Proof-of-Execution: Low-latency Consensus via Speculative Execution

JELLE HELLINGS, McMaster University, Canada
SUYASH GUPTA, University of Oregon, USA
SAJJAD RAHNAMA, University of California, Davis, USA
JUNCHAO CHEN, University of California, Davis, USA
CELINE SANA, McMaster University, Canada
MOHAMMAD SADOGHI, University of California, Davis, USA

The introduction of Bitcoin fueled the development of blockchain-based resilient data management systems that are resilient against failures, enable federated data management, and can support data provenance. The key factor determining the performance of such resilient data management systems is the consensus protocol used by the system to replicate client transactions among all participants. Unfortunately, existing high-throughput consensus protocols are costly and impose significant latency on transaction processing, which rules out their usage in responsive high-performance data management systems.

In this work, we improve on this situation by introducing the *Proof-of-Execution consensus protocol* (PoE), a consensus protocol designed for high-performance low-latency resilient data management. PoE introduces *speculative execution*, which minimizes latency by starting execution before consensus is reached, and PoE introduces *proof-of-executions* to guarantee successful execution to clients. Furthermore, PoE introduces a single-round check-commit protocol to reduce the overall communication costs of consensus. Hence, we believe that PoE is a promising step towards flexible general-purpose low-latency resilient data management systems.

CCS Concepts: • **Information systems → Distributed database transactions**.

Additional Key Words and Phrases: Consensus, Speculative execution, Byzantine fault tolerance, Low-latency transaction processing

ACM Reference Format:
Jelle Hellings, Suyash Gupta, Sajjad Rahnama, Junchao Chen, Celine Sana, and Mohammad Sadoghi. 2018. Proof-of-Execution: Low-latency Consensus via Speculative Execution. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 49 pages. https://doi.org/XXXXXXX.XXXXXXX

An extended abstract of this work appeared at the 24th International Conference on Extending Database Technology (EDBT 2021) [26].

Authors' Contact Information: Jelle Hellings, Department of Computing and Software, McMaster University, Hamilton, ON, Canada; Suyash Gupta, RISELab, Department of Electrical Engineering and Computer Science, University of Oregon, Eugene, OR, USA; Sajjad Rahnama, Exploratory Systems Lab, Department of Computer Science, University of California, Davis, Davis, CA, USA; Junchao Chen, Exploratory Systems Lab, Department of Computer Science, University of California, Davis, Davis, CA, USA; Celine Sana, Department of Computing and Software, McMaster University, Hamilton, ON, Canada; Mohammad Sadoghi, Exploratory Systems Lab, Department of Computer Science, University of California, Davis, Davis, CA, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The introduction of the cryptocurrency Bitcoin [37] marked the first wide-spread deployment of a *permissionless blockchain*. The emergence of Bitcoin and other blockchains has fueled the development of new resilient data management systems [4–6, 13, 17, 24, 25, 28, 31, 32, 38, 39, 42, 46]. These new systems are attractive for the database community, as they can be used to provide data management systems that are resilient against failures, enable cooperative (federated) data management with many independent parties, and can support data provenance. Due to these qualities, interest in blockchains is widespread and includes applications in health care, IoT, finance, agriculture, and the governance of supply chains for fraud-prone commodities (e.g., such as hardwood and fish) [20, 29, 30, 35, 41, 47, 50].

At their core, blockchain systems are distributed systems in which each participating replica maintains a copy of a ledger that stores an append-only list of all transactions requested by users and executed by the system [22]. This ledger is constructed and stored in a tamper-proof manner: changes (e.g., appending new transactions) are made via a *consensus protocol*, which will only allow changes that are supported by the majority of all participants, ruling out malicious changes (e.g., overwriting existing operations) by a minority of faulty participants [22, 45]. These *consensus protocols* can be seen as generalizations of the well-known two-phase commit [21] and three-phase commit [44] protocols (as uses in traditional replicated databases) toward dealing gracefully with *replica failures* and even *malicious behavior*. As the ledger is replicated over and maintained by all participating replicas, it is highly *resilient* and will survive even if individual participants fail.

The key factor determining the performance of blockchain-based systems is the choice of *consensus protocol* [10, 22, 23, 27]: the consensus protocol determines the *throughput* of the system, as the consensus protocol determines the speed by which transactions are replicated and appended to the ledger of each replica, and the *latency* clients observe on their requested transactions, as the operations necessary to reach consensus on these transactions determine the minimum time it takes for individual replicas to execute the requested transactions and inform clients of the result.

Unfortunately, existing consensus protocols typically focus on either providing high throughput or low latency, thereby failing to provide the combination of high throughput and low latency required for responsive high-performance data management systems. First, we observe that the Proof-of-Work style consensus protocols of permissionless blockchains such as Bitcoin and Ethereum suffer from high costs, very low throughput, and very high latency, making such permissionless designs impractical for high-performance data management [15, 40, 49]. Permissioned blockchains, e.g., those based on *Byzantine Fault Tolerance* (BFT) consensus, are more suitable for high-performance data management: fine-tuned permissioned systems can easily process up-to-hundreds-of-thousands transactions per second, this even in wide-area (Internet) deployments [10, 22, 25, 27]. Still, even the best permissioned consensus protocols cannot provide the low latency we are looking for, as all reliable consensus protocols require three-or-more subsequent rounds of internal communication before requests can be executed and clients can be informed.

To further unlock the development of new resilient data management systems, we designed the *Proof-of-Execution consensus protocol* (PoE), a *novel* consensus protocol that is able to provide high throughput, while minimizing client latency. At the core of PoE are two innovative techniques:

(1) PoE introduces *speculative execution*: PoE executes transactions requested by clients and informs clients of the result *before* consensus is reached on these transactions, while providing the clients a *proof-of-execution* that guarantees that speculatively-executed transactions will eventually reach consensus; and
(2) PoE introduces the *check-commit* protocol, a decentralized single-round protocol that, under normal conditions, can commit consensus decisions for which a *proof-of-execution* exists

and can replicate such decisions among all replicas without relying on specific replicas and without requiring several rounds of communication.

By combining these innovative techniques, PoE only imposes *two* rounds of communication on a consensus step before execution can commence and the client can be informed, while only requiring three rounds of communication to complete a consensus step in the normal case. Furthermore, the design of PoE is flexible and allows for optimizations that further balance communication costs, transaction latency, and recovery complexity. PoE can, for example, use digests to reduce communication costs (at the cost of more-complex recovery paths), use threshold signatures to further reduce communication costs (at the cost of higher latency), use message authentication codes to reduce computation costs (at the cost of more-complex recovery paths), and use out-of-order processing to significantly improve throughput (at the cost of higher resource usage).

This paper extends the work presented in [26], introducing several enhancements and novel contributions. We propose a single-round check-commit protocol that effectively identifies replicas left in the dark, enabling them to recover any lost transactions. Additionally, we introduce a proof-of-commit mechanism to ensure PoE is able to provide client services whenever communication is reliable. Moreover, we offer a comprehensive analysis and provide rigorous proofs for the check-commit and view-change protocols, that are designed to guarantee that all committed transactions will ultimately be recovered and consistently committed by all non-faulty replicas. Furthermore, we have re-implemented the protocol and conducted new evaluations using Apache ResilientDB, an open-source blockchain platform with high-performance capabilities, while the original evaluation was done on a work-in-progress research prototype.

In specific, we make the following contributions:

- In Section 3 we introduce the concept of *speculative execution* and formalize its usage in a client-oriented system that processes transactions via consensus.
- In Section 4, we provide an in-depth description of all parts of PoE. In specific:
  - Section 4.1 describes the normal-case operations that are optimized for high-performance low-latency transaction processing utilizing speculative execution and proof-of-execution;
  - Section 4.2 describes the situations in which the normal-case of PoE can fail and the impact this has on the state of individual replicas;
  - Section 4.3 introduces the novel single-round check-commit protocol that allows PoE replicas to recover from minor failures without interrupting the normal-case operations;
  - Section 4.4 introduces the view-change protocol that allows PoE replicas to recover from major failures (including network failures) without invalidating any transactions that have received a proof-of-execution;
  - Section 4.5 proves the correctness of *all parts* of PoE, showing that PoE provides consensus and maintains all speculatively-executed transactions that have received a proof-of-execution. Due to the level of detail (which includes all modes of operations), this proof of correctness is a major contribution in itself and can be used as a stepping stone in the analysis of other primary-backup consensus protocols; and
  - Section 4.6 proves that PoE provides services to clients. In the normal case, PoE does so by providing the client with a *proof-of-execution*, which can be provided with low latency. In the case of failures, PoE can always fall back to a *proof-of-commit*, which takes an additional round of communication to establish.
- In Section 5, we take an in-depth look at the complexity of PoE, we introduce PoE variants that use digests, threshold signatures, or message authentication codes to reduce complexity, and we show that PoE and all its variants can utilize out-of-order processing to maximize performance.

Protocol	Latency[a]		Message Complexity[b]		Environment[c]	Remarks
	Before Execution	Total	Total	Per Replica (max)		
PoE	2	3	$\mathbf{nC} + 2\mathbf{n}^2$	$\mathbf{nC} + \mathbf{n}$	Asynchronous (recovery) Partial Synchrony (progress)	Speculative execution.
Linear-PoE	3	5	$\mathbf{nC} + 4\mathbf{n}$	$\mathbf{nC} + \mathbf{n}$	Asynchronous (recovery) Partial Synchrony (progress)	Speculative execution.
Pbft	3	4	$\mathbf{nC} + 3\mathbf{n}^2$	$\mathbf{nC} + 2\mathbf{n}$	Asynchronous (recovery) Partial Synchrony (progress)	
Zyzzyva[d]	1	1	$\mathbf{nC}$	$\mathbf{nC}$	Asynchronous (recovery) Partial Synchrony (progress)	Requires reliable clients; has vulnerabilities [2, 3].
Sbft[e]	4	$5 + 1_{CP}$	$\mathbf{nC} + 4\mathbf{n} + \mathbf{n}^2{}_{CP}$	$\mathbf{nC} + 3$	Asynchronous (recovery) Partial Synchrony (progress)	
HotStuff[f]	7	8	$\mathbf{nC} + 3\mathbf{n}$	$\mathbf{nC} + 3$	Partial synchrony (progress and recovery)	No out-of-order processing.
MinBFT	2	2	$\mathbf{nC} + \mathbf{n}^2$	$\mathbf{nC} + \mathbf{n}$	Reliable communication (progress and recovery)	Requires trusted hardware; can tolerate more faulty replicas.

[a]The latency is measured in terms of the number of communication rounds the protocol performs *after* the protocol starts processing a client request. [b]For readability, we used a simplified notation for the message complexity by omitting off-by-one terms. For example, PoE requires a total of $(\mathbf{n} - 1)\mathbf{C} + 2\mathbf{n}(\mathbf{n} - 1)$ messages. [c]The *environment* specifies in which environment the protocol can operate. This does not mean that the protocol will be able to make *progress* (new consensus decisions are made), however. Indeed, all these protocols require sufficiently reliable communication to guarantee *progress*. Only the protocols labeled with *asynchronous (recovery)* can recover from any number of periods in which communication is not sufficiently reliable. [d]This entry only reflects the *optimistic* fast path of Zyzzyva, which cannot deal with any replica failures. The complexity of the slow path of Zyzzyva is akin that of Pbft. [e]This entry only reflects the *optimistic* fast path of Sbft, which cannot deal with any replica failures when replicas are either non-faulty or malicious. Furthermore, Sbft utilizes a checkpoint protocol akin to the one used by Pbft. As no explicit description of this checkpoint protocol is provided in the original Sbft paper, we have used the cost of the Pbft checkpoint protocol (complexity-terms related to the checkpoint protocol are subscripted with cp). [f]In the standard configuration of HotStuff, phases of up-to-four consensus decisions are overlapped. Even with this overlapping, each consensus decision has to go through four all-to-one-to-all communication phases. The costs in this table reflect the communication necessary to complete these four phases.

Fig. 1. A cost comparison of the normal-case operations of PoE and other consensus protocols when reaching consensus among $\mathbf{n}$ replicas on a client request with a size bounded by $\mathbf{C}$. We refer to Appendix A for an in-depth analysis and breakdown of the details in this table.

- In Section 5.4 we introduce Linear-PoE, a variant of PoE that uses threshold signatures to make the normal-case and the check-commit protocol fully linear. Central in this variant is a novel *Linear Check-Commit protocol* that utilizes aggregator rotation, aggregated (multi-round) check-commits, and recovery certificates to ensure a single-round decentralized design with only linear communication costs.
- Finally, in Section 6, we perform an in-depth evaluation of PoE in comparison with other frequently-used consensus protocols.

Furthermore, Section 2 introduces the notation used throughout this paper and Section 7 concludes on our findings.

A summary of the comparison between PoE[1] and other high-performance consensus protocols can be found in Figure 1. As one can see, PoE outperforms other high-performance consensus protocols, as PoE combines the lowest latency with low communication costs. Furthermore, PoE provides high resilience, as it can operate in fully asynchronous environments without any further assumptions on replicas or clients.

[1]PoE is renamed from [26] due to the double-blind policy.

2 Preliminaries

System Model. We model a *system* as a tuple $(\mathfrak{R}, \mathfrak{C})$, in which $\mathfrak{R}$ is a set of *replicas* and $\mathfrak{C}$ is a set of *clients*. We assign each replica $\mathrm{R} \in \mathfrak{R}$ a unique identifier $\mathrm{id}(\mathrm{R})$ with $0 \leq \mathrm{id}(\mathrm{R}) < |\mathfrak{R}|$. We write $\mathcal{F} \subseteq \mathfrak{R}$ to denote the set of *Byzantine replicas* that can behave in arbitrary, possibly coordinated and malicious, ways.

We assume that non-faulty replicas behave in accordance to the protocols they are executing. We do not make any assumptions on clients: all clients can be malicious without affecting PoE. We write $\mathbf{n} = |\mathfrak{R}|$, $\mathbf{f} = |\mathcal{F}|$, and $\mathbf{nf} = \mathbf{n} - \mathbf{f}$ to denote the number of replicas, faulty replicas, and non-faulty replicas, respectively.

Communication. We assume *authenticated communication*: Byzantine replicas are able to impersonate each other, but replicas cannot impersonate non-faulty replicas. Authenticated communication is a minimal requirement to deal with Byzantine behavior [22]. To enforce authenticated communication and simplify presentation, we assume that all messages are *digitally signed* (e.g., via public-key cryptography) [33]: every replica and every client $z \in (\mathfrak{R} \cup \mathfrak{C})$ can *sign* arbitrary messages m, resulting in a certificate $\langle m \rangle_z$. These certificates are non-forgeable and can be constructed only if z cooperates in constructing them. Based on only the certificate $\langle m \rangle_z$, anyone can verify that m was originally supported by z. We refer to Section 5.5 for a discussion on how to eliminate digital signatures from all messages used between replicas in a system.

Consensus. A *consensus protocol* [22, 45] coordinates decision making among the replicas $\mathfrak{R}$ of a system by providing a reliable ordered replication of *decisions*. To do so, consensus protocols provide the following guarantees:

Termination if non-faulty replica $\mathrm{R} \in \mathfrak{R}$ makes a ρ-th decision, then all non-faulty replicas $\mathrm{Q} \in \mathfrak{R}$ will make a ρ-th decision;

Non-Divergence if non-faulty replicas $\mathrm{R}_1, \mathrm{R}_2 \in \mathfrak{R}$ make ρ-th decisions D_1 and D_2, respectively, then $D_1 = D_2$ (they make the same ρ-th decisions); and

Non-Triviality whenever a non-faulty replica $\mathrm{R} \in \mathfrak{R}$ learns that a decision D needs to be made, then replica R can force consensus on D.

In this work, we assume that each decision represents one or more client transactions. Hence, in practice, the *non-triviality* guarantee simply specifies that replicas can force processing of new client requests whenever clients are requesting execution of transactions.

Consensus cannot be solved in environments in which communication is *asynchronous* (e.g., when messages can get lost or be arbitrarily delayed) [18]. Even though practical networks are reliable most of the time, they also have periods of failure during which they behave asynchronous. One way to deal with this is by providing *weak consensus*: weak consensus always guarantees non-divergence, while only guaranteeing termination and non-triviality in periods of *reliable communication* (during which messages are delivered within some unknown bounded delay) [10]. We assume $\mathbf{n} > 3\mathbf{f}$ ($\mathbf{nf} = \mathbf{n} - \mathbf{f} > 2\mathbf{f}$), a minimal requirement to provide consensus in an asynchronous environment [9, 16, 22].

3 From Consensus to Client Services

In the previous section, we defined consensus. The definition of consensus does not specify how one builds an effective service that clients can use for the execution of their transactions, however. Next, we take a look at how consensus-based systems can provide such client services.

Traditional Execution. First, we consider traditional consensus-based systems that provide client services. Consider a client γ requesting the execution of some transaction τ. In traditional systems,

replicas will *execute* τ as the ρ-th transaction and inform γ of the outcome *after they decided* upon τ (using consensus) as the ρ-th decision and after executing all transactions decided upon by preceding decisions.

Example 3.1. Consider a deployment of the *Practical Byzantine Fault Tolerance consensus protocol* (Pbft) [10, 22]. Under normal conditions, Pbft operates via a *primary-backup* design in which a designated replica (the primary) is responsible for proposing client transactions to all other replicas (the backups). The primary does so via a `PrePrepare` message. Next, all replicas exchange their local state to determine whether the primary properly proposed a decision. To do so, all replicas participate in two phases of all-to-all communication.

In the *first phase*, all (non-faulty) replicas that receive a proposal via `PrePrepare` message m broadcast a message `Prepare`(m). Then each replica R waits until it receives `Prepare` messages identical to `Prepare`(m) from at-least $\mathbf{nf}$ distinct replicas. After receiving these $\mathbf{nf}$ messages, the proposal m is *prepared*.

Notice that at-least $\mathbf{nf} - \mathbf{f} = \mathbf{n} - 2\mathbf{f}$ `Prepare` messages received by R are sent by non-faulty replicas. Hence, there are at-most $\mathbf{f}$ non-faulty replicas that did not participate in preparing m. As such, for any other `PrePrepare` message m, replicas will only be able to collect up-to $2\mathbf{f} < \mathbf{nf}$ `Prepare` messages, guaranteeing that only the message m will be prepared at non-faulty replicas.

In the *second phase*, all (non-faulty) replicas that prepared m broadcast a message `Commit`(m). Then each replica R waits until it receives `Commit` messages identical to `Commit`(m) from at-least $\mathbf{nf}$ distinct replicas. After receiving these $\mathbf{nf}$ messages, the proposal m is *committed*, after which R decides m (and, hence, executes the client transaction proposed by m).

In Pbft, a replica R *commits* m when it has a guarantee that m can always be recovered from the state of at-most $\mathbf{nf}$ replicas (e.g., all non-faulty replicas). To see this, consider the at-least $\mathbf{nf}$ `Commit` messages received by R due to which R commits m. Of these messages, at-least $\mathbf{nf} - \mathbf{f} = \mathbf{n} - 2\mathbf{f}$ are sent by non-faulty replicas. Now consider *any replica* Q trying to recover based on the state of any set S of at-least $\mathbf{nf}$ replicas. At-least $\mathbf{nf} - \mathbf{f} = \mathbf{n} - 2\mathbf{f}$ of the replicas in S are non-faulty replicas. Let C be the $\mathbf{nf}$ replicas from which R received `Commit` messages, let $D = C - \mathcal{F}$ be the non-faulty replicas in C and let $T = S - \mathcal{F}$ be the non-faulty replicas in S. We have $|D| > \mathbf{n} - 2\mathbf{f}$ and $|S| > \mathbf{n} - 2\mathbf{f}$. As we assumed $\mathbf{n} > 3\mathbf{f}$, we must have $(D \cap T) \neq \emptyset$ (as otherwise, $|D| + |T| \geq 2(\mathbf{n} - 2\mathbf{f})$ and $|D| + |T| \leq \mathbf{nf} = \mathbf{n} - \mathbf{f}$ must hold, which would imply $\mathbf{n} \leq 3\mathbf{f}$). As such, Q will be able to recover m from the state of any replica in $D \cap T$. Hence, after a non-faulty replica commits m, there is the guarantee that m can be recovered from the transactions prepared by any set of $\mathbf{nf}$ replicas.

We have sketched this working of Pbft in Figure 2. Besides the normal-case operations of Pbft outlined above, Pbft also has two recovery mechanisms to recover from primary failures and network failures, namely a checkpoint protocol and a view-change protocol. Crucially, these recovery mechanisms assure that *all* transactions that are ever decided (committed) by a non-faulty replica will eventually be recovered and committed by all non-faulty replicas whenever communication becomes sufficiently reliable.

We assume that execution of τ is deterministic: executing τ at any non-faulty replica yields identical outputs when executed upon identical inputs. Using this deterministic nature, the consensus-coordinated replication and execution of transactions will implement a fully-replicated system in which all non-faulty replicas will manage a copy of the same data. Under these assumptions, it is straightforward to deliver *client services*:

(1) To assure that client transactions get executed, a client simply needs to send their transaction τ to any non-faulty replica (whom can then use the non-triviality property of consensus to force a decision on τ).

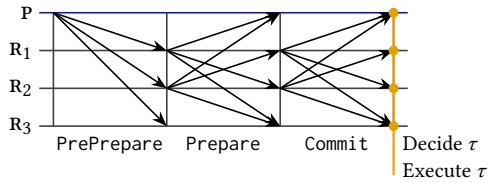

Fig. 2. A schematic representation of the normal-case of Pbft: the primary P proposes transaction τ to all replicas via a PrePrepare message. Next, replicas commit to τ via a two-phase all-to-all message exchange. In this example, replica R_3 is faulty and does not participate.

(2) To observe the result of execution of a transaction τ, the client simply awaits until it receives a single response by any non-faulty replica. As non-faulty replicas execute after they decided on τ, non-divergence and termination guarantee that *all* non-faulty replicas will eventually do the same.

When using traditional execution, a client can effectively *detect* whether it received a single response by any non-faulty replica after it received identical responses from at-least $\mathbf{f}+1$ distinct replicas (as at-most $\mathbf{f}$ of those responses can originate from faulty replicas). Using the assumption that execution is deterministic and that all $\mathbf{n}-\mathbf{f} > \mathbf{f}$ non-faulty replicas will eventually execute τ, all non-faulty replicas will eventually inform the client with the same identical result. Hence, independent of the behavior of faulty replicas, the client will receive at-least $\mathbf{f}+1$ identical responses and is able to reliably derive execution results.

Speculative Execution. Traditional execution assures that it is easy to reason about the operations of a system, both for its replicas (which have strong guarantees during execution) and for the clients (whom can easily derive execution results). This ease-of-use comes at a significant cost: by only executing transactions until replicas are able to make consensus decisions, we significantly increase the latency clients will observe on their requests, even when the system is operating entirely correctly.

