 10.** If suspicions made before GST are discarded, removing replicas from $C$, or removing edges from $\mathcal{G}$, $t'$ does not decrease, and $\mathcal{T}^n$ does not increase.

*Proof.* Clearly this does not reduce $C^n$ and $\mathcal{E}_d^n$. However, if an edge in $\mathcal{E}_d$ is removed, it can be that a replica in $\mathcal{T}^n$ is no longer part of a triangle. However, the replica in $\mathcal{T}^n$ is adjacent to an edge added after GST. This edge can now be added to $\mathcal{E}_d^n$, leaving $t'$ unchanged. □

**Theorem D.2.** *After GST, OptiTree finds a working configuration after at most $2t$ reconfigurations.*

*Proof.* Based on Lemma 7, we only have to show that after $2t$ reconfigurations, $t' = t$. According to Lemma 5, after a reconfiguration, either $t'$ increases, or $t'$ stays constant and $\mathcal{T}^n$ decreases. Since $\mathcal{T}^n$ can only decrease by $k$ after $\mathcal{T}^n$ and $t'$ have been increased by $k$. Thus at most $t'$ reconfigurations can leave $t'$ unchanged. □