Example 3.2. Consider again the operations of Pbft with traditional execution, as outlined in Example 3.1. Let τ be a transaction requested by some client and let δ be the message delay. At t, the primary receives τ and is able to propose τ. Assuming that the bandwidth and processing time required to send, receive, and process a message is negligible, these PrePrepare proposals will arrive after $t+\delta$ at all other replicas, whom then all broadcast Prepare messages. All these Prepare messages will arrive after $t+2\delta$. Only then are all replicas able to broadcast Commit messages, which will arrive after $t+3\delta$. Hence, execution will only happen 3δ after the initial proposal, and only after execution will the client be notified of any outcome.

Modern variants of Pbft such as Sbft [19] and HotStuff [52] use *threshold signatures* to replace some phases of all-to-all communication with a quadratic message complexity (e.g., the prepare-phase and the commit-phase), to subphases of all-to-one and one-to-all communication with a linear message complexity each. Such implementations typically trade computational complexity and latency for lower communication costs and will result in Pbft variants with much higher client latency, e.g., execution in Sbft happens after 4δ and execution in HotStuff happens after 7δ.

The many phases before execution in these consensus protocols is especially noticeable in practical deployments of consensus: to maximize resilience against disruptions at any location, individual replicas need to be spread out over a wide-area network. Due to this spread-out nature, the message delay will be high and a message delay of $15\,\text{ms} \leq \delta \leq 200\,\text{ms}$ is not uncommon [14, 25].

As an alternative to traditional execution, we propose *speculative execution*: replicas will *execute* τ as the ρ-th transaction and inform γ of the outcome *before they decided* upon τ as the ρ-th decision (but still after executing all preceding transactions). As replicas execute transactions before a final consensus decision is made, this introduces two challenges:

(1) A non-faulty replica R can execute τ as the ρ-th transaction only to later make an ρ-th decision for another transaction τ'. In this case, R needs a way to *rollback* the execution of τ and replace it with an execution of τ'.
(2) As non-faulty replicas can rollback their execution, clients can no longer observe the result of execution of a transaction τ via a single response of any non-faulty replica (or via identical responses from $\mathbf{f} + 1$ distinct replicas).

Even with these challenges, speculative execution is worthwhile: when the system operates correctly, speculative execution can greatly reduce the latency clients perceive upon their requests, this especially in systems utilizing threshold signatures. In the next section, we introduce the *Proof-of-Execution consensus protocol* (PoE) that utilizes speculative execution and shows methods to overcome both these challenges.

Remark 3.3. We assume that one can always rollback transaction execution. Without any knowledge on the type of transactions executed, the *brute-force approach* to implement rollbacks is to snapshot the system state before each transaction execution and restore previous snapshots to rollback execution. Obviously, this approach is impractical in almost all real-world cases.

In practice, given a transaction τ, one can usually derive an undo-transaction τ_{undo} whose execution reverses all side effects made by the execution of τ. Take, for example, the following transaction that conditionally transfers funds:

$$\begin{aligned}
\tau = \textbf{if } &B(\mathit{Wichian}) > \$1200 \textbf{ and } B(\mathit{Ya\text{-}Ting}) > \$700 \textbf{ then}\\
&B(\mathit{Wichian}), B(\mathit{Ximena}) := B(\mathit{Wichian}) - \$1100, B(\mathit{Ximena}) + \$1100;\\
&B(\mathit{Ya\text{-}Ting}), B(\mathit{Zoran}) := B(\mathit{Ya\text{-}Ting}) - \$600, B(\mathit{Zoran}) + \$600.\\
\textbf{else if } &B(\mathit{Wichian}) > \$700 \textbf{ and } B(\mathit{Ya\text{-}Ting}) > \$1200 \textbf{ then}\\
&B(\mathit{Wichian}), B(\mathit{Zoran}) := B(\mathit{Wichian}) - \$600, B(\mathit{Zoran}) + \$600;\\
&B(\mathit{Ya\text{-}Ting}), B(\mathit{Ximena}) := B(\mathit{Ya\text{-}Ting}) - \$1100, B(\mathit{Ximena}) + \$1100.\\
\textbf{end if}&
\end{aligned}$$

In the above, B stands for *account balance*. Executing transaction τ has three possible outcomes:

(1) The **if**-path is taken, in which case τ_{undo} is

$$\begin{aligned}
\tau_{\text{undo}} = &B(\mathit{Wichian}), B(\mathit{Ximena}) := B(\mathit{Wichian}) + \$1100, B(\mathit{Ximena}) - \$1100;\\
&B(\mathit{Ya\text{-}Ting}), B(\mathit{Zoran}) := B(\mathit{Ya\text{-}Ting}) + \$600, B(\mathit{Zoran}) - \$600.
\end{aligned}$$

(2) The **else if**-path is taken, in which case τ_{undo} is

$$\begin{aligned}
\tau_{\text{undo}} = &B(\mathit{Wichian}), B(\mathit{Zoran}) := B(\mathit{Wichian}) + \$600, B(\mathit{Zoran}) - \$600;\\
&B(\mathit{Ya\text{-}Ting}), B(\mathit{Ximena}) := B(\mathit{Ya\text{-}Ting}) + \$1100, B(\mathit{Ximena}) - \$1100.
\end{aligned}$$

(3) Neither path is taken, in which case τ_{undo} does nothing.

The complexity of rolling back transaction execution depends completely on the type of transactions used by the application running on top of a resilient data management systems.

4 Consensus via Proof-of-Execution

The *Proof-of-Execution consensus protocol* (PoE) shares the primary-backup design of PBFT and utilizes speculative execution to minimize the client latency in the normal case (when the primary is non-faulty and communication is reliable). To simplify presentation, we will present a non-optimized version of PoE, after which we take an in-depth look at optimizing the complexity of PoE in Section 5.

Our presentation of PoE is broken-up in six parts. First, in Section 4.1, we describe the *normal-case protocol* that is used by the primary to propose consensus decision. Next, in Section 4.2, we look at how replica and network failures can disrupt the normal-case protocol. Third, in Section 4.3, we describe the *check-commit protocol* to deal with failures that do not disrupt the normal-case. Then, in Section 4.4, we describe the *view-change protocol* to deal with failures that disrupt the normal-case. After presenting these three protocols in full detail, we will prove in Section 4.5 that PoE provides *weak consensus* and we will prove in Section 4.6 that PoE provides reliable service to clients.

4.1 The Normal-Case Protocol

PoE operates in views and in view v the replica P with $\mathrm{id}(\mathrm{P}) = v \bmod \mathbf{n}$ is the *primary* that coordinates the normal-case protocol.

Consider a client γ that wants to request transaction τ. For now, we assume that γ knows that P is the current primary, we refer to Section 4.6 for the case in which the primary is unknown to γ. To prevent any party to forge requests by client γ, the client γ signs any transactions it wants to request before sending them to the current primary. Hence, to request τ, the client will send $\langle \tau \rangle_\gamma$ to primary P.

After primary P receives $\langle \tau \rangle_\gamma$, a transaction τ requested and signed by client γ, it can propose τ. To *propose τ as the ρ-th transaction*, the primary broadcasts a message $\texttt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$ to all other replicas.

Next, all replicas exchange their local state to determine whether the primary properly proposed a decision. In PoE, the replicas do so in *one* phase of all-to-all communication. Upon arrival of the *first* proposal for round ρ of view v via some `Propose` message $m = \texttt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$, each (non-faulty) replica R that received m will enter the *prepare phase* for m. As the first step of the prepare phase, R broadcasts a message $\texttt{Prepare}(m)$. Next, R waits until it receives `Prepare` messages identical to $\texttt{Propose}(m)$ from at-least **nf** distinct replicas (here, R can include the messages $\texttt{Prepare}(m)$ broadcasted by itself and by the primary). After receiving these **nf** messages, the proposal m is *prepared*. As a proof of this prepared state for m, R stores a *prepared certificate* $\mathbb{P}(m)$ consisting of these **nf** `Prepare` messages.

When replica R prepared proposal m, it schedules τ for speculative execution. Whenever all preceding transactions are already executed, τ will be executed by R yielding some result r. Next, R will inform the client γ via an $\texttt{Inform}(\langle \tau \rangle_\gamma, v, \rho, r)$ of the outcome. Finally, client γ waits for a *proof-of-execution* for $\langle \tau \rangle_\gamma$ consisting of identical $\texttt{Inform}(\langle \tau \rangle_\gamma, v, \rho, r)$ messages from **nf** distinct replicas. When γ receives this proof-of-execution, it considers τ executed. Note that the client-role of PoE is almost identical to the client-role of PBFT outlined in Example 3.1, the only difference being that the client requires $\mathbf{n} - \mathbf{f}$ `Inform` responses instead of $\mathbf{f} + 1$. As we shall prove later on, the existence of this (v, ρ)-proof-of-execution for $\langle \tau \rangle_\gamma$ guarantees that the *speculative* execution of τ will be preserved by all replicas (and will never rollback).

The pseudo-code for this normal-case protocol can be found in Figure 3 and an illustration of the working of this protocol can be found in Figure 4.

The correctness of PoE is based on the following properties of the normal-case protocol:

Client role (used by client γ to request transaction τ) **:**

1: Send $\langle \tau \rangle_\gamma$ to the primary P.
2: Await a (v, ρ)-*proof-of-execution for* $\langle \tau \rangle_\gamma$,
consisting of identical messages $\texttt{Inform}(\langle \tau \rangle_\gamma, v, \rho, r)$ from **nf** distinct replicas.
3: Considers τ executed, with result r, as the ρ-th transaction.

Primary role (running at the primary P of view v) **:**

4: Let view v start after execution of the ρ-th transaction.
5: **while** P is the primary **do**
6: Await receipt of well-formed client requests $\langle \tau \rangle_\gamma$.
7: Broadcast $\texttt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$ to all replicas.
8: $\rho := \rho + 1$.
9: **end while**

Backup role (running at every replica $\mathrm{R} \in \mathfrak{R}$) **:**

10: **event** R receives message $m = \texttt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$ such that:
(1) v is the current view;
(2) m is signed by the primary of view v;
(3) R did not prepare a ρ-th proposal in view v; and
(4) $\langle \tau \rangle_\gamma$ is a well-formed client request
do
11: Prepare m as the proposal for round ρ in view v.
12: Broadcast $\texttt{Prepare}(m)$ to all replicas.
13: **end event**
14: **event** R receives **nf** messages $m_i = \texttt{Prepare}(m)$ such that:
(1) v is the current view;
(2) m is a well-formed proposal $\texttt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$;
(3) each message m_i is signed by a distinct replica; and
(4) R started the prepare phase for m
do
15: Wait until execution of all rounds preceding ρ.
16: Store *prepared certificate* $\mathbb{P}(m) = \{m_i \mid 1 \leq i \leq \mathbf{nf}\}$.
17: Execute τ as the ρ-th transaction, yielding result r.
18: Send $\texttt{Inform}(\langle \tau \rangle_\gamma, v, \rho, r)$ to γ.
19: **end event**

Fig. 3. The normal-case protocol in PoE.

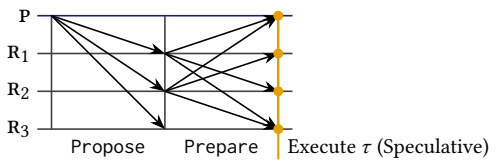

Fig. 4. A schematic representation of the normal-case protocol of PoE: the primary P proposes transaction τ to all replicas via a Propose message. Next, replicas prepare τ via a one-phase all-to-all message exchange. Notice that replicas do not explicitly *decide* on τ in the normal case, but do *execute* τ. In this example, replica R_3 is faulty and does not participate.

Theorem 4.1. *Round ρ of view v of the normal-case protocol of* PoE *satisfies the following two properties.*

(1) *If non-faulty replicas* R_1 *and* R_2*, prepared proposals* $m_1 = \texttt{Propose}(\langle\tau_1\rangle_{\gamma_1}, v, \rho)$ *and* $m_2 = \texttt{Propose}(\langle\tau_2\rangle_{\gamma_2}, v, \rho)$*, respectively, then* $m_1 = m_2$*.*
(2) *If a non-faulty primary* P *proposes* $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$*, communication is reliable, transaction execution is deterministic, and all non-faulty replicas executed the same sequence of* $\rho - 1$ *transactions, then* γ *will receive a* (v, ρ)*-proof-of-execution for* $\langle\tau\rangle_\gamma$*.*

Proof. We prove the two statements separately.

We prove the first statement by a standard contradiction-based quorum argument. Assume that non-faulty replicas R_1 and R_2 prepared proposals m_1 and m_2 with $m_1 \neq m_2$. As R_i, $i \in \{1, 2\}$, prepared proposal m_i, it must have received messages $\texttt{Prepare}(m_i)$ from **nf** distinct replicas (Line 14 of Figure 3). Let S_i be the set of **nf** replicas from which R_i received these messages and let $T_i = S_i \setminus \mathcal{F}$ be the non-faulty replicas in S_i. By construction, we have $|T_i| \geq \mathbf{nf} - \mathbf{f}$. As each non-faulty replica only sends `prepare` messages for a *single* proposal in round ρ of view v (Line 10 of Figure 3), $m_1 \neq m_2$ implies that $S_1 \cap S_2 = \emptyset$. Hence, we must have $|S_1 \cup S_2| = |S_1| + |S_2| \geq 2(\mathbf{nf} - \mathbf{f})$. As all replicas in $S_1 \cup S_2$ are non-faulty, we must also have $|S_1 \cup S_2| \leq \mathbf{nf}$. Hence, we must have $2(\mathbf{nf} - \mathbf{f}) \leq \mathbf{nf}$, which implies $\mathbf{nf} \leq 2\mathbf{f}$. As $\mathbf{n} = \mathbf{nf} + \mathbf{f}$, this implies $\mathbf{n} \leq 3\mathbf{f}$, a contradiction. Consequently, we conclude that $m_1 = m_2$ must hold.

Next, we prove the second statement. A non-faulty primary P will broadcast m to all non-faulty replicas (Line 7 of Figure 3). As communication is reliable, all **nf** non-faulty replicas will receive m as the first proposal of round ρ of view v (Line 10 of Figure 3) and broadcast a message $\texttt{Prepare}(m)$. As communication is reliable, all **nf** non-faulty replicas will receive $\texttt{Prepare}(m)$ from these **nf** non-faulty replicas (Line 14 of Figure 3) and execute τ (Line 17 of Figure 3). As all non-faulty replicas executed the same sequence of $\rho - 1$ transactions before executing τ, each non-faulty replica has the same state before executing τ. Consequently, due to deterministic execution of τ, all non-faulty replicas will obtain the same result r from execution of τ and send the same message $\texttt{Inform}(\langle\tau\rangle_\gamma, v, \rho, r)$ to γ (Line 18 of Figure 3). □

PoE only requires *two* phases of communication before execution commences, which is one less phase than Pbft. Consequently, the processing time of transactions within PoE is sharply reduced from at-least 3δ to 2δ, which will also reduce the latency clients perceive. Finally, the elimination of a phase of communication eliminates one round of messages, reducing the bandwidth cost for the normal-case of PoE (and, hence, allowing for an increase in throughput). As shown in Theorem 4.1(2), the elimination of a phase in PoE does *not* affect the service that is provided under normal conditions: clients still have a strong guarantee of service whenever the primary is non-faulty and communication is reliable.

The normal-case protocol of PoE provides only few guarantees, however. If non-faulty replica R speculatively executes some transaction τ proposed via $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$, then, due to Theorem 4.1(1), R *only* has the guarantee that no other transaction τ' is executed in round ρ of view v. In specific, R has no guarantees that any other non-faulty replicas prepared m or executed τ and has no guarantees that client γ received a (v, ρ)-proof-of-execution for $\langle\tau\rangle_\gamma$. As we shall see in Section 4.2 and Section 4.4, a subsequent failure of view v can easily lead to a situation in which the proposal m is *not preserved*, due to which R needs to rollback transaction τ. As we shall show in the remainder of Section 4, the design of PoE does provide the strong guarantee that m will always be guaranteed *if* a proof-of-execution for m *could* be received by client γ. To provide this strong guarantee (and to deal with certain kinds of failures), non-faulty replicas rely on the check-commit protocol of Section 4.3 and the view-change protocol of Section 4.4. Next, we look at the operations of PoE during failures: we look at how PoE deals with faulty (and possibly malicious) primaries and how PoE recovers from periods of unreliable communication.

4.2 Failure of the Normal-Case Protocol

The normal-case protocol described in Section 4.1 is designed to efficiently make consensus decisions and provide clients with proof-of-executions when operating under normal conditions. If the normal conditions are not met, then the normal-case protocol can fail in several ways, each following directly from the conditions stipulated in Theorem 4.1(2):

Example 4.2. Consider round ρ of view v in a deployment of PoE. The normal-case protocol of PoE can be disrupted in round ρ in the following ways:

(1) Any *non-primary faulty replica* can behave malicious by not participating or by sending invalid messages. Under normal conditions, this will not disrupt the normal-case protocol, however, as Theorem 4.1(2) does not depend on the behavior of any faulty replicas (that are not the primary).
(2) A *malicious primary* can choose to send different proposals for round ρ of view v to different non-faulty replicas or can choose to send no proposals to some non-faulty replicas. Due to Theorem 4.1(1), at-most one proposal will be prepared in round ρ of view v, this independent of the behavior of any faulty replicas. Hence, this malicious behavior can have only two outcomes:
   (a) A proposal $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$ will be prepared by some non-faulty replicas. The non-faulty replicas that did not receive m are *left in the dark*, as they will not be able to prepare in round ρ of view v and, consequently, are stuck. In this case, the client can still receive a *proof-of-execution*: if at-least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas prepared m, then the $\mathbf{f}$ faulty replicas can choose to also execute τ and send `Inform` messages to the client, this to assure the client receives $\mathbf{nf}$ identical `Inform` messages.
   (b) No proposal will be prepared by non-faulty replicas, in which case the primary *disrupts the progress* of the normal-case protocol and, consequently, prevents any transactions from being executed.
(3) Due to *unreliable communication*, messages can get lost (or arbitrarily delayed due to which receiving replicas consider them lost). Consequently, unreliable communication can prevent replicas from receiving proposals even from non-faulty primaries. Furthermore, unreliable communication can prevent replicas from receiving sufficient `Prepare` messages to finish their prepare phases.

Using digital signatures, it is rather straightforward to detect that a primary is sending conflicting proposals for a given round ρ of view v, as non-faulty replicas forward these conflicting proposals to each other during the prepare phase. Unfortunately, this does not extend to other malicious behavior, as we shall show next.

Example 4.3. Consider a system with $\mathfrak{R} = \{\text{P}, \text{R}_1, \text{R}_2, \text{R}_3\}$ such that P is the current primary of some view v. We consider the following three cases:

(1) The primary P is faulty and does *not* send any proposal to R_3. Hence, eventually R_3 detects a primary failure. To alert all other replicas of this failure, R_3 broadcasts message $\texttt{Failure}(v)$.
(2) The primary P is non-faulty and sends a proposal to R_3. Unfortunately, R_3 is faulty and pretends that the primary failed to propose. Consequently, R_3 broadcasts message $\texttt{Failure}(v)$.
(3) The primary P is non-faulty and sends a proposal to R_3. Unfortunately, communication is unreliable and this proposal is lost. This loss is interpreted by R_3 as a failure of the primary to propose. To alert all other replicas of this failure, R_3 broadcasts message $\texttt{Failure}(v)$.

We have sketched these three cases in Figure 5. As one can see, the replicas R_1 and R_2 receive the *exact same set of messages* in all three cases and, hence, observe identical behavior and cannot distinguish between the three cases.

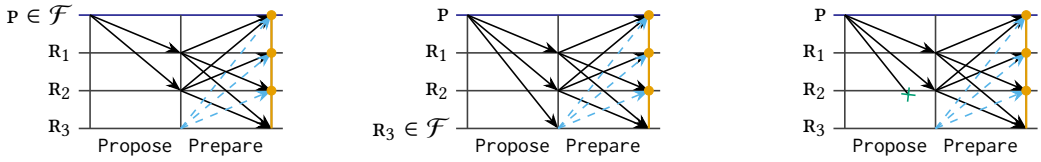

Fig. 5. A schematic representation of three failures in the normal-case protocol of PoE. *Left*, a faulty primary P does not propose to R_3. *Middle*, a faulty replica R_3 pretends that the primary P did not propose. *Right*, unreliable communication prevents the delivery of a proposal to R_3. In all three cases, R_3 alerts other replicas of failure via a `Failure` message (dashed arrow), while the replicas R_1 and R_2 observe identical behavior.

As Example 4.2 illustrates, any disruption of the normal-case protocol of PoE is caused by a faulty primary or by unreliable communication. If communication is unreliable, then there is no way to guarantee continuous service [18]. Hence, replicas assume failure of the current primary if the normal-case protocol is disrupted, while the design of PoE guarantees that unreliable communication does not affect the correctness of PoE and that the normal-case protocol of PoE will be able to recover when communication becomes reliable.

As Example 4.2(2) illustrated, a faulty primary can cause two kinds of disruptions. First, the primary can leave *replicas in the dark* without disrupting the progress of the normal-case protocol (Example 4.2(2a)) and we use the *check-commit protocol* of Section 4.3 to deal with such behavior. Second, the behavior of the primary can *disrupt the progress* of the normal-case protocol (Example 4.2(2b)) and we use the *view-change protocol* of Section 4.4 to deal with such behavior.

4.3 The Check-Commit Protocol

The main purpose of the check-commit protocol is to *commit* proposals $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$, after which non-faulty replicas that speculatively executed τ have the guarantee that τ will never rollback. Furthermore, we use the check-commit protocol to assure that non-faulty replicas cannot be left in the dark: the check-commit protocol assures that all non-faulty replicas receive prepared certificates for m if any replica can commit m.

For non-faulty replicas that successfully prepared via the normal-case protocol of Section 4.1, the check-commit protocol operates in a *single phase* of communication. Let $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$ be some proposal. The non-faulty replica R uses the check-commit protocol for m to indicate its willingness to commit to m and to determine whether a sufficient number of other replicas are willing to commit m. Replica R is willing to commit to m if the following conditions are met:

(1) R prepared m and speculatively executed τ and, hence, stored a prepared certificate for m and informed the client;
(2) R committed in all rounds before round ρ;[2] and
(3) R is currently still in view v.

If these conditions are met, then R starts the process to commit to m by broadcasting a message $\texttt{CheckCommit}(\mathbb{P}(m))$ with $\mathbb{P}(m)$ the prepared certificate for m stored by R. Next, each replica R waits until it receives well-formed `CheckCommit` messages for proposal m from at-least **nf** distinct replicas. After receiving these **nf** messages, the proposal m is *committed*. As a proof of this committed state for m, R stores a *commit certificate* $\mathbb{C}(m)$ consisting of these **nf** `CheckCommit` messages.

[2]To maximize throughput, one can choose for a design with out-of-order commit steps (in which round $\rho + 1$ can commit before round ρ). We have omitted such a design in this presentation of PoE, as it would complicate view-changes and substantially increase the complexity of the correctness proofs of PoE.

Check-commit role (running at every replica $\mathrm{R} \in \mathfrak{R}$) :

1: **event** R prepared $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$ and executed τ **do**
2: Wait until all previous rounds have a commit certificate.
3: **if** v is the current view **then**
4: Let $\mathbb{P}(m)$ be the prepared certificate stored for m.
5: Broadcast $\texttt{CheckCommit}(\mathbb{P}(m))$ to all replicas.
6: **end if**
7: **end event**
8: **event** R receives a well-formed message $\texttt{CheckCommit}(\mathbb{P}(m))$ **do**
9: Let $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v', \rho')$ and $\mathbb{P}(m) = \{m_1, \ldots, m_{\mathbf{nf}}\}$.
10: **if** $v = v'$ and R did not prepare m **then**
11: Prepare m and execute τ using Lines 16–18 of Figure 3 with Prepare messages $m_1, \ldots, m_{\mathbf{nf}}$.
12: **end if**
13: **end event**
14: **event** R receives **nf** messages $m_i = \texttt{CheckCommit}(\mathbb{P}(m)_i)$ such that:
(1) m is a well-formed proposal $\texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$;
(2) v is the current view;
(3) $\mathbb{P}(m)_i$ is a well-formed prepared certificate for m;
(4) each message is signed by a distinct replica; and
(5) R prepared m
do
15: Wait until all previous rounds have a commit certificate.
16: Store *commit certificate* $\mathbb{C}(m) = \{\mathbb{P}(m)_i \mid 1 \leq i \leq \mathbf{nf}\}$.
17: **end event**

Fig. 6. The check-commit protocol in PoE.

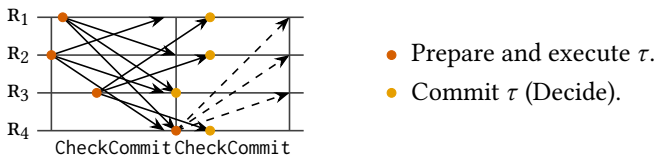

Fig. 7. A schematic representation of the check-commit protocol of PoE. In this illustration, replicas R_1, R_2, and R_3 initiate the check-commit protocol due to preparing and executing a transaction τ (which they finish at different times), whereas R_4 learns τ via the CheckCommit message it receives from replicas R_1 and R_2. The replicas explicitly *decide* on τ upon finishing the protocol.

Let $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$ be a proposal. If any replica receives a well-formed message $\texttt{CheckCommit}(\mathbb{P}(m))$, then it can use the provided prepared certificate $\mathbb{P}(m)$ for m to prepare m and execute τ (if it has not yet done so).[3]

The pseudo-code for this check-commit protocol can be found in Figure 6 and an illustration of the working of this protocol can be found in Figure 7.

The correctness of PoE is based on the following properties of the check-commit protocol:

[3]If digital signatures are not used on Prepare messages, then prepared certificates cannot be reliably forwarded. In that case, a replica needs to receive $\mathbf{nf} - \mathbf{f} > \mathbf{f}$ identical CheckCommit messages for proposal m from distinct replicas before it can use the provided information to prepare m. We refer to Section 5.2 and Section 5.5 for further details. We note that if communication is reliable, then a replica is guaranteed to receive $\mathbf{nf} - \mathbf{f}$ identical CheckCommit messages *unless* the behavior of the primary disrupts the progress of the normal-case protocol, in which case a view-change will happen (see Section 4.4).

Theorem 4.4. *Assume that communication is reliable, transaction execution is deterministic, and all non-faulty replicas are in view v. Round ρ of view v of the check-commit protocol of* PoE *satisfies the following three properties:*

(1) *if a client γ receives a (v, ρ)-proof-of-execution for $\langle\tau\rangle_\gamma$, then all non-faulty replicas will prepare some proposal proposing $\langle\tau\rangle_\gamma$ in round ρ;*
(2) *if all non-faulty replicas executed the same sequence of $\rho - 1$ transactions and a non-faulty replica stored a commit certificate for proposal $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$, then all non-faulty replicas will store a commit certificate for m and client γ will receive a (v, ρ)-proof-of-execution for $\langle\tau\rangle_\gamma$; and*
(3) *if there exists a commit certificate $\mathbb{C}(m)$ for proposal $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$, then at-least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas stored prepared certificates for proposal m, executed m, and stored commit certificates for proposals in every round ρ' preceding round ρ ($\rho' < \rho$).*

Proof. We prove the three statements separately.

Client γ only receives a (v, ρ)-proof-of-execution for $\langle\tau\rangle_\gamma$ if at-least $\mathbf{nf}$ distinct replicas signed some message $\texttt{Inform}(\langle\tau\rangle_\gamma, v, \rho, r)$. As there are at-most $\mathbf{f}$ faulty replicas and $\mathbf{nf} > 2\mathbf{f}$, there exist at-least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas that must have sent these Inform messages to γ. Choose such a non-faulty replica Q. Replica Q will only do so after preparing some proposal $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$ and executing τ (Line 18 of Figure 3). Consequently, Q satisfies the conditions to start the commit-phase for m and will broadcast a well-formed message $\texttt{CheckCommit}(\mathbb{P}(m)_Q)$ to all replicas (Line 5 of Figure 6). Hence, as communication is reliable, all non-faulty replicas will receive a well-formed message $\texttt{CheckCommit}(\mathbb{P}(m)_Q)$ and will be able to prepare m (Line 11 of Figure 6).

Next, we prove the second statement. Assume that non-faulty replica R stored a commit certificate $\mathbb{C}(m)$. This certificate is based on well-formed CheckCommit messages from $\mathbf{nf}$ distinct replicas (Line 14 of Figure 6). Consequently, there exists a non-faulty replica Q that must have sent one of these CheckCommit messages to R. This replica Q must have broadcasted its well-formed message $\texttt{CheckCommit}(\mathbb{P}(m)_Q)$ to all replicas (Line 5 of Figure 6). Hence, as communication is reliable, all non-faulty replicas will receive a well-formed message $\texttt{CheckCommit}(\mathbb{P}(m)_Q)$, will be able to prepare m (Line 11 of Figure 6), will receive well-formed messages $\texttt{CheckCommit}(\mathbb{P}(m)_i)$ from all $\mathbf{nf}$ non-faulty replicas (Line 5 of Figure 6), and will be able to commit m (Line 14 of Figure 6). Finally, we can use the same argument as in the proof of Theorem 4.1(2) to prove that γ will receive a (v, ρ)-proof-of-execution for $\langle\tau\rangle_\gamma$.

Finally, we prove the third statement. The commit certificate $\mathbb{C}(m)$ can only exist if $\mathbf{nf}$ distinct replicas signed the message $\texttt{CheckCommit}(m)$. Hence, there are at-least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas that signed message $\texttt{CheckCommit}(m)$. These $\mathbf{nf} - \mathbf{f}$ non-faulty replicas will only sign $\texttt{CheckCommit}(m)$ after they prepared and executed m and committed all previous rounds (Line 2 of Figure 6), completing the proof. □

4.4 The View-Change Protocol

To deal with disruptions of the normal-case protocol of PoE, PoE employs a view-change protocol. This protocol has two goals:

(G1) The view-change protocol must always *preserve requests $\langle\tau\rangle_\gamma$ with a proof-of-execution*: if client γ could have observed successful execution of transaction τ, then the view-change protocol must guarantee that the execution of τ is preserved by the system and included in *all* future views. Furthermore, the view-change protocol must always *preserve committed transactions*.
(G2) The view-change protocol must *resume the normal-case protocol* when communication is reliable: if communication is reliable and the normal-case protocol is disrupted to the point

where the check-commit protocol can no longer assure that all non-faulty replicas commit, then the view-change protocol eventually puts all non-faulty replicas in the same future view v' in which the normal-case protocol can operate without disruptions.

These goals reflect the guarantees provided by weak consensus: Goal G1 will be used to provide *non-divergence*, whereas Goal G2 will be used to provide *termination* (when communication is sufficiently reliable).

The view-change protocol for view v operates in three stages:

(1) Replicas enter the *failure detection stage* when they detect failure of view v. In this first stage, each replica that detects failure of the current primary will *alert* all other replicas of this failure by broadcasting a failure alert message `Failure`(v). These messages `Failure`(v) are not only used to assure that all replicas detect failure of the primary, but will also assure sufficient *synchronization* of the replicas to guarantee the view-change protocol will succeed whenever communication is reliable.
(2) When replicas receive messages `Failure`(v) for the same view v from $\mathbf{nf}$ distinct replicas, they enter the *new-view proposal stage*. During this second stage, replicas provide the *new primary*, the replica P' with $\text{id}(\text{P}') = (v + 1) \bmod \mathbf{n}$, with a `ViewState` message that summarizes their internal state. After the new primary P' receives internal state summaries from sufficient replicas, P' can construct a *new-view proposal* that determines the state in which the next view starts. The new primary broadcasts the new-view proposal by broadcasting a `NewView` message to all other replicas. All replicas wait for this new-view proposal from P' (and detect failure of view $v + 1$ if no valid new-view proposal arrives on time).
(3) When replicas receive a valid `NewView` message representing a new-view proposal from the new primary, they enter the *new-view accept stage*. In this third and final stage, the replicas will validate the new-view proposal they received, update their local state based on the information in the new-view proposal, and start the normal-case protocol of PoE for this new view.

The pseudo-code for the view-change protocol can be found in Figure 8, and an illustration of the working of this protocol can be found in Figure 9. Next, we will detail each of the three stages in more detail.

The Failure Detection Stage. In the failure detection stage, replicas detect failure of view v and alert other replicas of this failure. To alert other replicas of a failure of view v, replicas will broadcast messages `Failure`(v). Before a replica R enters the failure detection stage, replica R needs to detect failure of the primary. Replica R can do so in two ways.

First, R can set a timer whenever it *expects a proposal* for some round ρ from the current primary. If this timer expires and no proposal for round ρ was finished (proposed, executed, and committed), then R detects failure. Replica R can expect a proposal whenever it forwarded a valid client request to the current primary or whenever it receives valid `Prepare` messages for round ρ from non-faulty replicas (e.g., by receiving such `Prepare` messages from $\mathbf{f} + 1$ distinct replicas) without receiving any corresponding `Propose` messages.

Second, R can receive failure alerts for the current (or future) view of at-least $\mathbf{f} + 1$ distinct other replicas. As there are at-most $\mathbf{f}$ faulty replicas, at-least one of these alerts must have originated from some non-faulty replica Q. In this case, R can simply use this observation to detect failure.

The New-View Proposal Stage. Replicas enter the new-view proposal stage for view v after they receive messages `Failure`(v'), $v' \geq v$, from $\mathbf{nf}$ distinct replicas. This condition will *synchronize* the view-change in all non-faulty replicas whenever communication is reliable:

Failure detection stage (running at every replica $\text{R} \in \mathfrak{R}$) :

event R detects failure of view v **do**
  if R did not previously detect failure of view v **then**
    Broadcast `Failure`(v) to all replicas and periodically rebroadcast until the new-view proposal stage is entered.
  end if
end event
event R receives messages `Failure`(v') with $v' \geq v$ and signed by $\mathbf{f}+1$ distinct replicas **do**
  R detects failure of view v.
end event

New-view proposal stage (running at every replica $\text{R} \in \mathfrak{R}$) :

event R receives messages `Failure`(v') with $v' \geq v$ and signed by **nf** distinct replicas **do**
  Halt the normal-case protocol of Section 4.1 for view v.
  Halt the check-commit protocol of Section 4.3 for view v.
  /* Let $\mathbb{C}(m)$ be the last commit certificate stored by R and let E be the set of prepared certificates $\mathbb{P}(m')$ for all proposals m' that R executed (without rollback) after proposal m. */
  Send `ViewState`$(v, \mathbb{C}(m), \text{E})$ to replica P′, $\text{id}(\text{P}') = (v+1) \bmod \mathbf{n}$.
  Await a valid `NewView` message for view $v+1$. If no such message arrives on time, then detect failure of view $v+1$.
end event
event R receives well-formed messages `ViewState`$(v, \mathbb{C}(m_i)_i, \text{E}_i)$, $1 \leq i \leq \mathbf{nf}$,
    signed by **nf** distinct replicas **do**
  if $\text{id}(\text{R}) = (v+1) \bmod \mathbf{n}$ (R is the primary of view $v+1$) **then**
    Let V = $\{$`ViewState`$(v, \mathbb{C}(m_i)_i, \text{E}_i) \mid 1 \leq i \leq \mathbf{nf}\}$.
    Broadcast `NewView`$(v+1, \text{V})$ to all replicas.
  end if
end event

New-view accept stage (running at every replica $\text{R} \in \mathfrak{R}$) :

event R receives well-formed message `NewView`$(v+1, \text{V})$ from replica P′, $\text{id}(\text{P}') = (v+1) \bmod \mathbf{n}$ **do**
  Update the internal state in accordance to V and start the normal-case protocol of Section 4.1 for view $v+1$.
end event

Fig. 8. The view-change protocol of PoE.

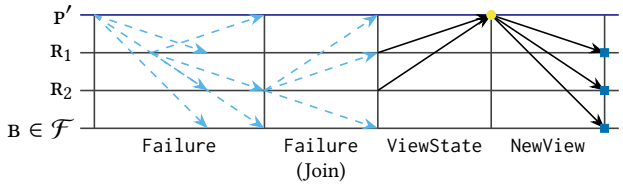

Fig. 9. A schematic representation of the view-change protocol of PoE. The current primary B is faulty and needs to be replaced. The next primary, P′, and the replica R_1 detected this failure first and alerted all replicas via `Failure` messages. The replica R_2 joins in on this failure. After replicas receive $\mathbf{nf} = 3$ `Failure` messages, they send their state to P′ via `ViewState` messages. Finally, P′ uses **nf** such `ViewState` messages to propose a new view via a `NewView` message.

Lemma 4.5. *Assume communication is reliable and has message delay δ. If the first non-faulty replica to enter the new-view proposal stage does so at time t, then all non-faulty replicas will enter the new-view proposal stage at-or-before $t + 2\delta$.*

Proof. Let R be the first non-faulty replica that enters the new-view proposal stage. As replica R entered the new-view proposal stage at t, it received messages $\texttt{Failure}(v')$, $v' \geq v$, from $\mathbf{nf}$ distinct replicas at-or-before t (Line 9 of Figure 8). As there are at-most $\mathbf{f}$ faulty replicas and $\mathbf{nf} > 2\mathbf{f}$, at-least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ of these messages originated from non-faulty replicas, whom always broadcast their Failure messages (Line 3 of Figure 8). Hence, if communication is reliable, then all replicas will receive at-least $\mathbf{f} + 1$ Failure messages within at-most a message-delay δ. Consequently, all non-faulty replicas will have detected failure of view v at-most at $t + \delta$ and broadcast Failure messages themselves (Line 7 of Figure 8), due to which all replicas will receive $\mathbf{nf}$ Failure messages at-most at $t + 2\delta$ and enter the new-view proposal stage for view v (Line 9 of Figure 8). □

Consider a period of reliable communication in an asynchronous environment. In this environment, the message delay δ as used in Lemma 4.5 is bounded by a value unknown to the non-faulty replicas. In Lemma 4.13 and Theorem 4.15, we show how non-faulty replicas can determine a sufficiently high upper bound for δ.

When a non-faulty replica R enters the new-view proposal stage for view v, R first halts its participation in the normal-case protocol of Section 4.1 and the check-commit protocol of Section 4.3 for view v. Next, R constructs a summary of its internal state. To do so, R constructs the set E that holds the prepared certificates $\mathbb{P}(m')$ of proposals executed (without rollback) by R and stored *since* the last commit certificate $\mathbb{C}(m)$ stored by R. To simplify presentation, we assume that each replica R has a *dummy commit certificate* for round $\rho = 0$ (that does not propose any request), which R uses when it has not yet committed proposals. The pair $(\mathbb{C}(m), \mathsf{E})$ will serve as the summary of the current state of R. Finally, R sends $(\mathbb{C}(m), \mathsf{E})$ to the next primary, the replica P′ with $\mathrm{id}(\mathrm{P}') = (v + 1) \bmod \mathbf{n}$, via a $\texttt{ViewState}(v, \mathbb{C}(m), \mathsf{E})$ message.[4]

The next primary, the replica P′ with $\mathrm{id}(\mathrm{P}') = (v+1) \bmod \mathbf{n}$, will wait until it receives well-formed messages

$$\mathsf{V} = \{\texttt{ViewState}(v, \mathbb{C}(m_i)_i, \mathsf{E}_i) \mid 1 \leq i \leq \mathbf{nf}\},$$

signed by $\mathbf{nf}$ distinct replicas. After P′ has received these messages V, P′ broadcasts the message $\texttt{NewView}(v+1, \mathsf{V})$ to all replicas. This message announces a new-view whose initial state is based on the information in V. To assure timely arrival of these messages, we use the following assumption:

Assumption 4.6. *All* PoE *messages have a predetermined bounded size. In specific, there is a known upper bound on the size of proposals of the form* $\texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$ *and there is an upper bound on the number of rounds with prepared proposals that are not yet committed. We refer to this upper bound as the* window size.

Using Assumption 4.6, we can assume, without loss of generality, that message delivery times are independent of the message size and are fully determined by some unknown message delay.

Lemma 4.7. *Assume communication is reliable and has message delay δ. If the first non-faulty replica to enter the new-view proposal stage does so at time t, then any non-faulty next primary will be able to deliver a new-view proposal at-or-before $t + 4\delta$.*

Proof. Due to Lemma 4.5, all non-faulty replicas will have entered the new-view proposal stage at $t + 2\delta$. Hence, the next primary will have received sufficient ViewState messages at $t + 3\delta$ to propose a new-view (Line 16 of Figure 8), and a well-formed new-view proposal will be broadcast at-or-before $t + 3\delta$ to all replicas (Line 19 of Figure 8). Consequently, all replicas will receive this new-view proposal at-or-before $t + 4\delta$. □

[4]In PoE, we distinguish between Failure messages, which are *small* and broadcasted to all other replicas, that are used to detect failures; and ViewState messages, which are *large* and send only to the next primary, that are used to construct a NewView message. If bandwidth is not a limiting factor, then one can simply broadcast ViewState messages to all replicas to fulfill both roles.

Due to synchronized entry of the new-view proposal stage, every non-faulty replica R will expect a timely new-view proposal. If no such proposal is received, then R will detect failure of view $v + 1$.

The New-View Accept Stage. Replicas enter the new-view accept stage after they receive a well-formed $\texttt{NewView}(v + 1, \mathsf{V})$ message from the primary of view $v + 1$, the replica P′ with $\mathrm{id}(\mathrm{P}') = (v + 1) \bmod \mathbf{n}$. Based on the information included in V, each replica will determine the state in which $v + 1$ starts. This state consists of the set of transactions that have been proposed before view $v + 1$ and, hence, determines at which round the primary P′ of $v + 1$ can start proposing. We make the following key assumption:

Assumption 4.8. *If there exists a commit certificate $\mathbb{C}(m)$ for proposal $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$, then every commit certificate $\mathbb{C}(m')$ for round ρ is a commit certificate for a proposal proposing $\langle\tau\rangle_\gamma$.*

As part of proving the correctness of the view-change protocol, we will prove that Assumption 4.8 holds. Before we do so, we first show how we interpret the unique state represented by a well-formed proposal $n = \texttt{NewView}(v + 1, \mathsf{V})$ using Assumption 4.8. Furthermore, we will show that this derived state satisfies Goal G1. Let $\mathsf{V} = \{\texttt{ViewState}(v, \mathbb{C}(m_i)_i, \mathsf{E}_i) \mid 1 \leq i \leq \mathbf{nf}\}$ be the set of ViewState messages included in n. Let

$$\mathsf{C}(\rho) = \{\langle\tau\rangle_\gamma \mid \exists v\, \exists \mathbb{C}(m)\ (m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho))\},$$

be the set of client requests $\langle\tau\rangle_\gamma$ that have been committed in round ρ (a view v exists for which a commit certificate $\mathbb{C}(m)$ exists with $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$). Let

$$\mathsf{P}(n, \rho) = \{m \mid (\mathbb{P}(m) \in \mathsf{E}_i) \wedge (m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho))\}$$

be the set of all proposals included in V for round ρ, let

$$\mathsf{M}(n, \rho) = \{\langle\tau\rangle_\gamma \mid (\texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho) \in \mathsf{P}(n, \rho)) \wedge v = \max\{v' \mid \texttt{Propose}(\langle\tau'\rangle_{\gamma'}, v', \rho) \in \mathsf{P}(n, \rho)\}\}$$

be the set of client requests proposed in the most-recent view of any proposals in $\mathsf{P}(n, \rho)$, let

$$\mathsf{LC}(n) = \max\{\rho_i \mid m_i = \texttt{Propose}(\langle\tau_i\rangle_{\gamma_i}, v_i, \rho_i)\}$$

be the latest round for which a commit certificate is included in V, and let

$$\mathsf{LP}(n) = \max(\{\rho \mid \mathsf{P}(n, \rho) \neq \emptyset\} \cup \mathsf{LC}(n))$$

be the latest round for which n includes proposals. We will use the sets $\mathsf{C}(\cdot)$ and $\mathsf{M}(n, \cdot)$ to define the unique sequence of client transactions preserved by new-view proposal n. To do so, we use the following technical result:

Lemma 4.9. *Let $n = \texttt{NewView}(v + 1, \mathsf{V})$ be a well-formed new-view proposal. We have:*

(1) *if Assumption 4.8 holds, then $\mathsf{C}(\rho)$ is a singleton set ($|\mathsf{C}(\rho)| = 1$) for all $\rho \leq \mathsf{LC}(n)$; and*
(2) *$\mathsf{M}(n, \rho)$ is a singleton set ($|\mathsf{M}(n, \rho)| = 1$) for all $\mathsf{LC}(n) < \rho \leq \mathsf{LP}(n)$.*

Proof. First, we prove $|\mathsf{C}(\rho)| = 1$ for all $\rho \leq \mathsf{LC}(n)$. By the definition of $\mathsf{LC}(n)$, there exists a commit certificate for round $\mathsf{LC}(n)$. By Theorem 4.4(3), there exist commit certificates for all rounds $\rho \leq \mathsf{LC}(n)$. Hence, for all $\rho \leq \mathsf{LC}(n)$, $\mathsf{C}(\rho) \neq \emptyset$. Finally, by Assumption 4.8, we conclude $|\mathsf{C}(\rho)| = 1$.

Next, we prove $|\mathsf{M}(n, \rho)| = 1$ for all $\mathsf{LC}(n) < \rho \leq \mathsf{LP}(n)$. By the definition of $\mathsf{LP}(n)$, we must have $\mathsf{P}(n, \mathsf{LP}(n)) \neq \emptyset$ and $\mathsf{M}(n, \mathsf{LP}(n)) \neq \emptyset$. Let $\langle\tau\rangle_\gamma \in \mathsf{M}(n, \mathsf{LP}(n))$. By the definition of $\mathsf{M}(n, \mathsf{LP}(n))$, there exists a message $m_{\mathsf{vs}} = \texttt{ViewState}(v, \mathbb{C}(m), \mathsf{E}) \in \mathsf{V}$ such that there exists a

prepared certificate $\mathbb{P}(m') \in \mathsf{E}$ with $m' = \mathtt{Propose}(\langle\tau\rangle_\gamma, v', \mathsf{LP}(n))$. Let R be the replica that signed this message m_{vs} and let ρ' be the round for which m was proposed. By the definition of $\mathsf{LC}(n)$, we have $\rho' \leq \mathsf{LC}(n)$. As $\mathtt{NewView}(v+1, \mathsf{V})$ is well-formed, also the message m_{vs} is well-formed. Hence, $v' \leq v$. As non-faulty replicas only execute proposals for round ρ after they executed proposals for all preceding rounds (Line 17 of Figure 3) and E is well-formed, it must contain proposals for all rounds ρ, $\rho' \leq \mathsf{LC}(n) < \rho \leq \mathsf{LP}(n)$. Hence, for all $\mathsf{LC}(n) < \rho \leq \mathsf{LP}(n)$, $\mathsf{P}(n, \rho) \neq \emptyset$ which implies $\mathsf{M}(n, \rho) \neq \emptyset$. Due to Theorem 4.1(1), there can be only one prepared certificate for round ρ of view v'. Hence, we have $|\mathsf{M}(n, \rho)| \leq 1$ and we conclude $|\mathsf{M}(n, \rho)| = 1$. □

Due to Lemma 4.9, the sequence of client requests

$$\mathcal{L}(n) = \langle\tau_1\rangle_{\gamma_1}, \ldots, \langle\tau_{\mathsf{LP}(n)}\rangle_{\gamma_{\mathsf{LP}(n)}} \text{ with } \langle\tau_\rho\rangle_{\gamma_\rho} \in \begin{cases} \mathsf{C}(\rho) & \text{if } 1 \leq \rho \leq \mathsf{LC}(n); \\ \mathsf{M}(n, \rho) & \text{if } \mathsf{LC}(n) < \rho \leq \mathsf{LP}(n), \end{cases}$$

is uniquely defined by the new-view proposal n and specifies the state in which view $v+1$ starts. Replica R will update its internal state (Line 23 of Figure 8) in accordance to $\mathcal{L}(n)$ in the following way:

(1) R will *rollback* every client request it executed and that is not included in $\mathcal{L}(n)$;
(2) R will obtain a commit certificate for each round ρ, $1 \leq \rho \leq \mathsf{LC}(n)$, for which it does not yet have a commit certificate, execute the newly obtained requests in order, and inform the client of the result; and
(3) R will expect the new primary to *repropose* client requests $\langle\tau_\rho\rangle_{\gamma_\rho}$, $\mathsf{LC}(n) < \rho \leq \mathsf{LP}(n)$, in round ρ of view $v+1$. If the new primary fails to do so, then failure of view $v+1$ is detected.

The last step assures that all non-faulty replicas that receive a new-view proposal for view $v+1$ will only proceed in this new view if they all received *compatible* new-view proposals that each represent the same *unique* ledger.

After updating its internal state, replica R will start the normal-case protocol for view $v+1$ by accepting any proposal from P′ for rounds after $\mathsf{LP}(n)$.

Next, we prove that the view-change protocol outlined above satisfies Goal G1 and Goal G2.

The View-Change Protocol Satisfies Goal G1. A client request $\langle\tau\rangle_\gamma$ needs to be preserved by the view-change protocol as the ρ-th request if it either has a (v, ρ)-proof-of-execution or a replica stored a commit certificate $\mathbb{C}(m)$ for some proposal $m = \mathtt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$. If $\langle\tau\rangle_\gamma$ has a (v, ρ)-proof-of-execution, then there must be a set of identical messages $\mathtt{Inform}(\langle\tau\rangle_\gamma, v, \rho, r)$ signed by **nf** distinct replicas. Likewise, if $\langle\tau\rangle_\gamma$ has a commit certificate $\mathbb{C}(m)$, then there must be a set of identical messages $\mathtt{CheckCommit}(m)$ signed by **nf** distinct replicas. As there are at-most **f** faulty replicas, at-least $\mathbf{nf} - \mathbf{f}$ of these messages must originate from non-faulty replicas, which will only produce these messages after they prepared and executed proposal m. Hence, a *necessary condition* for the preservation of $\langle\tau\rangle_\gamma$ is the existence of a proposal m that is executed by at-least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas. Next, we prove that the view-change protocol preserves such requests:

Theorem 4.10. *Let v be the first view in which a proposal $m = \mathtt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$ for round ρ was executed by $\mathbf{nf} - \mathbf{f}$ non-faulty replicas, let $m' = \mathtt{Propose}(\langle\tau'\rangle_{\gamma'}, v', \rho)$ be any proposal for round ρ with $v \leq v'$, and let $n = \mathtt{NewView}(w+1, \mathsf{V})$, $v \leq v' \leq w$, be a well-formed new-view proposal. The following properties hold:*

(1) *there exist no commit certificates $\mathbb{C}(\mathtt{Propose}(\langle\tau''\rangle_{\gamma''}, v'', \rho))$ with $v'' < v$;*
(2) *in views v', $v < v'$, non-faulty replicas only sign $\mathtt{Prepare}(m')$ if $\langle\tau'\rangle_{\gamma'} = \langle\tau\rangle_\gamma$;*
(3) *if there exists a prepared certificate $\mathbb{P}(m')$, then $\langle\tau'\rangle_{\gamma'} = \langle\tau\rangle_\gamma$;*
(4) *if $\mathbf{nf} - \mathbf{f}$ non-faulty replicas executed m', then $\rho \leq \mathsf{LP}(n)$;*

(5) *if* $\mathbf{nf} - \mathbf{f}$ *non-faulty replicas committed* m'*, then* $\rho \leq \mathsf{LC}(n)$*; and*
(6) $\mathcal{L}(n)[\rho] = \langle \tau \rangle_\gamma$*;*

Proof. First, we prove **(1)** by contradiction. Assume that a commit certificate $\mathbb{C}(\mathtt{Propose}(\langle \tau'' \rangle_{\gamma''}, v'', \rho))$ with $v'' < v$ exists. Hence, there exist messages $\mathtt{CheckCommit}(\mathbb{P}(m'')_i)$, $1 \leq i \leq \mathbf{nf}$, signed by $\mathbf{nf}$ distinct replicas (Line 16 of Figure 6). At-least $\mathbf{nf} - \mathbf{f}$ of these messages are signed by non-faulty replicas. As non-faulty replicas only construct and sign messages $\mathtt{CheckCommit}(\mathbb{P}(m'')_i)$ if they prepared and executed m'' (Line 5 of Figure 6) and v was the first view in which $\mathbf{nf} - \mathbf{f}$ non-faulty replicas executed a proposal for round ρ, we must have $v \leq v''$, a contradiction, and we conclude that $\mathbb{C}(m'')$ does not exist.

We prove **(2)**–**(6)** by induction on the current view w. As the base case, we prove that each of the statements **(2)**–**(6)** hold in view $w = v$.

(2). We have $w = v' = v$. Hence, the statement voidly holds.

(3). As $w = v$, we only need to consider $v = v'$. If $v = v'$, then, by Theorem 4.1(1), we have $m' = m$ and $\langle \tau' \rangle_{\gamma'} = \langle \tau \rangle_\gamma$.

(4) and **(5)**. As $w = v$, we can use **(1)** and the proof of **(3)**, to derive that $m' = m$. Let $n = \mathtt{NewView}(w+1, \mathsf{V})$, $w = v$, be a well-formed new-view proposal. Let C be the set of $\mathbf{nf} - \mathbf{f}$ non-faulty replicas that executed m (proof for **(4)**) or committed m (proof for **(5)**), let S be the set of $\mathbf{nf}$ distinct replicas that signed the $\mathtt{ViewState}$ messages included in V, and let $T = S \setminus \mathcal{F}$ be the non-faulty replicas in S. By construction, we have $|T| \geq \mathbf{nf} - \mathbf{f}$. Hence, using the same contradiction argument as used in the proof of Theorem 4.1(1), we can conclude that $(C \cap T) \neq \emptyset$. Let $\mathrm{R} \in (C \cap T)$ be such a non-faulty replica that executed m (proof for **(4)**) or committed m (proof for **(5)**) and that signed a message $\mathtt{ViewState}(v, \mathbb{C}(m_\mathrm{R}), \mathsf{E}) \in \mathsf{V}$ with $m_\mathrm{R} = \mathtt{Propose}(\langle \tau_\mathrm{R} \rangle_{\gamma_\mathrm{R}}, v_\mathrm{R}, \rho_\mathrm{R})$.

If R committed m_R, then we must have $\rho \leq \rho_\mathrm{R}$ as non-faulty replicas commit proposals in order (Line 15 of Figure 6) and m_R is the last proposal R committed (Line 12 of Figure 8). Hence, by the definition of $\mathsf{LC}(n)$ and $\mathsf{LP}(n)$, we have $\rho \leq \rho_\mathrm{R} \leq \mathsf{LC}(n) \leq \mathsf{LP}(n)$.

If R executed m_R without committing m_R, then $\rho > \rho_\mathrm{R}$. As R executed m_R, R must have stored a prepared certificate $\mathbb{P}(m_\mathrm{R})$ (Line 16 of Figure 3) and we conclude $\mathbb{P}(m_\mathrm{R}) \in \mathsf{E}$ (Line 12 of Figure 8). By the definition of $\mathsf{P}(n, \rho)$ and the definition of $\mathsf{LP}(n)$, we conclude $m_\mathrm{R} \in \mathsf{P}(n, \rho)$ and $\rho \leq \mathsf{LP}(n)$.

(6). As $\mathbf{nf} - \mathbf{f}$ non-faulty replicas executed m in view $w = v$, we have $\rho \leq \mathsf{LP}(n)$ by **(4)**. Hence, $\mathcal{L}(n)[\rho]$ is defined.

If $\rho \leq \mathsf{LC}(n)$, then, by the definition of $\mathcal{L}(n)$ and $\mathsf{C}(\rho)$, we have $\mathcal{L}(n)[\rho] = \langle \tau_n \rangle_{\gamma_n}$ with $\langle \tau_n \rangle_{\gamma_n} \in \mathsf{C}(\rho)$ and there exists a commit certificate $\mathbb{C}(m_n)$ for some proposal $m_n = \mathtt{Propose}(\langle \tau_n \rangle_{\gamma_n}, v_n, \rho)$. By **(1)**, we conclude that $v \leq v_n$ and, hence, $v_n = v$. Due to Theorem 4.4(3), there exists a prepared certificate $\mathbb{P}(m_n)$ and, by **(3)**, we conclude $\langle \tau_n \rangle_{\gamma_n} = \langle \tau \rangle_\gamma$.

If $\mathsf{LC}(n) < \rho \leq \mathsf{LP}(n)$, then, by the definition of $\mathcal{L}(n)$, $\mathsf{M}(n, \rho)$, and $\mathsf{P}(n, \rho)$, we have $\mathcal{L}(n)[\rho] = \langle \tau_n \rangle_{\gamma_n}$ with $\langle \tau_n \rangle_{\gamma_n} \in \mathsf{M}(n, \rho)$ and there exists a prepared certificate $\mathbb{P}(m_n)$ for some proposal $m_n = \mathtt{Propose}(\langle \tau_n \rangle_{\gamma_n}, v_n, \rho)$ with $m_n \in \mathsf{P}(n, \rho)$. Using the same reasoning as in the proof of **(4)**, there exists a non-faulty replica R that executed m and signed a message $\mathtt{ViewState}(v, \mathbb{C}(m_\mathrm{R}), \mathsf{E}) \in \mathsf{V}$. As $\mathsf{LC}(n) < \rho$, R did not commit m, $\mathbb{P}(m) \in \mathsf{E}$, and $m \in \mathsf{P}(n, \rho)$. By the definition of $\mathsf{M}(n, \rho)$, we conclude $v \leq v_n$. Hence, $v_n = v$ and, by **(3)**, we conclude $\langle \tau_n \rangle_{\gamma_n} = \langle \tau \rangle_\gamma$.

As the induction hypothesis, we assume that **(2)**–**(6)** hold in every view j, $v \leq j < w$. Now consider view w and let $n_w = \mathtt{NewView}(w, \mathsf{V}_w)$ be a well-formed new-view proposal that can be used to enter view w (Line 22 of Figure 8). Next, we prove the inductive step for each of the statements **(2)**–**(6)**.

(2). Let $\mathtt{Prepare}(m')$, $v < v' \leq w$, be a message signed by a non-faulty replica. We apply induction hypothesis **(4)** on n_w and conclude $\rho \leq \mathsf{LP}(n_w)$. If, furthermore, $\rho \leq \mathsf{LC}(n_w)$, then the primary of view w cannot propose for round ρ and we must have $v \leq v' < w$. We apply induction

hypothesis **(2)** on $\texttt{Prepare}(m')$ to conclude $\langle\tau'\rangle_{\gamma'} = \langle\tau\rangle_{\gamma}$. Otherwise, if $\mathsf{LC}(n_w) < \rho \leq \mathsf{LP}(n_w)$, then the primary of view w can only propose $\langle\tau\rangle_{\gamma}$ for round ρ (Line 23 of Figure 8), and we conclude $\langle\tau'\rangle_{\gamma'} = \langle\tau\rangle_{\gamma}$.

(3). Let $\mathbb{P}(m')$ be a prepared certificate, $v \leq v' \leq w$. If $v' < w$, then we apply induction hypothesis **(3)** to conclude that $\langle\tau'\rangle_{\gamma'} = \langle\tau\rangle_{\gamma}$. Otherwise, if $v' = v$, then there exist messages $\texttt{Prepare}(m)$ signed by **nf** distinct replicas (Line 16 of Figure 3). At-least $\mathbf{nf} - \mathbf{f}$ of these messages are signed by non-faulty replicas. By **(2)**, these non-faulty replicas will only sign m' if $\langle\tau'\rangle_{\gamma'} = \langle\tau\rangle_{\gamma}$.

(4) and **(5)**. Let $n = \texttt{NewView}(w+1, \mathsf{V})$ be a well-formed new-view proposal. Let C be the set of $\mathbf{nf} - \mathbf{f}$ non-faulty replicas that executed m' (proof for **(4)**) or committed m' (proof for **(5)**), let S be the set of **nf** distinct replicas that signed the $\texttt{ViewState}$ messages included in V, and let $T = S \setminus \mathcal{F}$ be the non-faulty replicas in S. By construction, we have $|T| \geq \mathbf{nf} - \mathbf{f}$. Hence, using the same contradiction argument as used in the proof of Theorem 4.1(1), we can conclude that $(C \cap T) \neq \emptyset$. Let $\mathrm{R} \in (C \cap T)$ be such a non-faulty replica that executed m' (proof for **(4)**) or committed m' (proof for **(5)**) and that signed a message $\texttt{ViewState}(v, \mathbb{C}(m_{\mathrm{R}}), \mathsf{E}) \in \mathsf{V}$ with $m_{\mathrm{R}} = \texttt{Propose}(\langle\tau_{\mathrm{R}}\rangle_{\gamma_{\mathrm{R}}}, v_{\mathrm{R}}, \rho_{\mathrm{R}})$. Let v'' be the last view in which R executed transactions. As R executed m', we have $v \leq v' \leq v'' \leq w$.

If $v' = v''$, then R is guaranteed to execute m' in view v'' (proof of **(4)**) or to commit m' in view v'' (proof of **(5)**). If $v' < v''$, then R used a well-formed new-view message $n'' = \texttt{NewView}(v'', \mathsf{V}_{v''})$ to enter view $v'' \leq w$ (Line 22 of Figure 8). For the proof of **(4)**, we apply induction hypothesis **(4)** on n'' to conclude that $\rho \leq \mathsf{LP}(n'')$. As R executed transactions in view v'', R is guaranteed to have executed some proposal for round ρ while updating its internal state (Line 23 of Figure 8). For the proof of **(5)**, we apply induction hypothesis **(5)** on n'' to conclude that $\rho \leq \mathsf{LC}(n'')$. As R executed transactions in view v'', R is guaranteed to have committed some proposal for round ρ while updating its internal state (Line 23 of Figure 8).

If R is guaranteed to have committed some proposal m'' in round ρ of view v'' or in round ρ when entering view v'', then we must have $\rho \leq \rho_{\mathrm{R}}$ as non-faulty replicas commit proposals in order (Line 15 of Figure 6) and m_{R} is the last proposal R committed (Line 12 of Figure 8). Hence, by the definition of $\mathsf{LC}(n)$ and $\mathsf{LP}(n)$, we have $\rho \leq \rho_{\mathrm{R}} \leq \mathsf{LC}(n) \leq \mathsf{LP}(n)$.

If R is guaranteed to have executed some proposal m'' in round ρ of view v'' or in round ρ when entering view v'' without committing m'', then $\rho > \rho_{\mathrm{R}}$. As R executed m'', R must have stored a prepared certificate $\mathbb{P}(m'')$ (Line 16 of Figure 3) and we conclude $\mathbb{P}(m'') \in \mathsf{E}$ (Line 12 of Figure 8). By the definition of $\mathsf{P}(n, \rho)$ and the definition of $\mathsf{LP}(n)$, we conclude $m \in \mathsf{P}(n, \rho)$ and $\rho \leq \mathsf{LP}(n)$.

(6) As $\mathbf{nf} - \mathbf{f}$ non-faulty replicas executed m in view v, we have $\rho \leq \mathsf{LP}(n)$ by **(5)**. Hence, $\mathcal{L}(n)[\rho]$ is defined.

If $\rho \leq \mathsf{LC}(n)$, then, by the definition of $\mathcal{L}(n)$ and $\mathsf{C}(\rho)$, we have $\mathcal{L}(n)[\rho] = \langle\tau_n\rangle_{\gamma_n}$ with $\langle\tau_n\rangle_{\gamma_n} \in \mathsf{C}(\rho)$ and there exists a commit certificate $\mathbb{C}(m_n)$ for some proposal $m_n = \texttt{Propose}(\langle\tau_n\rangle_{\gamma_n}, v_n, \rho)$. By **(1)**, we conclude that $v \leq v_n$. Due to Theorem 4.4(3), there exists a prepared certificate $\mathbb{P}(m_n)$ and, by **(3)**, we conclude $\langle\tau_n\rangle_{\gamma_n} = \langle\tau\rangle_{\gamma}$.

If $\mathsf{LC}(n) < \rho \leq \mathsf{LP}(n)$, then, by the definition of $\mathcal{L}(n)$, $\mathsf{M}(n, \rho)$, and $\mathsf{P}(n, \rho)$, we have $\mathcal{L}(n)[\rho] = \langle\tau_n\rangle_{\gamma_n}$ with $\langle\tau_n\rangle_{\gamma_n} \in \mathsf{M}(n, \rho)$ and there exists a prepared certificate $\mathbb{P}(m_n)$ for some proposal $m_n = \texttt{Propose}(\langle\tau_n\rangle_{\gamma_n}, v_n, \rho)$ with $m_n \in \mathsf{P}(n, \rho)$. As $\mathbf{nf} - \mathbf{f}$ replicas executed m, we can use the same reasoning as in the proof of **(4)** to obtain a non-faulty replica R that executed m and signed a message $\texttt{ViewState}(v, \mathbb{C}(m_{\mathrm{R}}), \mathsf{E}) \in \mathsf{V}$ with $m_{\mathrm{R}} = \texttt{Propose}(\langle\tau_{\mathrm{R}}\rangle_{\gamma_{\mathrm{R}}}, v_{\mathrm{R}}, \rho_{\mathrm{R}})$. Let v'' be the last view in which R executed transactions. As R executed m, we have $v \leq v'' \leq w$.

If $v = v''$, then R executed some proposal $m'' = m$ in view v''. If $v' < v''$, then R used some new-view message $n'' = \texttt{NewView}(v'', \mathsf{V}_{v''})$ to enter view $v'' \leq w$ (Line 22 of Figure 8). As R executed transactions in view v'', R is guaranteed to have executed some proposal m'' for round ρ while updating its internal state (Line 23 of Figure 8). As $\mathsf{LC}(n) < \rho$, R did not commit m'' in round ρ ,

$\mathbb{P}(m'') \in \mathsf{E}$, and $m'' \in \mathsf{P}(n, \rho)$. By the definition of $\mathsf{M}(n, \rho)$, we conclude $v \leq v_n$. By **(3)**, we conclude $\langle \tau_n \rangle_{\gamma_n} = \langle \tau \rangle_\gamma$. □

Theorem 4.10 not only proves that the view-change protocol satisfies Goal G1, it also proves that Assumption 4.8 holds.

Corollary 4.11 (Assumption 4.8). *If there exists a commit certificate $\mathbb{C}(m)$ for proposal $m = \mathtt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$, then every commit certificate $\mathbb{C}(m')$ for round ρ is a commit certificate for a proposal proposing $\langle \tau \rangle_\gamma$.*

Proof. By Theorem 4.4(3), at-least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas stored a prepared certificate for m and executed m whenever a commit certificate $\mathbb{C}(m)$ exists. Let $m' = \mathtt{Propose}(\langle \tau' \rangle_{\gamma'}, v', \rho)$, $v' \leq v$, be the first proposal for round ρ that is executed by $\mathbf{nf} - \mathbf{f}$ non-faulty replicas. By Theorem 4.10(3), we conclude that $\langle \tau' \rangle_{\gamma'} = \langle \tau \rangle_\gamma$. □

Theorem 4.10 and Corollary 4.11 assure that one can derive a unique ledger from each well-formed new-view proposal $n = \mathtt{NewView}(v + 1, \mathsf{V})$. To resume the normal-case protocol, we also need the guarantee that each non-faulty replica derives the *same* ledger from the new-view proposal they receive, even if they receive different new-view proposals. For rounds for which a commit certificate exists, Theorem 4.10 already provides this guarantee, whereas for rounds for which only prepared certificates exists, the *repropose* mechanisms will enforce this guarantee. Finally, to resume the normal-case protocol, each individual replica also needs to be able to derive from n the exact content of $\mathcal{L}(n)$. Unfortunately, the message n itself only contains all necessary information to derive $\mathcal{L}(n)[\rho]$ for all ρ, $\mathsf{LC}(n) \leq \rho \leq \mathsf{LP}(n)$. Fortunately, also the remainder of $\mathcal{L}(n)$ can be derived:

Lemma 4.12. *Let $n = \mathtt{NewView}(v + 1, \mathsf{V})$ be a well-formed new-view proposal. If communication is reliable, then every non-faulty replica can derive $\mathcal{L}(n)$ (this independent of their internal state).*

Proof. Let R be a non-faulty replica that receives n and is unaware of a client request $\mathcal{L}(n)[\rho]$, $\rho < \mathsf{LC}(n)$. By the definition of $\mathsf{LC}(n)$, there exists a message $m_{\mathsf{vs}} = \mathtt{ViewState}(v, \mathbb{C}(m), \mathsf{E})$ with $m = \mathtt{Propose}(\langle \tau \rangle_\gamma, w, \mathsf{LC}(n))$, $w \leq v$. By Theorem 4.4(3), the existence of $\mathbb{C}(m)$ assures that at-least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas stored commit certificates for proposals $m' = \mathtt{Propose}(\langle \tau' \rangle_{\gamma'}, w', \rho)$ for all rounds ρ, $\rho < \mathsf{LC}(n)$.

Hence, R can query all replicas for the transaction they committed in round ρ. To do so, R uses the query protocol outlined in Figure 10. As the first step, R broadcasts a message $\mathtt{QueryCC}(\rho)$ to all replicas (Line 1 of Figure 10). Non-faulty replicas will respond with the message $\mathtt{RespondCC}(\mathbb{P}(m''), \mathbb{C}(m''))$ if they committed some proposal m'' in round ρ and will not respond otherwise (Line 12 of Figure 10). By Corollary 4.11, no commit certificates for round ρ can exist that proposes a request other than $\langle \tau' \rangle_{\gamma'}$. Hence, using any of the received messages $\mathtt{RespondCC}(\mathbb{C}(m''))$, R will be able to prepare, execute, and commit a proposal for $\langle \tau' \rangle_{\gamma'}$ in round ρ (Line 4 of Figure 10). □

The View-Change Protocol Satisfies Goal G2. In Lemma 4.5 and Lemma 4.7, we already outlined the main timing-based synchronization steps used by the view-change protocol to guarantee Goal G2. This mechanism relies on the availability of some message delay value δ known to all non-faulty replicas that is long enough to guarantee the delivery of any message used by PoE (see also Assumption 4.6). In practice, non-faulty replicas are not aware of this bound and must use some internal message delay estimate.

Let R be a non-faulty replica that uses internal message delay estimate $\Delta(v, \mathrm{R})$ in view v. If $\Delta(v, \mathrm{R})$ is too small, then R can erroneously detect failure of view $v + 1$ (e.g., at Line 14 of Figure 8). To assure that R will not erroneously detect failure of consecutive views, R needs to eventually use an

Query role (running at replica R) **:**

1: Broadcast QueryCC(ρ) to all replicas.
2: **event** R receives a well-formed message RespondCC($\mathbb{P}(m), \mathbb{C}(m)$) **do**
3: Let m = Propose($\langle\tau\rangle_\gamma, v', \rho'$) and $\mathbb{P}(m) = \{m_1, \ldots, m_{\text{nf}}\}$.
4: **if** R did not commit in round ρ' **then**
5: Wait until all previous rounds have a commit certificate.
6: Prepare m and execute τ using Lines 16–18 of Figure 3 with Prepare messages $m_1, \ldots, m_{\text{nf}}$.
7: Store *commit certificate* $\mathbb{C}(m)$.
8: **end if**
9: **end event**

Query response role (running at every replica Q $\in \mathfrak{R}$) **:**

10: **event** Q receives messages QueryCC(ρ) from R **do**
11: **if** Q stored prepared certificate $\mathbb{P}(m)$ and commit certificate $\mathbb{C}(m)$
for some proposal m = Propose($\langle\tau\rangle_\gamma, v, \rho$) **then**
12: Send RespondCC($\mathbb{P}(m), \mathbb{C}(m)$) to replica R.
13: **end if**
14: **end event**

Fig. 10. The query protocol of Lemma 4.12 used by replicas R $\in \mathfrak{R}$ when recovering a missing committed proposal for round ρ.

internal message delay estimate $\Delta(v + i, \text{R})$ in view $v + i$ for which $\Delta(v + i, \text{R}) \geq \delta$ holds. To do so, we assume that all non-faulty replicas Q have some *backoff function* $f(i)$ for which $f(i) \geq i$ holds and use the internal message delay estimate $\Delta(v + i, \text{Q}) = f(i) \cdot \Delta(v, \text{Q})$. We have:

Lemma 4.13. *Let $\Delta(v, \text{R})$ be the internal message delay estimate of replica R at view v, let $f(i)$ be the backoff function used by R, and let δ be a message delay value. There exists a j such that $\Delta(v + j, \text{R}) \geq \delta$.*

Proof. By definition, we have $\Delta(v + i, \text{R}) = f(i) \cdot \Delta(v, \text{R}) \geq i \cdot \Delta(v, \text{R})$. Hence, for all $j \geq \left\lceil \frac{\delta}{\Delta(v,\text{R})} \right\rceil$, we have $\Delta(v + j, \text{R}) \geq \delta$. □

Frequently, an exponential function is used as the backoff function, leading to so-called *exponential backoff*:

Example 4.14. Consider a system with $\mathfrak{R} = \{\text{R}_1, \text{R}_2, \text{R}_3, \text{R}_4\}$ in an environment with a message delay of $\delta = 30$ ms. Assume that all replicas are non-faulty and that we have the following internal message delay estimates in view v:

$$\Delta(v, \text{R}_1) = 7\,\text{ms}; \qquad \Delta(v, \text{R}_2) = 3\,\text{ms};$$
$$\Delta(v, \text{R}_3) = 5\,\text{ms}; \qquad \Delta(v, \text{R}_4) = 1\,\text{ms},$$

and assume that all replicas use the *exponential backup function* $f(i) = 2^i$. We have:

$$\Delta(v + 3, \text{R}_1) = 2^3 \cdot 7 = 56\,\text{ms}; \qquad \Delta(v + 3, \text{R}_2) = 2^3 \cdot 3 = 24\,\text{ms};$$
$$\Delta(v + 3, \text{R}_3) = 2^3 \cdot 5 = 40\,\text{ms}; \qquad \Delta(v + 3, \text{R}_4) = 2^3 \cdot 1 = 8\,\text{ms}.$$

Hence, within three *backoffs*, two out of four replicas have an internal message delay estimate that is sufficiently large. The other replicas will require four and five backoffs, respectively.

Next, we shall use backoff-based internal message delay estimates to prove that the view-change protocol satisfies Goal G2.

Theorem 4.15. *If non-faulty replicas use backoff-based internal message delay estimates, the normal-case protocol was disrupted, and communication becomes reliable, then the view-change protocol guarantees that the normal-case protocol will eventually be reestablished.*

Proof. Assume the normal-case protocol was disrupted in some view v. Consequently, non-faulty replicas will perform view-changes for consecutive views $v' \geq v$ until the normal-case protocol is reestablished (Line 14 of Figure 8). During the view-change for such view v', non-faulty replicas periodically rebroadcast their messages $\texttt{Failure}(v')$ until they reach the new-view proposal stage for v' (Line 3 of Figure 8). Hence, each non-faulty replica will eventually broadcast messages $\texttt{Failure}(w)$ for some view $w \geq v$ whenever communication becomes reliable. As all **nf** non-faulty replicas are guaranteed to receive these messages from all **nf** non-faulty replicas when communication becomes reliable, all non-faulty replicas will enter the new-view proposal stage for consecutive views $v' \geq v$ when communication becomes reliable (Line 9 of Figure 8).

Assume that communication becomes reliable with message delay δ unknown to all non-faulty replicas. Without loss of generality, we can assume that each non-faulty replica $\mathrm{R} \in (\mathfrak{R} \setminus \mathcal{F})$ uses a backoff function f_{R} and uses the internal message delay estimate $\Delta(v+i, \mathrm{R}) = f_{\mathrm{R}}(i) \cdot \Delta(v, \mathrm{R})$ in view $v+i$. Hence, by Lemma 4.13, there exists a j_{R} such that, for all views $j \geq j_{\mathrm{R}}$, $\Delta(v+j, \mathrm{R}) \geq \delta$. Consequently, in all views $v+j'$ with $j' \geq \max\{j_{\mathrm{R}} \mid \mathrm{R} \in (\mathfrak{R} \setminus \mathcal{F})\}$, the internal message delay estimate of all non-faulty replicas is at-least δ.

Choose $i \geq \max\{j_{\mathrm{R}} \mid \mathrm{R} \in (\mathfrak{R} \setminus \mathcal{F})\}$ such that view $v+i$ is a view for which all non-faulty replicas entered the new-view proposal stage after communication became reliable. Let P_j be the primary of view $v+i+j$ (with $\mathrm{id}(\mathrm{P}_j) = (v+i+j) \bmod \mathbf{n}$). Finally, choose the first view $v+i+\sigma$ for which P_σ is non-faulty. As there are at-most **f** faulty replicas, we must have $0 \leq \sigma \leq \mathbf{f}$. If the view-change protocol reestablish the normal-case protocol in any view before view $v+i+\sigma$, then the statement of this theorem voidly holds.

To complete the proof, we assume that the view-change protocol did not reestablish the normal-case protocol before view $v+i+\sigma$ and we show that the view-change protocol will reestablish the normal-case protocol for view $v+i+\sigma$. Let $t(\mathrm{R})$ be the time at which a non-faulty replica $\mathrm{R} \in (\mathfrak{R} \setminus \mathcal{F})$ enters the new-view proposal stage of view $v+i+\sigma-1$ and let $t_{\min} = \min\{t(\mathrm{R}) \mid \mathrm{R} \in (\mathfrak{R} \setminus \mathcal{F})\}$ be the time at which the first non-faulty replica enters this new-view proposal stage. By Lemma 4.7, all replicas will receive an identical new-view proposal $\texttt{NewView}(v+i+\sigma, \mathrm{V})$ (Line 3 of Figure 8) at-or-before $t_{\min} + 4\delta$. Finally, let Q be any non-faulty replica. Due to the internal message delay estimate used by Q, replica Q will expect $\texttt{NewView}(v+i+\sigma, \mathrm{V})$ before

$$t(\mathrm{Q}) + 4\Delta(v+i+\sigma, \mathrm{Q}) \geq t + 4\Delta(v+i+\sigma, \mathrm{Q}).$$

As $i \geq j(\mathrm{R})$, we can conclude $t_{\min} + 4\Delta(v+i+\sigma, \mathrm{Q}) \geq t + 4\delta$, that Q receives $\texttt{NewView}(v+i+\sigma, \mathrm{V})$, and that Q enters the new-view accept stage. Hence, all non-faulty replicas will enter the new-view accept stage with the same new-view proposal $\texttt{NewView}(v+i+\sigma, \mathrm{V})$, will update their internal state accordingly, and reestablish the normal-case protocol in view $v+i+\sigma$. □

Next, we further illustrate Theorem 4.15 in an environment with exponential backoff.

Example 4.16. Consider the situation of Example 4.14. Due to some disruptions, view v fails and all replicas participate in consecutive view-changes (until one of these view-changes succeeds). Let $t_j(\mathrm{R}_i)$ be the time at which replica R_i, $1 \leq i \leq 4$, enters the new-view proposal phase for view $v+j$ and let $t_{j,\min} = \min\{t_j(\mathrm{R}_i) \mid 1 \leq i \leq 4\}$ be the time at which the first replica enters the new-view proposal phase for view $v+j$.

If all replicas are non-faulty, then one can expect a valid new-view proposal for view $v+j+1$ at $t_{j,\min} + 4\delta$, while replica R_i, $1 \leq i \leq 4$, will expect a valid new-view proposal for view $v+j+1$

before

$$t_j(\mathrm{R}_i) + 4\Delta(v + j, \mathrm{R}_i) \geq t_j + 4 \cdot 2^j \cdot \Delta(v, \mathrm{R}_i).$$

Consequently, the new-view proposal for view $v + j + 1$ is guaranteed to arrive on time for replica R_i whenever $t_{j,\min} + 4 \cdot 2^j \cdot \Delta(v, \mathrm{R}_i) \geq t_{j,\min} + 4\delta$ and, hence, when $2^j \cdot \Delta(v, \mathrm{R}_i) \geq \delta$, which simplifies to $j \geq \max(\log_2 \left\lceil \frac{\delta}{\Delta(v,\mathrm{R}_i)} \right\rceil, 1)$.[5] Filling in yields

$$j \geq \log_2 \left\lceil \frac{\delta}{\Delta(v,\mathrm{R}_1)} \right\rceil = \log_2 \left\lceil \tfrac{30}{7} \right\rceil = 3 \qquad \text{(for replica } \mathrm{R}_1\text{)};$$

$$j \geq \log_2 \left\lceil \frac{\delta}{\Delta(v,\mathrm{R}_2)} \right\rceil = \log_2 \left\lceil \tfrac{30}{3} \right\rceil = 4 \qquad \text{(for replica } \mathrm{R}_2\text{)};$$

$$j \geq \log_2 \left\lceil \frac{\delta}{\Delta(v,\mathrm{R}_3)} \right\rceil = \log_2 \left\lceil \tfrac{30}{5} \right\rceil = 3 \qquad \text{(for replica } \mathrm{R}_3\text{)};$$

$$j \geq \log_2 \left\lceil \frac{\delta}{\Delta(v,\mathrm{R}_4)} \right\rceil = \log_2 \left\lceil \tfrac{30}{1} \right\rceil = 5 \qquad \text{(for replica } \mathrm{R}_4\text{)}.$$

We conclude that within at-most five consecutive view-changes, all replicas will have a sufficiently large internal message delay estimate to guarantee a successful view-change.

Remark 4.17. Theorem 4.15 will use arbitrarily large internal message delay estimates to assure that all non-faulty replicas are eventually sufficiently synchronized. In most practical deployments, one can utilize a reasonable upper bound on any message estimate (e.g., 10 s when sending small messages over a wide-area network), as this upper bound will always hold whenever the network is operating correctly (and communication is reliable).

4.5 PoE **Provides Consensus**

In Sections 4.1–4.4, we have laid out the design of PoE. From the details presented, one can already derive how PoE provides non-divergence (Theorem 4.1(1) and Theorem 4.10) and termination (Theorem 4.4(1) and Theorem 4.15). To prove that PoE provides weak consensus, we also need to detail how PoE provides *non-triviality*. To do so, we introduce a mechanism that allows non-faulty clients to force replication of their transactions whenever communication is sufficiently reliable.

Consider a client γ that wants to request transaction τ. In the normal-case protocol, it is assumed that γ knows primary P of the current view, in which case γ will simply send $\langle \tau \rangle_\gamma$ to P. If this primary P is non-faulty and communication is reliable, then this normal-case protocol will assure a proof-of-execution (Theorem 4.1(2)). To deal with deviations of the normal-case (e.g., an unknown primary or a faulty primary), PoE allows client γ to send its request $\langle \tau \rangle_\gamma$ to *any* replica R. If R is non-faulty, then it will forward $\langle \tau \rangle_\gamma$ to the primary of the current view, after which R expects a timely proposal of $\langle \tau \rangle_\gamma$ via the normal-case protocol. If no such timely proposal arrives, R will detect failure of the current view (Line 1 of Figure 8), and, after a successful view-change, forward $\langle \tau \rangle_\gamma$ to the next primary.

Using the above request-forward mechanism, clients can force consensus on requests whenever communication is sufficiently reliable: to force consensus on request $\langle \tau \rangle_\gamma$, client γ simply sends this request to all replicas. Consequently, all non-faulty replicas will forward $\langle \tau \rangle_\gamma$ to the current primary and expect a timely proposal of $\langle \tau \rangle_\gamma$. If this does not happen, then all non-faulty replicas will detect failure of the current view. As there are **nf** non-faulty replicas, this is sufficient to trigger a view-change (Line 9 of Figure 8). After a new view is established, the non-faulty replicas forward $\langle \tau \rangle_\gamma$ to the next primary, this until the request is proposed and executed.

To prevent abuse of the request-forward mechanism by malicious clients, PoE uses two heuristics. First, malicious clients can try to suppress requests by other clients by continuously sending requests to all replicas. To deal with such behavior, non-faulty replicas can limit the rate at which they

[5] If $\delta < \Delta(v, \mathrm{R}_i)$, then $\left\lceil \frac{\delta}{\Delta(v,\mathrm{R}_i)} \right\rceil \leq 0$. We always do at-least $j = 1$ view changes, however.

forward requests of a single client to the primary (assuring that the primary has the room to timely propose requests for all clients). Second, malicious clients can send distinct, conflicting, requests to different replicas. In this case, the primary has to choose one of these transactions to propose first, due to which the primary might be unable to propose all client requests forwarded to it within a timely fashion. To prevent failure detection of the view due to this malicious client behavior, any replica R that forwarded a request of client γ to the current primary will consider any subsequent proposal of a transaction requested by γ as a timely proposal by the primary (even if that proposal does not match the request forwarded by R).

Using the request-forward mechanism, we are finally able to prove that PoE provides weak consensus:

Theorem 4.18. *If* PoE *is operated in a system with* $\mathbf{n} > 3\mathbf{f}$*, then* PoE *provides weak consensus.*

Proof. We say that a non-faulty replica R decides on $\langle\tau\rangle_\gamma$ in round ρ whenever it stores a commit certificate $\mathbb{C}(m)$ with $m = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$. Due to Theorem 4.4(3), the existence of $\mathbb{C}(m)$ implies that at-least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas executed m. Using these facts, we will prove that PoE provides the three guarantees of consensus.

Termination *(in periods of reliable communication).* Due to Theorem 4.4(3), the existence of $\mathbb{C}(m)$ implies that at-least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas executed m. If communication is reliable, then Theorem 4.4(2) guarantees termination. If communication is unreliable and leads to a view failure, then Theorem 4.15 assures that a new view, view w, will be established whenever communications becomes reliable and Theorem 4.10(6) and Lemma 4.12 assure that all replicas participating in view w will recover $\langle\tau\rangle_\gamma$ as the ρ-th decision.

Now consider non-faulty replica Q that is not able to participate in the new view w, e.g., due to faulty behavior of the primary of view w. Replica Q can make a decision for round ρ in two ways. First, if the primary of view w can propose in round ρ, then either a proposal $m' = \texttt{Propose}(\langle\tau\rangle_\gamma, w, \rho)$ gets committed by a non-faulty replica, in which case Theorem 4.4(1) guarantees that Q commits m', or the failure of view w will be detected. If the primary of view w cannot propose in round ρ, then Q can use Lemma 4.12 to derive $\langle\tau\rangle_\gamma$ as the ρ-th request whenever Q detects the existence of a commit certificate for round ρ. Replica Q can detect the existence of commit certificates for round ρ, $\rho < \rho'$, after receiving well-formed `CheckCommit` messages for round ρ' from at-least a single non-faulty replica (by receiving such messages from $\mathbf{f} + 1$ distinct replicas).

Non-Divergence. Let v' be the first view in which a proposal $m' = \texttt{Propose}(\langle\tau'\rangle_{\gamma'}, v', \rho)$ was executed by $\mathbf{nf} - \mathbf{f}$ non-faulty replicas. By Theorem 4.10(1), $v' \leq v$. Hence, by Theorem 4.10(3), we must have $\langle\tau'\rangle_{\gamma'} = \langle\tau\rangle_\gamma$.

Non-Triviality *(in periods of reliable communication).* We say that a decision needs to be made for request $\langle\tau\rangle_\gamma$ of client γ whenever γ is non-faulty. In this case, a non-faulty replica Q learns that a decision needs to be made for $\langle\tau\rangle_\gamma$ whenever it receives $\langle\tau\rangle_\gamma$. As non-faulty clients will repeatedly send $\langle\tau\rangle_\gamma$ to all replicas whenever they do not receive a proof-of-execution for τ, every non-faulty replica will receive $\langle\tau\rangle_\gamma$ when communication becomes reliable. Assume that communication becomes reliable in view v. Either the primary of view v will propose $\langle\tau\rangle_\gamma$, or all non-faulty replicas will trigger view changes until reaching a view whose primary will propose $\langle\tau\rangle_\gamma$. As there are at-most $\mathbf{f}$ faulty replicas, at-most $\mathbf{f}$ such view changes will happen before $\langle\tau\rangle_\gamma$ is proposed. □

Remark 4.19. PoE is designed to be able to deal with many types of failures, e.g., Byzantine failures, network failures, and crashes. First, as proven in Theorem 4.18, PoE provides both *consensus* and *client services* if at-most $\mathbf{f}$ concurrent replicas fail (e.g., Byzantine replicas or crashed replicas),

$\mathbf{n} > 3\mathbf{f}$. Second, as a consequence of Theorem 4.1(1) and Theorem 4.10, PoE will always provide *non-divergence* if at-most $\mathbf{f}$ replicas are Byzantine. Third, PoE can recover form any number of non-Byzantine replica crashes *as long as* these crashes are *recoverable*: if replicas store any certificates they produce in permanent store (Line 16 of Figure 3 and Line 16 of Figure 6) such that they can always recover these certificates after a crash, then non-Byzantine crashes can only lead to service disruption (similar to network failure), and service will be recovered when sufficient replicas have recovered.

4.6 PoE **Provides Client Service**

Theorem 4.18 proves that PoE provides weak consensus. Hence, due to the non-triviality guarantee of PoE (as proven in Theorem 4.18), every client can force a consensus decision on their requests. Hence, to provide *client services*, we only need to show that PoE guarantees that a consensus decision for request $\langle \tau \rangle_\gamma$ will lead to a proof-of-execution for τ. If communication is reliable and no view changes happen, then either Theorem 4.1(2) or Theorem 4.4(2) guarantees such proof-of-execution. Due to the speculative design of PoE, view changes can result in consensus decisions *without* an accompanying proof-of-execution:

Example 4.20. Consider a system with $\mathbf{n} = 3\mathbf{f} + 1$ replicas. We partition the non-faulty replicas in sets A, B, and C with $|A| = |B| = \mathbf{f}$ and $|C| = 1$. Now let $\langle \tau \rangle_\gamma$ be a client request and consider the following sequence of events in round ρ of views v, $v + 1$, and $v + 2$.

View v. Due to *unreliable communication* the $\mathbf{f}$ replicas in B become unreachable while the primary P_v of view v proposes $m_v = \texttt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$. Hence, only the $\mathbf{nf}$ replicas in $A \cup C \cup \mathcal{F}$ receive m_v and exchange `Prepare` messages. Consequently, the $\mathbf{nf} - \mathbf{f}$ non-faulty replicas in $A \cup C$ are able to execute τ and inform the client γ. The faulty replicas in $\mathcal{F}$ decide to *not* inform the client. Hence, the client γ only receives $\mathbf{f} + 1$ messages of the form $\texttt{Inform}(\langle \tau \rangle_\gamma, v, \rho, r)$, which is insufficient for a proof-of-execution.

Next, the replicas in $A \cup C$ exchange `CheckCommit` messages, while the faulty replicas only send `CheckCommit` messages to A. Consequently, the replicas in A are able to construct commit certificates $\mathbb{C}(m_v)$, while the replica in C is unable to do so. When communication becomes reliable, the $\mathbf{f} + 1$ non-faulty replicas in $B \cup C$ will be able to successfully trigger a view change.

View $v + 1$. Due to *unreliable communication*, the $\mathbf{f}$ non-faulty replicas in A become unreachable. Hence, the primary P_{v+1} of view $v + 1$ ends up proposing a new view based on the information provided by the replicas in $B \cup C \cup \mathcal{F}$. In specific, the non-faulty replicas in B only provide information on rounds before ρ, the non-faulty replica in C provides a prepared certificate for m_v, and the faulty replicas in $\mathcal{F}$ *lie* and only provide information on rounds before ρ. Hence, any replica that enters view $v + 1$ will rollback transaction τ and will expect the primary P_{v+1} to propose $m_{v+1} = \texttt{Propose}(\langle \tau \rangle_\gamma, v + 1, \rho)$.

After primary P_{v+1} proposes m_{v+1}, the $\mathbf{nf}$ replicas in $B \cup C \cup \mathcal{F}$ receive m_{v+1} and exchange `Prepare` messages. Consequently, the $\mathbf{nf} - \mathbf{f}$ non-faulty replicas in $B \cup C$ are able to execute τ and inform the client γ. The faulty replicas in $\mathcal{F}$ once again decide to *not* inform the client. Hence, at this point, the client received $\mathbf{f} + 1$ messages of the form $\texttt{Inform}(\langle \tau \rangle_\gamma, v, \rho, r)$ and $\mathbf{f} + 1$ messages of the form $\texttt{Inform}(\langle \tau \rangle_\gamma, v + 1, \rho, r)$, which is still insufficient for a proof-of-execution.

Next, the replicas in $B \cup C$ exchange `CheckCommit` messages, while the faulty replicas only send `CheckCommit` messages to B. Consequently, the replicas in B are able to construct commit certificates $\mathbb{C}(m_{v+1})$, while the replica in C is unable to do so. When communication becomes reliable, the $\mathbf{f} + 1$ non-faulty replicas in $A \cup C$ will be able to successfully trigger a view change.

View $v+2$. The (faulty) primary P_{v+2} of view $v+2$ will receive commit certificates $\mathbb{C}(m_v)$ from the replicas in A, commit certificates $\mathbb{C}(m_{v+1})$ from the replicas in B, a prepared certificate for m_{v+1} from C, and, again, the faulty replicas in $\mathcal{F}$ *lie* and only provide information on rounds before ρ. Using this information, the primary P_{v+2} decides to construct *two* distinct new-view proposals n_A and n_B that are based on the information provided by the **nf** replicas in $A \cup C \cup \mathcal{F}$ and $B \cup C \cup \mathcal{F}$, respectively.

The replicas in A receive n_A, the replicas in B receive n_B, and replica C also receives n_B (due to which C only has to store an additional commit certificate). Hence, after the view-change, the **f** replicas in A hold commit certificates for $m_v = \texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$ while the $\mathbf{f}+1$ replicas in $B \cup C$ hold commit certificates for $m_{v+1} = \texttt{Propose}(\langle\tau\rangle_\gamma, v+1, \rho)$. Hence, at-most $\mathbf{f}+1$ non-faulty replicas hold identical commit certificates for round ρ, while *all* non-faulty replicas hold a commit certificate for a proposal proposing $\langle\tau\rangle_\gamma$ in round ρ. Consequently, no non-faulty replica will send additional `Inform` messages for the request $\langle\tau\rangle_\gamma$ to client γ. As such, the client γ fails to obtain a proof-of-execution for a fully-decided request.

As all non-faulty replicas start view $v+2$ with a commit certificate for round ρ, the (faulty) primary P_{v+2} does not have to propose requests for round ρ and can directly propose for round $\rho+1$. As the final step in this example, primary P_{v+2} successfully proposes $m = \texttt{Propose}(\langle\tau'\rangle_{\gamma'}, v+2, \rho+1)$. Due to this proposal, all non-faulty replicas end up with commit certificates $\mathbb{C}(m)$ for round $\rho+1$, thereby assuring that future view changes will omit any information on round ρ and, thus, assuring that no proof-of-execution will be produced for round ρ in future views.

To deal with the issue raised in Example 4.20, we utilize a recovery protocol by which clients can obtain a proof-of-commit that *implies* a proof-of-execution. This recovery protocol is based on a straightforward observation: if a client γ can observe that a non-faulty replica *committed* a proposal $\texttt{Propose}(\langle\tau\rangle_\gamma, v, \rho)$ for request $\langle\tau\rangle_\gamma$, then, by Theorem 4.18, this observation provides proof that all non-faulty replicas will eventually execute $\langle\tau\rangle_\gamma$ in round ρ.

Note that as part of providing non-triviality, a client γ that requires execution for a request $\langle\tau\rangle_\gamma$ will eventually send $\langle\tau\rangle_\gamma$ to all replicas, this to enforce an eventual consensus decision. When a non-faulty replica R receives $\langle\tau\rangle_\gamma$ after the replica stored a commit certificate for a proposal proposing $\langle\tau\rangle_\gamma$, then, as the first step of the proof-of-commit recovery protocol, replica R responds with a message $\texttt{InformCC}(\langle\tau\rangle_\gamma, \rho, r)$, in which r was the original execution result R obtained while executing τ in round ρ. As the second step of the proof-of-commit recovery protocol, client γ will considers τ executed after it receives a *proof-of-commit* for $\langle\tau\rangle_\gamma$ consisting of identical $\texttt{InformCC}(\langle\tau\rangle_\gamma, \rho, r)$ messages from $\mathbf{f}+1$ distinct replicas. The pseudo-code for the proof-of-commit recovery protocol can be found in Figure 11.[6]

Using the proof-of-commit recovery protocol, we can finally prove that PoE provides client services:

Theorem 4.21. *If* PoE *is operated in a system with* $\mathbf{n} > 3\mathbf{f}$, *then* PoE *provides client service whenever communication is reliable.*

Proof. When communication is reliable and client γ requests $\langle\tau\rangle_\gamma$, then, by Theorem 4.18, client γ can force execution of $\langle\tau\rangle_\gamma$. If client γ receives a (v, ρ)-proof-of-execution for τ, then client γ considers τ executed and, due to Theorem 4.10, the executed state observed by the client will be preserved.

[6]Non-faulty replicas in PoE can opt to *always* send an `InformCC` message to clients after they commit a proposal: `InformCC` messages carry the same information to the client as normal client replies do in consensus protocols such as Pbft. Such a design will trade an increase of network bandwidth for a decrease of latency when replicas are left in the dark.

Client recovery role (used by client γ to force transaction τ) **:**

1: Send $\langle \tau \rangle_\gamma$ to all replicas $\mathrm{R} \in \mathfrak{R}$.
2: Await a *ρ-proof-of-commit for* $\langle \tau \rangle_\gamma$ consisting of identical messages $\texttt{InformCC}(\langle \tau \rangle_\gamma, \rho, r)$ from $\mathbf{f} + 1$ distinct replicas.
3: Considers τ executed, with result r, as the ρ-th transaction.

Replica recovery role (running at every replica $\mathrm{R} \in \mathfrak{R}$) **:**

4: **event** R receives $\langle \tau \rangle_\gamma$ **do**
5: **if** R stored commit certificate $\mathbb{C}(m)$ for some proposal $m = \texttt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$ **then**
6: Let r be the result R obtained while executing τ in round ρ.
7: Send $\texttt{InformCC}(\langle \tau \rangle_\gamma, \rho, r)$ to γ.
8: **end if**
9: **end event**

Fig. 11. The proof-of-commit recovery protocol in PoE.

Next, we consider the case in which client γ does not receive a proof-of-execution for τ. Due to Theorem 4.18, all non-faulty replicas will eventually commit a proposal for $\langle \tau \rangle_\gamma$ in some round ρ. By Theorem 4.4(3) and Corollary 4.11, all non-faulty replicas executed the same sequence of $\rho - 1$ transactions before executing τ. Hence, if client γ resends $\langle \tau \rangle_\gamma$ to the non-faulty replicas, then these non-faulty replicas will eventually all be able to send identical messages $\texttt{InformCC}(\langle \tau \rangle_\gamma, \rho, r)$ to the client (Line 7 of Figure 11). Consequently, the client γ will receive at-least $\mathbf{nf} > \mathbf{f}$ such messages and client γ will consider τ executed (Line 3 of Figure 11). Due to Theorem 4.10, the executed state observed by the client will be preserved. Finally, as there are at-most $\mathbf{f}$ faulty replicas, the faulty replicas can produce at-most $\texttt{InformCC}(\langle \tau \rangle_\gamma, \rho', r')$ messages distinct form $\texttt{InformCC}(\langle \tau \rangle_\gamma, \rho, r)$ and, hence, cannot produce an invalid proof-of-commit. □

Remark 4.22. PoE does not enforce that client requests are proposed *in order*: if a client γ first sends $\langle \tau_1 \rangle_\gamma$ to the current primary P and then sends $\langle \tau_2 \rangle_\gamma$, then nothing prevents P of first proposing $\langle \tau_2 \rangle_\gamma$ in round ρ_2 and then proposing $\langle \tau_1 \rangle_\gamma$ in round ρ_1 ($\rho_2 < \rho_1$). If a client requires ordered execution, then it can simply request τ_2 only after it receives a proof-of-execution of τ_1. If the application that utilizes PoE requires ordered execution of all client requests, then one can require that valid client requests come with a baked-in counter indicating its order and that subsequent proposals use subsequent counter values.

5 On the Complexity of PoE

In Section 4 we provided an in-detail description and proof of correctness of the Proof-of-Execution consensus protocol (PoE). Next, we will take a deep dive into the exact message complexity of this protocol. Furthermore, as we kept the presentation of PoE described in Section 4 as simple as possible, we will also introduce techniques one can employ to obtain PoE variants with a reduced message complexity. We have summarized the complexity of PoE and its variants in Figure 12.

5.1 The Complexity of Standard PoE

As the first step, we take a look at the complexity of the *standard* variant of PoE as it is presented in Section 4. To simplify presentation, we have assumed in Section 4 that replicas send messages to themselves whenever they broadcast messages. In practice, these messages can be eliminated.

In the normal case protocol of Figure 3, the primary broadcasts $\mathbf{n}-1$ messages $m = \texttt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$ whose size is directly determined by the size $\mathbf{C} = \|\langle \tau \rangle_\gamma\|$ of the proposed client request $\langle \tau \rangle_\gamma$. Next, all $\mathbf{n}$ replicas broadcast messages $\texttt{Prepare}(m)$ to all other $(\mathbf{n} - 1)$ replicas, resulting in $\mathbf{n}(\mathbf{n} - 1)$

Element	Example	Size			Amount
		Standard PoE (Section 5.1)	Using Digests (Section 5.2)	Using Threshold Certificates (Section 5.3)	
Client request	$\langle \tau \rangle_\gamma$	$\mathbf{C}$	$\mathbf{C}$	$\mathbf{C}$	1
Propose message	$m = \texttt{Propose}(\langle \tau \rangle_\gamma, v, \rho)$	$O(\mathbf{C})$	$O(\mathbf{C})$	$O(\mathbf{C})$	$\mathbf{n} - 1$
Prepare message	$\texttt{Prepare}(m)$	$O(\mathbf{C})$	$O(1)$	$O(1)$	$\mathbf{n}(\mathbf{n}-1)$ [a]
Inform message	$\texttt{Inform}(\langle \tau \rangle_\gamma, v, \rho, r)$	$O(\mathbf{C} + \|r\|)$	$O(1 + \|r\|)$	$O(1 + \|r\|)$	$\mathbf{n}$
CheckCommit message	$\texttt{CheckCommit}(\mathbb{P}(m))$	$O(\mathbf{nf} + \mathbf{C})$	$O(1)$ [b]	$O(1)$ [b]	$\mathbf{n}(\mathbf{n}-1)$
Failure message	$\texttt{Failure}(v)$	$O(1)$	$O(1)$	$O(1)$	$\mathbf{n}(\mathbf{n}-1)$
ViewState message	$\texttt{ViewState}(v, \mathbb{C}(m), \mathsf{E})$	$O\left(\mathbf{nf}^2 + \mathbf{W}(\mathbf{nf} + \mathbf{C})\right)$	$O(\mathbf{Wnf})$ [b]	$O(\mathbf{W})$ [b]	$(\mathbf{n}-1)$
NewView message	$\texttt{NewView}(v+1, \mathsf{V})$	$O\left(\mathbf{nf}\left(\mathbf{nf}^2 + \mathbf{W}(\mathbf{nf} + \mathbf{C})\right)\right)$	$O\left(\mathbf{Wnf}^2\right)$ [b]	$O(\mathbf{Wnf})$ [b]	$(\mathbf{n}-1)$
QueryCC message	$\texttt{QueryCC}(\rho)$	$O(1)$	$O(1)$	$O(1)$	$\mathbf{n}$ [c]
RespondCC message	$\texttt{RespondCC}(\mathbb{P}(m), \mathbb{C}(m))$	$O\left(\mathbf{nf}^2 + \mathbf{C}\right)$	$O(\mathbf{nf})$ [b]	$O(1)$ [b]	$\mathbf{n}$ [c]
InformCC message	$\texttt{InformCC}(\langle \tau \rangle_\gamma, \rho, r)$	$O(\mathbf{C} + \|r\|)$	$O(1 + \|r\|)$	$O(1 + \|r\|)$	$\mathbf{n}$ [c]
Prepared certificate	$\mathbb{P}(m)$	$O(\mathbf{nf} + \mathbf{C})$	$O(\mathbf{nf})$ [d]	$O(1)$ [d]	
Commit certificate	$\mathbb{C}(m)$	$O\left(\mathbf{nf}^2 + \mathbf{C}\right)$	$O(\mathbf{nf})$ [d]	$O(1)$ [d]	

[a] The Prepare message of the primary can be merged into the Propose it broadcasts, reducing the amount of Prepare messages to $(\mathbf{n}-1)^2$. [b] The complexities noted are for versions that minimize message size at the cost of additional recovery steps for replicas to obtain missing client requests. See Remark 5.2 for details. [c] These messages are only required for individual recovery processes and are not used when communication is sufficiently reliable. [d] As client requests are not part of these certificates, replicas need to store a copy of the client request alongside these certificates.

Fig. 12. The message complexity (of messages exchanged) and storage complexity (of certificates stored) in the standard variants of PoE. We assume that client requests have a size bounded by $\mathbf{C}$, that the primary can propose at-most $\mathbf{W}$ requests out-of-order after the last commit certificate (the window size), and that the execution result r has size $\|r\|$.

messages of size $O(\mathbf{C})$. Finally, all $\mathbf{n}$ replicas respond to the client via messages $\texttt{Inform}(\langle \tau \rangle_\gamma, v, \rho, r)$, whose size is directly determined by the size $\mathbf{C}$ of a client request and the size $\|r\|$ of the result r.

In the check-commit protocol of Figure 6, all replicas broadcast a message $\texttt{CheckCommit}(\mathbb{P}(m))$, in which $\mathbb{P}(m)$ is a prepared certificate for proposal m. This prepared certificate $\mathbb{P}(m)$ consists of $\mathbf{nf}$ messages $\texttt{Prepare}(m)$ signed by $\mathbf{nf}$ distinct replicas. As these $\mathbf{nf}$ messages are identical, only a single copy needs to be included (with size $O(\mathbf{C})$) together with $\mathbf{nf}$ signatures that have a constant size each.[7]

In the view-change protocol of Figure 8, all at-most $\mathbf{n}$ replicas that detect failure of view v broadcast a message $\texttt{Failure}(v)$ of constant size to all other $(\mathbf{n}-1)$ replicas. All at-most $\mathbf{n}$ replicas that enter the new-view proposal stage in view v will send a single message of the form $\texttt{ViewState}(v, \mathbb{C}(m), \mathsf{E})$, with $\mathbb{C}(m)$ a commit certificate and E a set of prepared certificates, to the new primary. The commit certificate $\mathbb{C}(m)$ consists of $\mathbf{nf}$ messages $\texttt{Commit}(\mathbb{P}(m)_i)$, $1 \leq i \leq \mathbf{nf}$, signed by $\mathbf{nf}$ distinct replicas. As each replica can construct a prepared certificate $\mathbb{P}(m)_i$ distinct from all other prepared certificates and the signature on $\texttt{Commit}(\mathbb{P}(m)_i)$ can only be verified when the message $\texttt{Commit}(\mathbb{P}(m)_i)$ is known, a commit certificate needs to store the $\mathbf{nf}$ individual prepared certificates: one can only eliminate $\mathbf{nf} - 1$ redundant copies of the original proposal m. Hence, each commit certificate consists of a proposal m, $\mathbf{nf}^2$ signatures of prepare message $\texttt{Prepare}(m)$, and $\mathbf{nf}$ signatures of messages $\texttt{Commit}(\mathbb{P}(m)_i)$, $1 \leq i \leq \mathbf{nf}$. By Assumption 4.6, the number of prepared certificates in E is upper bounded by some *window size* $\mathbf{W}$. Finally, the new primary will broadcast a

[7] To simplify presentation in Section 4, the CheckCommit messages carry prepared certificates. Without affecting the correctness of PoE or the complexity of recovery, these prepared certificates can be eliminated in favor of constant-sized CheckCommit messages. We refer to Section 5.2 and Section 5.5 for further details.

single message of the form NewView$(v+1, \mathsf{V})$, with V a set of **nf** ViewState messages, to all $(\mathbf{n}-1)$ other replicas.

Finally, in the query protocol of Figure 10, each replica that needs to recover the status of round ρ after a period of unreliable communication will broadcast constant-sized messages of the form QueryCC(ρ) and get responses RespondCC$(\mathbb{P}(m), \mathbb{C}(m))$ with $\mathbb{P}(m)$ a prepared certificate and $\mathbb{C}(m)$ a commit certificate. In the proof-of-commit recovery protocol of Figure 11, each replica will send a message InformCC$(\langle\tau\rangle_\gamma, \rho, r)$ whose size is determined by **C** to clients that are recovering from unreliable communication.

5.2 Reducing Message Sizes with Digests

A close look at PoE shows that in the original design of PoE, full copies of Propose messages are included in Prepare, CheckCommit, ViewState, and NewView messages. Furthermore, full copies of the client request are included in Inform and InformCC messages.

When client requests are large, this will incur a substantial communication cost. The typical way to support large client requests is by replacing client requests $\langle\tau\rangle_\gamma$ by a constant-sized message digest digest$(\langle\tau\rangle_\gamma)$ obtained from $\langle\tau\rangle_\gamma$ using a strong cryptographic hash function digest$(\cdot)$ [33]. To do so, one simply replaces the message Propose$(\langle\tau\rangle_\gamma, v, \rho)$ the primary broadcasts when proposing transaction τ, requested by client γ, as the ρ-th transaction in some view v by the pair $(m_d = \mathsf{Propose}(d, v, \rho), \langle\tau\rangle_\gamma)$ such that $d = \mathsf{digest}(\langle\tau\rangle_\gamma)$. All other messages will use the constant-sized proposal m_d, which only includes the constant-sized digest d, in all messages that contain a Propose message. Furthermore, also the client request $\langle\tau\rangle_\gamma$ in Inform and InformCC messages can be replaced by the constant-sized digest d.

Receipt of a message CheckCommit$(\mathbb{P}(m_d))$ is no longer sufficient to prepare and execute client request $\langle\tau\rangle_\gamma$ (Line 11 of Figure 6): after receiving message CheckCommit$(\mathbb{P}(m_d))$, one only has a digest $d = \mathsf{digest}(\langle\tau\rangle_\gamma)$ of client request $\langle\tau\rangle_\gamma$, but not the client request itself. Fortunately, the existence of prepared certificate $\mathbb{P}(m_d)$ guarantees that at-least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas have sent Prepare(m_d) and, hence, have received a pair $(m_d, \langle\tau\rangle_\gamma)$. Hence, after a replica Q receives prepared certificate $\mathbb{P}(m_d)$, it can query the replicas whose signed prepared certificates are in $\mathbb{P}(m_d)$ for the missing client request. As Q can derive the digest d from $\mathbb{P}(m_d)$, it can verify whether any request $\langle\tau'\rangle_{\gamma'}$ it received in response to its query is valid by verifying whether $d = \mathsf{digest}(\langle\tau'\rangle_{\gamma'})$ (under normal cryptographic assumptions). Similar strategies can be made to recover missing client requests due to NewView messages. As such, the usage of digests to eliminate copies of client requests trades a reduction of message complexity with a more-complex recovery path.

Remark 5.1. Although digests reduce the impact of *very large* client requests on throughput, they do not eliminate the need for a limit on the size of client requests as specified in Assumption 4.6: if the size of client requests is not limited, then non-faulty replicas cannot expect a timely response of *any* forwarded client requests, as the primary could always be busy with sending an arbitrarily large proposal.

The second source of large messages in PoE are CheckCommit messages, which impacts the size of commit certificates and of ViewState, NewView, and RespondCC messages. To further reduce the size of CheckCommit messages, one can replace messages of the form CheckCommit$(\mathbb{P}(m))$, with $\mathbb{P}(m)$ a prepared certificate for proposal m, by messages of the form CheckCommit(m_d) (where m_d is the proposal obtained from m when the client request is represented by a digest). Doing so will reduce the size of a CheckCommit message from $O(\mathbf{nf} + \mathbf{C})$ to $O(1)$. Due to this change, all commit certificates $\mathbb{C}(m_d)$ will consists of **nf** messages CheckCommit(m_d) signed by **nf** distinct replicas. As these **nf** messages are identical, only a single copy needs to be included in $\mathbb{C}(m_d)$ (with size $O(1)$)

together with **nf** signatures that have a constant size each, thereby reducing the size of commit certificates from $O(\mathbf{nf}^2 + \mathbf{C})$ to $O(\mathbf{nf})$.

Eliminating prepared certificates from CheckCommit messages, by replacing messages of the form CheckCommit($\mathbb{P}(m)$) by messages of the form CheckCommit(m_d), does impact the recovery mechanism of Line 8 of Figure 6 by which replicas can prepare and execute proposals: a *single* message CheckCommit(m_d) not only lacks a client request, it also lacks a prepared certificate that provides the necessary information to obtain $\langle\tau\rangle_\gamma$, to prepare m_d, and to execute $\langle\tau\rangle_\gamma$. To deal with this lack of information, a replica Q that wants to use the recovery mechanism of Line 8 of Figure 6 has to wait for identical CheckCommit(m_d) messages signed by $\mathbf{f}+1$ distinct replicas, after which Q has a guarantee that at-least one of these signing replicas is non-faulty and can be queried for the necessary prepared certificate $\mathbb{P}(m_d)$ and client request $\langle\tau\rangle_\gamma$. (For completeness, we note that this change in the number of required CheckCommit messages necessary to recover prepared certificates does not invalidate the core properties of the check-commit protocol proven in Theorem 4.4). As such, also this simplification of CheckCommit messages trades a reduction of message complexity with a more-complex recovery path.

Remark 5.2. One can strike other balances between message complexity and the complexity of the recovery path than the approach outlined above. For example, we can retain most message size benefits, while keeping the recovery path simple with a slightly different approach. In specific,

(1) To fully retain the recovery path of Line 11 of Figure 6, while also reducing the size of commit certificates to $O(\mathbf{nf})$, one can replace messages of the form CheckCommit($\mathbb{P}(m)$) by triples of the form (CheckCommit(m_d), $\mathbb{P}(m_d)$, $\langle\tau\rangle_\gamma$), in which the signed constant-sized message CheckCommit(m_d) is used in the construction of commit certificates of size $O(\mathbf{nf})$, while the prepared certificate $\mathbb{P}(m_d)$ and the client request $\langle\tau\rangle_\gamma$ can be used for the recovery path of Line 11 of Figure 6.
(2) To assure that the new primary has access to all necessary client requests, while also reducing the size of ViewState and, consequently, NewView messages, one can replace messages of the form ViewState($v, \mathbb{C}(m)$, E) by pairs of the form (ViewState($v, \mathbb{C}(m_d)$, E_d), S) in which S is the set of client requests that correspond to the commit certificate $\mathbb{C}(m_d)$ and the prepare certificates $\mathbb{P}(m'_d) \in \mathsf{E}$.
(3) To assure that replicas have knowledge of all client requests necessary to start a new view, while also reducing the size of NewView mesages, the new primary can replace messages of the form NewView($v+1$, V) by pairs of the form (NewView($v+1, \mathsf{V}_d$), $\langle\tau\rangle_\gamma$), with n_d = NewView($v+1, \mathsf{V}_d$) and $\mathcal{L}n_d[\mathsf{LC}(n_d)] = \langle\tau\rangle_\gamma$ the client request proposed by any commit certificate included in V in the last round $\mathsf{LC}(n_d)$ for which V includes commit certificates. We note that the NewView message n_d only requires this single client request to convey sufficient information: all client requests for rounds ρ, $\mathsf{LC}(n_d) < \rho \leq \mathsf{LP}(n_d)$ need to be re-proposed by the new primary, while all replicas can validate whether the new primary proposed the right requests via the digests available in n_d.

5.3 Threshold Signatures for Certificates

A further look at the usage of certificates in PoE shows that the prepared and commit certificates, which are used in several messages, will grow large in deployments with many replicas and will cause high storage costs and communication costs in such deployments. Consider, for example, the prepared and commit certificates for a Propose message m = Propose($\langle\tau\rangle_\gamma, v, \rho$). Even when using digests, both $\mathbb{P}(m)$ and $\mathbb{C}(m)$ will consists of a set of **nf** distinct (constant-sized) signatures. Hence, these certificates have a size that is linear with respect to the number of (non-faulty) replicas.

To further reduce the size of certificates, one can employ *threshold signatures* [8, 43].

Definition 5.3. A $x : y$ *threshold signature scheme* is a digital signature scheme for a set X of $|X| = x$ participants in which any set $Y \subseteq X$ of $|Y| = y$ participants can cooperate to produce a certificate for any given value v. To do so, each participant in Y produces their signature share for v. Using only valid signature shares for v produced by at-least y distinct participants, anyone can produce a valid *constant-sized* certificate for v.

In specific, each participant $p \in X$ will receive a distinct private key $k(p)$ that p can use to sign any value v, resulting in a *signature share* $\langle\!|v|\!\rangle_p$. As with traditional public-key cryptography, the signature share of p can only be produced using $k(p)$ and anyone can verify the authenticity of signature share $\langle\!|v|\!\rangle_p$ using the public key associated with $k(p)$. Given a set of signature shares $\{k(p) \mid p \in Y\}$, one can produce a single constant-sized *threshold signature* that certifies that value v was signed by at-least y distinct participants from the set X of participants in the threshold signature scheme.

In the setting of PoE, one can use a $\mathbf{n} : \mathbf{nf}$ threshold signature scheme in which each replica is a participant. Next, instead of using normal digital signatures to sign a message m', each replica R uses their private key $p(\text{R})$ to produce signature shares $\langle\!|m|\!\rangle_{\text{R}}$ for m'. These signature shares can be used to provide authenticated communication. If a replica receives **nf** signature shares for a given `Prepare` message for proposal m, then one can use these signature shares to produce a prepared certificate for m of constant size. Likewise, one can use **nf** signature shares for a given `CheckCommit` message to produce a commit certificate of constant size.

5.4 A Linear Proof-of-Execution

Although using digests (Section 5.2) or threshold certificates (Section 5.3) drastically reduces the size of individual messages, they do not change the amount of messages being sent: all replicas will broadcast `Prepare` and `CheckCommit` messages in the normal case, a *quadratic amount*. Next, we show how to further apply threshold signatures to assure that each phase of communication only uses a linear amount of messages. We note that such applications of threshold signatures are common among modern variants of Pbft such as Sbft [19], LinBFT [51], and HotStuff [52]. Hence, we shall mainly focus on the novel design necessary to make *all* normal-case communication of PoE linear, this including the decentralized check-commit protocol. Next, we detail the design of Linear-PoE.

The normal-case of PoE consists of *two* phases of quadratic communication: in the *first* phase, `Prepare` messages are exchanged, while in the *second* phase `CheckCommit` messages are exchanged. These two phases are rather different in their design: the *prepare phase* (the first phase) follows the traditional primary-backup design of Pbft and only has to succeed *when the primary is non-faulty*, whereas the *check-commit phase* (the second phase) is fully decentralized in the sense that it should succeed *independent of any non-faulty behavior* in all rounds that do not lead to primary failure.

To make the prepare phase linear, we can apply the well-known transformation from *all-to-all* communication to *all-to-one-to-all* communication using threshold signatures (e.g., [19, 22, 51, 52]). To do so, we replace the prepare phase by two subphases. In specific, upon arrival of the *first* proposal for round ρ of view v via some `Propose` message pair $(m_d = \texttt{Propose}(d, v, \rho), \langle\tau\rangle_\gamma)$, with $d = \texttt{digest}(\langle\tau\rangle_\gamma)$, each (non-faulty) replica R that received this pair will enter the *prepare phase* for m_d. As the first subphase of the prepare phase, R supports the proposal m_d by sending a reply message $\texttt{Support}(\langle\!|\texttt{Prepare}(m_d)|\!\rangle_{\text{R}})$ to the primary. As the last subphase, the primary collects well-formed messages of the form $\texttt{Support}(\langle\!|\texttt{Prepare}(m_d)|\!\rangle_{\text{Q}})$, $\text{Q} \in S, S \subset \mathfrak{R}$, from a set of $\mathbf{nf} = |S|$ replicas. Next, the primary uses the provided signature shares to produce a constant-sized prepared certificate $\mathbb{P}(m_d)$. Finally, the primary broadcasts $\mathbb{P}(m_d)$ to all replicas via a message $\texttt{Certify}(\mathbb{P}(m_d))$. After replicas receive $\texttt{Certify}(\mathbb{P}(m_d))$, they use the prepared certificate $\mathbb{P}(m_d)$

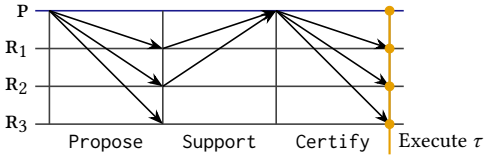

Fig. 13. A schematic representation of the *linear* normal-case protocol of PoE: the primary P proposes transaction τ to all replicas via a Propose message m. Next, all other replicas respond with Support message holding a signature share for a message of the form Prepare(m). The primary combines **nf** of these signature shares to construct a constant-sized prepared certificate $\mathbb{P}(m)$ holding a constant-sized threshold signature. Finally, the primary sends the prepared certificate $\mathbb{P}(m)$ to all replicas via a Certify message, after which replicas can execute τ. In this example, replica R_3 is faulty and does not participate.

included in Certify($\mathbb{P}(m_d)$) to prepare m and execute τ using Lines 16–18 of Figure 3. We note that the linear version of the prepare phase relies on a single replica, the primary, to aggregate Support messages into a prepared certificate. Primaries will only fail to do so if they are faulty or communication is unreliable. Hence, the normal view-change protocol can deal with failures of this linear version of the prepare phase. An illustration of the working of this linear version of the normal-case protocol can be found in Figure 13.

Next, we look at the check-commit phase. We reiterate that the check-commit phase operates rather differently than the prepare phase (and than the commit phases in most PBFT-style primary-backup consensus protocols). Indeed, the *decentralized* design of the check-commit protocol of Figure 6 is crucial for its correctness: the check-commit protocol of Figure 6 guarantees that any replicas that are left in the dark by the primary (without the primary disrupting the progress of the normal-case protocol) will be able to recover any missing proposals via the check-commit protocol (this independent of the behavior of any faulty replicas).

Consequently, we cannot simply make the check-commit protocol linear by assigning a fixed replica Q that aggregates signature shares of the form $\langle\!|\texttt{Commit}(m_d)|\!\rangle_R$ into commit certificates $\mathbb{C}(m_d)$: replica Q can be faulty and will have the power to keep individual replicas in the dark, thereby breaking a main correctness guarantee of the check-commit protocol.

To make the ckeck-commit protocol linear, we employ three techniques:

aggregator rotation. In round ρ, the replica Q with id(Q) = ρ mod **n** will be the aggregator that receives signature shares. Hence, every round has a different aggregator, assuring that **nf** out of **n** consecutive rounds have a non-faulty aggregator (when communication is reliable) and can succeed even with interference of other replicas.

aggregated multi-round check-commits. As the previous rounds can have faulty aggregators, due to which these rounds did not properly finish their check-commit steps, the aggregator Q of round ρ will perform a *multi-round* check-commit for round ρ and the preceding **n** − 1 rounds (hence, a multi-round check-commit for all rounds *since* the previous round for which Q was the aggregator). In this way, the correct behavior of Q can guarantee that it can always perform a successful check-commit when it is the aggregator. To assure that the aggregated check-commit messages for round ρ *can* have a constant size, we do not check-commit on an individual Propose message, but on the digest digest($\mathbb{P}(m_{d,1}), \ldots, \mathbb{P}(m_{d,\mathbf{n}})$) of **n** propose certificates, one for each Propose message of the last **n** rounds. (For simplicity, we assume that all replicas have agreed on default propose certificates and proposals for any rounds before the first round).

Check-commit role (running at every replica $\mathrm{R} \in \mathfrak{R}$) :

1: **event** R prepared $m = \texttt{Propose}(\langle \tau \rangle_Y, v, \rho)$ and executed τ **do**
2: Wait until all rounds up-to round $\rho - \mathbf{n}$ have a commit certificate.
3: **if** v is the current view **then**
4: Let $D = \texttt{digest}(\mathbb{P}(m_{d,1}), \ldots, \mathbb{P}(m_{d,\mathbf{n}}))$ be the digest for the prepared certificates stored in the last $\mathbf{n}$ rounds (the rounds $\rho - (\mathbf{n}-1), \ldots, \rho$).
5: Send $\texttt{SupportCC}(\rho, D, \langle |D| \rangle_{\mathrm{R}}, \langle |D| \rangle'_{\mathrm{R}})$ to the aggregator Q of round ρ (replica Q with $\mathrm{id}(\mathrm{Q}) = \rho \bmod \mathbf{n}$).
6: **end if**
7: **end event**
8: **event** R with $\mathrm{id}(\mathrm{R}) = \rho \bmod \mathbf{n}$ receives $\mathbf{f}+1$ well-formed messages $m_i = \texttt{SupportCC}(\rho, D, \langle |D| \rangle_{\mathrm{R}_i}, \langle |D| \rangle'_{\mathrm{R}_i})$, $1 \leq i \leq \mathbf{f}+1$, from $\mathbf{f}+1$ distinct replicas **do**
9: Broadcast $\texttt{RecoveryCC}(\rho, D, \mathbb{R}(D))$ to all replicas, in which $\mathbb{R}(D)$ is the *recovery certificate* constructed using the signature shares $\langle |D| \rangle'_{\mathrm{R}_i}$, $1 \leq i \leq \mathbf{f}+1$.
10: **end event**
11: **event** R receives a message $m_r = \texttt{RecoveryCC}(\rho, D, \mathbb{R}(D))$ from aggregator Q of round ρ (Q with $\mathrm{id}(\mathrm{Q}) = \rho \bmod \mathbf{n}$) **do**
12: **if** R cannot construct D with its local prepared certificates **then**
13: Query Q for the missing prepared certificates. If Q does not have these, then Q can query the $\mathbf{f}+1$ distinct replicas (of which at-least one is non-faulty) from which Q received the SupportCC messages used to construct m_r for the prepared certificates they used to construct D.
14: **end if**
15: **end event**
16: **event** R with $\mathrm{id}(\mathrm{R}) = \rho \bmod \mathbf{n}$ receives $\mathbf{nf}$ well-formed messages $m_i = \texttt{SupportCC}(\rho, D, \langle |D| \rangle_{\mathrm{R}_i}, \langle |D| \rangle'_{\mathrm{R}_i})$, $1 \leq i \leq \mathbf{nf}$, from $\mathbf{nf}$ distinct replicas **do**
17: Broadcast $\texttt{CertifyCC}(\rho, D, \mathbb{C}(\rho))$ to all replicas, in which $\mathbb{C}(\rho)$ is the *commit certificate* constructed using the signature shares $\langle |D| \rangle_{\mathrm{R}_i}$, $1 \leq i \leq \mathbf{nf}$.
18: **end event**
19: **event** R receives a message $m_c = \texttt{CertifyCC}(\rho, D, \mathbb{C}(\rho))$ from aggregator Q of round ρ (Q with $\mathrm{id}(\mathrm{Q}) = \rho \bmod \mathbf{n}$) **do**
20: Wait until all rounds up-to round $\rho - \mathbf{n}$ have a commit certificate.
21: Store *commit certificate* $\mathbb{C}(\rho)$ for round ρ, which can also be used as a commit certificate for the $\mathbf{n}-1$ rounds preceding round ρ.
22: **end event**

Fig. 14. The linear check-commit protocol in PoE.

recovery certificates. The aggregator Q of round ρ is only guaranteed to be able to construct a commit certificate for $\texttt{digest}(\mathbb{P}(m_{d,1}), \ldots, \mathbb{P}(m_{d,\mathbf{n}}))$ if *all* non-faulty replicas have a prepared certificates for the last $\mathbf{n}$ rounds. A core functionality of the check-commit protocol is to provide such prepared certificates to non-faulty replicas that are missing them (if at-least $\mathbf{f}+1$ non-faulty replicas obtained such prepared certificates, as otherwise the view-change protocol will take care of recovery). To be able to provide such recovery, the linear check-commit protocol will employ a recovery step that uses a $\mathbf{n} : \mathbf{f}+1$ threshold signature scheme to produce *recovery certificates* that can be used by non-faulty replicas to reliably obtain any missing prepared certificates. This $\mathbf{n} : \mathbf{f}+1$ threshold signature scheme is distinct from the normal $\mathbf{n} : \mathbf{nf}$ threshold signature scheme we use to generate prepared and commit certificates and we write $\langle\!| v |\!\rangle'_{\mathrm{R}}$ to denote a *signature share* produced by R using this $\mathbf{n} : \mathbf{f}+1$ threshold signature scheme.

The pseudo-code for this *linear* check-commit protocol can be found in Figure 14 and an illustration of the working of this linear version of the check-commit protocol can be found in Figure 15.

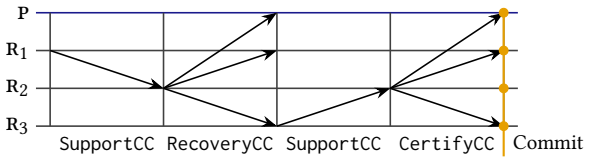

Fig. 15. A schematic representation of the *linear* check-commit protocol of PoE: in round $\rho = 6$, all replicas send a SupportCC message holding a signature share for the digest D of the last $\mathbf{n}$ proposals to the aggregator of round ρ (replica Q with $\mathrm{id}(Q) = \rho \bmod \mathbf{n}$). In this case, all replicas send to replica R_2. The aggregator Q uses the first $\mathbf{f} + 1$ of these messages to construct a recovery certificate, which can be used for replicas in the dark to recover missing rounds. Next, the aggregator Q uses $\mathbf{nf}$ of these messages to construct a *commit certificate* $\mathbb{C}(\rho)$ for rounds $\rho - (\mathbf{n} - 1), \ldots, \rho$. Finally, the aggregator Q sends the commit certificate $\mathbb{C}(\rho)$ to all replicas via a CertifyCC message, after which replicas can commit $m_{\rho-\mathbf{n}+1}, \ldots, m_\rho$. In this example, replica R_3 is left in the dark by a faulty primary and can only participate *after* receiving a recovery certificate.

Remark 5.4. We note that the failure detection stage still exchanges a quadratic number of Failure messages. As shown in Lemma 4.5, PoE depends on these all-to-all messages to *synchronize* replicas after periods of unreliable communication. We do not believe we can fully eliminate such quadratic communication when operating in an asynchronous environment in which communication has unreliable periods in which messages are *arbitrary delayed or lost* without making additional assumptions (e.g., by assuming reliable or synchronous communication, or by assuming external services to detect and deal with failures).

We complete our treatment of Linear-PoE with a caution that applies to *all* consensus protocols that use threshold signatures. First, the usage of threshold signatures will increase the latency between receiving client requests and executing client request. In PoE, this increase stems from the increase from two communication rounds to three communication rounds. In addition, producing a constant-sized prepared certificate $\mathbb{P}(m_d)$ from the signature shares provided to the primary is a step that is computationally costly. As all replicas need the prepared certificate $\mathbb{P}(m_d)$ produced by the primary to complete the consensus round, this costly step is inserted in the critical processing path of each consensus round, further increasing processing latency. Furthermore, even though the usage of threshold signatures does sharply reduces the *overall* communication complexity of consensus (the total number of messages sent between all replicas), this overall reduction does not imply a significant reduction of the communication complexity at the level of individual replicas, however, as illustrated next:

Example 5.5. Consider a deployment of PoE with $\mathbf{n} = 31$ replicas. We assume that Propose messages are larger than all other messages, which is typically the case due to the usage of digests (Section 5.2). Consequently, the primary, whom broadcasts Propose messages to all other replicas, has a much higher bandwidth usage than all other replicas.

Now consider a deployment in which Propose messages with client requests have size $\mathbf{C} = 10$ KiB and in which all other messages have a size of $\mathbf{M} = 256$ B. Without using threshold signatures, the primary will send $\mathbf{n} - 1$ Propose messages, receive $\mathbf{n} - 1$ Prepare messages, and send and receive $\mathbf{n} - 1$ CheckCommit messages per consensus decision. Hence, in this case, a single consensus decision costs $(\mathbf{n} - 1)(\mathbf{C} + 3\mathbf{M}) = 322$ KiB of bandwidth for the primary. If we switch to using threshold signatures, then the primary will still send $\mathbf{n} - 1$ Propose messages, will receive $\mathbf{n} - 1$ Support message, and send $\mathbf{n} - 1$ Certify messages. Furthermore, once every $\mathbf{n}$ rounds the primary will send and receive $3(\mathbf{n} - 1)$ messages associated with the linear check-commit protocol. Hence,

in this case, a single consensus decision still costs $(\mathbf{n}-1)(\mathbf{C}+2\mathbf{M}+\frac{3\mathbf{M}}{\mathbf{n}}) \approx 315\,\text{KiB}$ of bandwidth for the primary, a reduction of only 2%.

Finally, we notice that the linear check-commit protocol can also be used in conjunction with the non-linear normal-case of PoE, thereby eliminating the minor 2% bandwidth benefit that Linear-PoE has over the traditional design of PoE.

Due to Example 5.5, the usage of Linear-PoE over PoE in deployments in which the bandwidth at the primary is the bottleneck for performance will only yield a minor improvement in throughput, this at the cost of a roughly-$\frac{1}{3}$-th increase in latency.

5.5 MAC-based message authentication

Up till now, we have presented the design of PoE using digital signatures and threshold signatures, two powerful forms of asymmetric cryptography that provide *strong message authentication*: messages signed by some replica R using either digital signatures or threshold signatures can be safely *forwarded* by replicas, while faulty replicas are unable to forge or tamper with such signed messages. This strong form of message authentication is used throughout the design of PoE, especially within the view-change protocol that relies on message forwarding.

Unfortunately, the usage of asymmetric cryptography for message authentication comes at a high computational cost [22, 27]. As an alternative, one can consider using message authentication codes (MACs) [33], which are based on symmetric cryptography, to provide message authentication with much lower costs. Unfortunately, MACs only provide a *weaker form* of message authentication: MACs can only be used to verify the sender of messages (if the sender is non-faulty), as MACs prevent faulty replicas from impersonating non-faulty replicas. Hence, MACs do not protect messages against tampering when they are forwarded.

Still, it is well-known that Pbft-style consensus protocols can be built using MACs only [10, 22, 27]. This is also the case for PoE: when using message digests (as outlined in Section 5.2), neither the normal-case protocol, nor the check-commit protocol, nor the failure detection stage of the view-change protocol rely on message forwarding, as each of these protocols only rely on counting the number of senders of valid messages. Hence, to assure that PoE can operate using only MACs, one only needs to redesign the new-view proposal stage and new-view accept stage of the view-change protocol. Such a redesign is possible in an analogous way as the redesign of Pbft to a MAC-based version [10].

5.6 Out-of-Order Processing

All variants of PoE support out-of-order processing of consensus decisions, which is crucial for providing high consensus throughput in environments with high message delays such as wide-area (Internet) deployments.

In out-of-order consensus protocols such as PoE, the primary can freely propose requests for future rounds even if the current consensus decision is not yet finalized (as long as those future rounds lay in the current window determined by the window size). By doing so, the primary can fully utilize its outgoing bandwidth to replicate future requests, instead of waiting for other replicas to finish their consensus steps. The following example illustrates the impact of out-of-order processing:

Example 5.6. Consider a deployment with $\mathbf{n} = 31$ replicas, with a message delay of $\delta = 15\,\text{ms}$ (e.g., replicas are distributed over datacenters in a country), and in which all replicas have an outgoing bandwidth of $\mathbf{B} = 1\,\text{Gbit/s}$. In this environment, a single PoE consensus round takes at-least 3 consecutive messages (`Propose`, `Prepare`, and `CheckCommit`) and takes at-least $3\delta = 45\,\text{ms}$. Hence,

if consensus processing is not out-of-order (sequential), then this deployment of PoE will only have a throughput of less-than $\frac{1s}{3\delta} \approx 22$ consensus decisions per second.

As in Example 5.5, we consider a deployment in which Propose messages with client requests have size $\mathbf{C} = 10\,\text{kB}$ and in which all other messages have a size of $\mathbf{M} = 256\,\text{B}$. As noted in Example 5.5, the primary has the highest bandwidth usage in PoE and in this deployment, each consensus decision will cost $(\mathbf{n}-1)(\mathbf{C}+3\mathbf{M})$. Hence, if consensus processing is out-of-order, then this deployment of PoE can have a throughput of $\frac{\mathbf{B}}{(\mathbf{n}-1)(\mathbf{C}+3\mathbf{M})} \approx 3028$ consensus decisions per second, which is two orders of magnitude larger than the sequential approach.

Out-of-order processing has both a positive and negative impact on *latency*. On the one hand, as illustrated in the above example, out-of-order processing vastly improves throughput. As such, out-of-order processing will sharply reduce the average response times clients will observe for their requests when the system is processing many incoming requests. On the other hand, an out-of-order processing implementation is likely to work on multiple client requests (consensus rounds) concurrently, due to which the system latency (the duration between the primary proposing a request and replicas executing the request) is likely to be slightly higher.

6 Practical Evaluation

This section presents an evaluation of the performance of PoE and Linear-PoE when compared with other state-of-the-art consensus protocols. In specific, we compare the performance of PoE and Linear-PoE with Pbft, Zyzzyva, and HotStuff (each detailed in Section **??**). We implement these protocols on Apache ResilientDB [1].

6.1 Evaluation Setup

We set up our experiments on *AWS c5.4xlarge* nodes that each consist of a 16-core vCPU and 32 GiB of main memory. During our experiments, we varied the number of replicas from 16 to 64. To ensure sufficient incoming request to saturate the throughput of the system, we deployed 10 clients that constantly send transactions to the primary.

Each client produces a workload of 600 000 transactions (600 ktxn), each transaction in this workload containing a key-value pair with a 10 B payload. Clients send their transactions in batches to the primary and by default a batch of transactions consists of 100 txn. In both PoE and Linear-PoE, we have set the the period of the check-commit protocol used as once per 1000 proposals.

We run each experiment for 180s: the first 60s are warmup, and measurement results are collected over the next 120s. We average our results over three runs. We measure throughput as transactions executed per second. We measure latency as the time from when the client sends a request to the time when the client receives a response.

6.2 Experiments and their Results

We performed three experiments to study the performance of PoE and verify whether the *practical performance* of PoE aligns with the performance analysis of Section **??**.

6.2.1 Performance (fault-free). In the first experiment, we measure the performance (both throughput and latency) of the consensus protocols as a function of the number of replicas under the assumption all replicas are non-faulty. The results of this experiment can be found in Figure 16.

Both PoE and Pbft perform better than existing protocols by providing higher throughput and lower latency. PoE can handle about 1800 ktxn/s with 16 replicas, while Pbft manages around 1400 ktxn/s. With 64 replicas, PoE achieves double the throughput of PBFT, with rates of 720 ktxn/s and 480 ktxn/s, respectively.

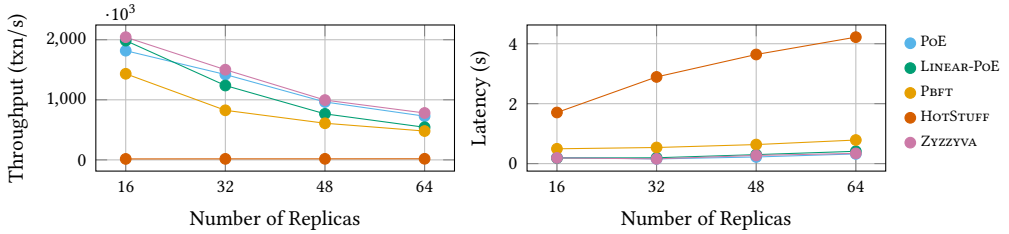

Fig. 16. Throughput and latency of PoE and Linear-PoE and other state-of-the-art consensus protocols as a function of the number of replicas. In this experiment, all replicas are non-faulty.

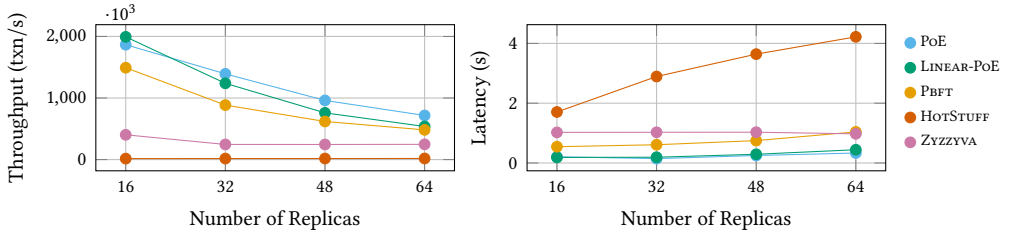

Fig. 17. Throughput and latency of PoE and Linear-PoE and other state-of-the-art consensus protocols as a function of the number of replicas. In this experiment, a single backup replica is faulty.

PoE's advantage comes from its two-phase commit and regular check commits. Zyzzyva also shows high throughput by allowing quick transaction commitments. The difference in throughput between PoE and Zyzzyva is small. PoE's throughput is 10% lower than Zyzzyva with 16 replicas and only 2% lower with 64 replicas.

In contrast, Linear-PoE achieves a throughput of 2000 ktxn/s with 16 replicas but only 540 ktxn/s with 64 replicas. Linear-PoE outperforms PoE with 16 replicas due to its linear message complexity. However, at higher replica counts we see a lower performance due to bottlenecks associated with larger payload.

6.2.2 Performance (a single backup failure). In the second experiment, we measure the performance of the consensus protocols under the assumption there is one faulty replica. In this experiment, the faulty replica is *not* the primary and the results of this experiment can be found in Figure 17. Note that Zyzzyva has to wait for messages from all n replicas. As predicting an optimal value for timeouts is hard [11, 12], we chose a very small value for the timeout (1 s) for replicas and clients.

Figure 17 compares the performance of various protocols with 1 faulty replica and does not propose any transactions. During the experiments, we intentionally forced one backup replica to stop working. In fault-free conditions, PoE, Linear-PoE, and Pbft achieved similar throughputs of 1800 ktxn/s, 2000 ktxn/s, and 1500 ktxn/s, respectively, with 16 replicas.

When compared to the fault-free experiments, we see a minor increase of latency by approximately 0.1× to 0.2× in PoE and Pbft. In Linear-PoE we see no noticeable increase, however, since all replicas, except the faulty one, must wait for the primary replica.

In the case of Zyzzyva, clients experience timeouts while waiting, which causes them to stop sending requests. This empties the pipeline at the primary, preventing it from receiving new requests to propose. Consequently, Zyzzyva maintains a stable throughputs of only 200 ktxn/s and a latency of 1 s, even as the number of replicas increases.

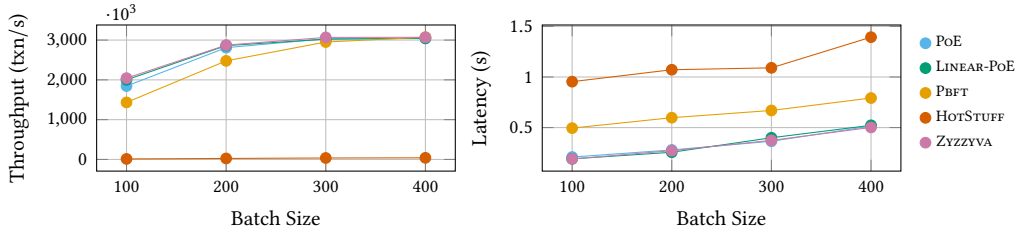

Fig. 18. Throughput and latency of PoE and Linear-PoE and other state-of-the-art consensus protocols as a function of the batch size. In this experiment, we use 16 replicas (none of which are faulty).

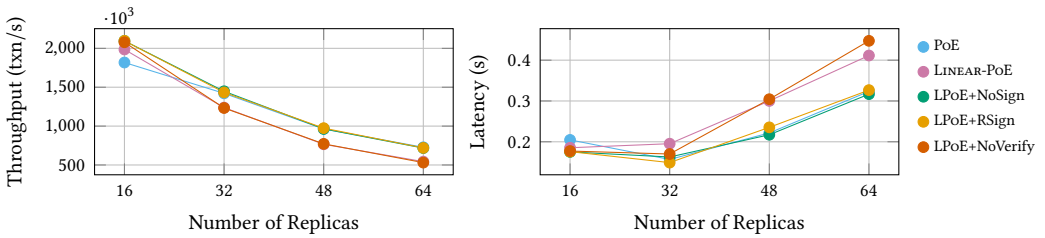

Fig. 19. Throughput and latency of different components. In the legend, LPoE is shorthand for Linear-PoE.

6.2.3 Batch Size. In the third experiment, we measure the performance (both throughput and performance) of the consensus protocols as a function of the batch size in a system with 16 replicas and under the assumption that all replicas are non-faulty. We vary the batch size between 100 and 400 and the results of this experiment can be found in Figure 18.

For each protocol, increasing the batch size boosts throughput while reducing latency. This occurs because larger batch sizes require fewer consensus rounds to complete the same set of requests, which lowers the cost of ordering and executing transactions. As a result, throughput improves, and clients benefit from faster response times for their requests. However, while increasing the batch size decreases the number of consensus rounds, the larger message size leads to a proportional decline in throughput or an increase in latency. This is evident in experiments conducted at higher batch sizes; for instance, increasing the batch size beyond 200 gradually flattens the throughput plots as they approach a limit. Specifically, when the batch size is increased from 200 to 400, protocols such as PoE, Linear-PoE, Pbft, and Zyzzyva all demonstrate increased throughput. Notably, these protocols nearly converge in their throughput when the batch size reaches 400. This indicates that the bottleneck in the protocol is shifting from communication among replicas to primary execution. In the case of HotStuff, an increase in batch size does enhance its throughput, but due to the high scaling of the graph, this change appears relatively insignificant.

6.2.4 Impact of using threshold signatures in Linear-PoE. In the previous experiments, we have already illustrated that the performance of Linear-PoE and PoE are close to each other. This is to be expected: although Linear-PoE sharply reduces communication for the backups, it does not do so for the primary. Furthermore, as noted in Section 5.3, Linear-PoE adds costly computations on the critical processing path of consensus round. Next, we will investigate the impact of these costly computations on the performance of Linear-PoE. We do so by analyzing each component involved in handling certificates based on threshold signatures in Linear-PoE. In specific, we analyze the performance of *four* Linear-PoE variants with increasing computational costs:

- Linear-PoE*+NoSign*. This configuration completely *eliminates all cryptographic components* of using threshold signatures in Linear-PoE: backups *do not* sign their responses, the primary does not verify signatures or produce certificates, and backups do not verify certificates.
- Linear-PoE*+RSign*. In this configuration, backups sign their responses, the primary *does not* verify these signatures or produce certificates, and backups do not verify certificates.
- Linear-PoE*+NoVerify*: In this configuration, backups sign their responses, the primary verifies these signatures and produces certificates, and backups *do not* verify these certificates.
- Linear-PoE: This configuration *includes all cryptograhpic components* of using threshold signatures in Linear-PoE: backups sign their responses, the primary verifies these signatures and produces certificates, and backups verify certificates.

This structured approach allows for a comprehensive understanding of the impact of the added computational costs of using threshold signatures in Linear-PoE.

Note that each of the above Linear-PoE variants only differ in the *computational cost* of dealing with threshold signatures and the minimal communication costs of including signatures, whereas all other parameters are kept the same.

For each of the four variants, we measure the performance (both throughput and latency) of the consensus protocol as a function of the number of replicas under the assumption all replicas are non-faulty. The results of this experiment can be found in Figure 19. As is clear from Figure 19, adding costly computations to the critical processing path of a consensus round has a small impact on performance and latency. In specific, we see:

(1) As the results for Linear-PoE*+NoSign* and Linear-PoE*+RSign* indicate, the impact of signing responses by backups and including signatures is negligible. This is to be expected: signatures are small and the computational step of signing responses is done in parallel by all backups. Furthermore, the cost of signing responses is low, as it is comparable in cost to a normal (non-threshold) signature-based signing step.
(2) As the results for Linear-PoE*+NoVerify* indicate, the impact of producing certificates by the primary is small when only a few replicas are involved, but clearly grows when the number of replicas increases. This is an unfortunate consequence of the current state-of-the-art threshold signature implementations: to the best of our knowledge, the algorithms used to produce certificates in all have complexities that are super-linear in the number of signature shares (and, hence, in Linear-PoE super-linear in the number of replicas).
(3) As the results for Linear-PoE indicate, the impact of verifying certificates is negligible. Again, this is to be expected: this step is done in parallel by all backups and is a traditional (non-threshold) signature verification step.

6.3 Discussion

The results of our practical evaluation aligns with the analytical analysis of Section **??**. In specific, when comparing with Pbft, Zyzzyva, and HotStuff, we see that

(1) PoE indeed outperforms Pbft in all situations. Furthermore, also Linear-PoE outperforms Pbft.
(2) PoE outperforms Zyzzyva in all but the most-optimistic case without failures. The usage of larger batches can minimize the higher all-case communication cost of PoE in the most-optimistic case and ensure PoE has comparable performance to Zyzzyva even in the most-optimistic case.
(3) PoE, Linear-PoE, and Pbft outperform HotStuff by orders-of-magnitude due to the usage of high-performance out-of-order implementations of PoE, Linear-PoE, and Pbft.

When comparing PoE with Linear-PoE, the results are more nuanced: as expected, both protocols have great performance. Still, we see that PoE slightly outperforms Linear-PoE in our experiments, this despite the quadratic message cost of PoE (versus the linear cost of Linear-PoE). As illustrated in Example 5.5, this is to be expected: when both protocols use the same check-commit protocols, then both protocols put the same bandwidth costs on the primary. At the same time, Linear-PoE puts higher computational costs on the primary.

In general, we would advise the usage of PoE over Linear-PoE due to its slightly better performance. Still, there are plenty of cases where Linear-PoE is a good choice. First, Linear-PoE drastically reduces overall network usage and, hence, is the best choice if network usage comes at a premium. Furthermore, Linear-PoE is more suitable for non-homogeneous deployments (e.g., the Internet-of-Things [35, 50]), in which only a few replicas are very powerful and are most-likely to fulfill the primary role: due to the sharp decrease in the number of messages backups have to process in Linear-PoE, Linear-PoE sharply reduces both the communication costs and computational costs for all backups involved.

We notice that the results of Section 6.2.4 also provide modern insight in the limited benefits of using *message authentication codes* (MACs) instead of digital signatures: as argued in Section 5.5, using MACs reduces the computational complexity of message handling. In the past, using MACs had clear benefits due to the limited compute power available to replicas [10]. Modern deployments on many-core hardware can utilize the abundance of parallel compute power to sign messages and verify signatures, thereby minimizing the benefits of MACs. Our experiments verify the low practical cost of using digital signatures on modern hardware: the costs of signing messages (the difference between Linear-PoE*+NoSign* and Linear-PoE*+RSign*) and the costs of verifying signatures (the difference between Linear-PoE*+NoVerify* and Linear-PoE have only a minimal impact on performance. Based on these results, we can only recommend the usage of MACs in deployments that combine very high network bandwidths with limited replicas with very low-performance hardware.

7 Conclusion

In this paper, we introduced PoE, a consensus protocol designed for high-performance low-latency resilient data management systems and that can operate in practical asynchronous deployments. PoE introduces the usage of speculative execution and *proof-of-execution* to minimize the latency of transaction processing in such resilient systems and introduces a single-round check-commit protocol to further reduce communication costs of consensus. Furthermore, the flexible design of PoE allows for optimizations that further balance communication costs, transaction latency, and recovery complexity.

The flexible low-latency high-throughput design of PoE is especially suited for resilient data management systems. To illustrate this, we performed an in-depth analytical and experimental comparison with other consensus protocols, that underlined the outstanding performance of PoE. Hence, we believe that PoE is a promising step towards flexible general-purpose resilient data management systems.

References

[1] Apache ResilientDB. URL: https://resilientdb.incubator.apache.org/.

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance, 2017. URL: https://arxiv.org/abs/1712.01367.

[3] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma, 2018. URL: https://arxiv.org/abs/1801.10022.

[4] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. CAPER: A cross-application permissioned blockchain. *Proc. VLDB Endow.*, 12(11):1385–1398, 2019. doi:10.14778/3342263.3342275.

[5] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*, pages 76–88. ACM, 2021. doi:10.1145/3448016.3452807.

[6] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The bedrock of byzantine fault tolerance: a unified platform for bft protocols analysis, implementation, and experimentation. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2024.

[7] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4), 2015. doi:10.1145/2658994.

[8] Yuliang Baek, Joonsang Zheng. Simple and efficient threshold cryptosystem from the gap diffie-hellman group. In *GLOBECOM '03. IEEE Global Telecommunications Conference*, volume 3, pages 1491–1495. IEEE, 2003. doi:10.1109/GLOCOM.2003.1258486.

[9] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985. doi:10.1145/4221.214134.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002. doi:10.1145/571637.571640.

[11] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 277–290. ACM, 2009. doi:10.1145/1629575.1629602.

[12] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 153–168. USENIX, 2009.

[13] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50. ACM, 2022. doi:10.1145/3492321.3519594.

[14] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, pages 123–140. ACM, 2019. doi:10.1145/3299869.3319889.

[15] Alex de Vries. Bitcoin's growing energy problem. *Joule*, 2(5):801–805, 2018. doi:10.1016/j.joule.2018.04.016.

[16] Danny Dolev. The byzantine generals strike again. *J. Algorithm*, 3(1):14–30, 1982. doi:10.1016/0196-6774(82)90004-9.

[17] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A shared database on blockchains. *Proc. VLDB Endow.*, 12(11):1597–1609, 2019. doi:10.14778/3342263.3342636.

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

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

[20] William J. Gordon and Christian Catalini. Blockchain technology for healthcare: Facilitating the transition to patient-driven interoperability. *Comput. Struct. Biotechnol. J.*, 16:224–230, 2018. doi:10.1016/j.csbj.2018.06.003.

[21] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978. doi:10.1007/3-540-08755-9_9.

[22] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. *Fault-Tolerant Distributed Transactions on Blockchain*. Synthesis Lectures on Data Management. Morgan & Claypool, 2021. doi:10.2200/S01068ED1V01Y202012DTM065.

[23] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. RCC: Resilient concurrent consensus for high-throughput secure transaction processing. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1392–1403. IEEE, 2021. doi:10.1109/ICDE51399.2021.00124.

[24] Suyash Gupta, Dakai Kang, Dahlia Malkhi, and Mohammad Sadoghi. Brief announcement: Carry the tail in consensus protocols. In *39th International Symposium on Distributed Computing (DISC 2025)*, 2025.

[25] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *Proc. VLDB Endow.*, 13(6):868–883, 2020. doi:10.14778/3380750.3380757.

[26] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. Proof-of-Execution: Reaching consensus through fault-tolerant speculation. In *Proceedings of the 24th International Conference on Extending Database Technology (EDBT)*, pages 301–312. OpenProceedings.org, 2021. doi:10.5441/002/edbt.2021.27.

[27] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In *2020 IEEE 40th International Conference on Distributed Computing*

Systems (ICDCS), pages 754–764. IEEE, 2020. doi:10.1109/ICDCS47774.2020.00012.

[28] Jelle Hellings and Mohammad Sadoghi. Byshard: Sharding in a byzantine environment. *Proc. VLDB Endow.*, 14(11):2230–2243, 2021. doi:10.14778/3476249.3476275.

[29] Maged N. Kamel Boulos, James T. Wilson, and Kevin A. Clauson. Geospatial blockchain: promises, challenges, and scenarios in health and healthcare. *Int. J. Health. Geogr*, 17(1):1211–1220, 2018. doi:10.1186/s12942-018-0144-x.

[30] Andreas Kamilaris, Agusti Fonts, and Francesc X. Prenafeta-Boldύ. The rise of blockchain technology in agriculture and food supply chains. *Trends in Food Science & Technology*, 91:640–652, 2019. doi:10.1016/j.tifs.2019.07.034.

[31] Dakai Kang, Suyash Gupta, Dahlia Malkhi, and Mohammad Sadoghi. HotStuff-1: Linear consensus with one-phase speculation. *Proc. ACM Manag. Data*, 3(3), June 2025. doi:10.1145/3725308.

[32] Dakai Kang, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. SpotLess: Concurrent rotational consensus made practical through rapid view synchronization. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 1916–1929, 2024. doi:10.1109/ICDE60146.2024.00157.

[33] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2nd edition, 2014.

[34] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4), 2010. doi:10.1145/1658357.1658358.

[35] Laphou Lao, Zecheng Li, Songlin Hou, Bin Xiao, Songtao Guo, and Yuanyuan Yang. A survey of iot applications in blockchain systems: Architecture, consensus, and traffic modeling. *ACM Comput. Surv.*, 53(1), 2020. doi:10.1145/3372136.

[36] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.*, 3(3):202–215, 2006. doi:10.1109/TDSC.2006.35.

[37] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. URL: https://bitcoin.org/en/bitcoin-paper.

[38] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database. *Proc. VLDB Endow.*, 12(11):1539–1552, 2019. doi:10.14778/3342263.3342632.

[39] Faisal Nawab and Mohammad Sadoghi. Blockplane: A global-scale byzantizing middleware. In *35th International Conference on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019. doi:10.1109/ICDE.2019.00020.

[40] Michael Pisa and Matt Juden. Blockchain and economic development: Hype vs. reality. Technical report, Center for Global Development, 2017. URL: https://www.cgdev.org/publication/blockchain-and-economic-development-hype-vs-reality.

[41] Abderahman Rejeb, John G. Keogh, Suhaiza Zailani, Horst Treiblmaier, and Karim Rejeb. Blockchain technology in the food industry: A review of potentials, challenges and future research directions. *Logistics*, 4(4), 2020. doi:10.3390/logistics4040027.

[42] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, Gang Chen, Qian Lin, and Beng Chin Ooi. Blockchains vs. distributed databases: Dichotomy and fusion. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1504–1517. ACM, 2021. doi:10.1145/3448016.3452789.

[43] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology — EUROCRYPT 2000*, pages 207–220. Springer, 2000. doi:10.1007/3-540-45539-6_15.

[44] Dale Skeen. A quorum-based commit protocol. Technical report, Cornell University, 1982.

[45] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2001.

[46] The Hyperledger White Paper Working Group. An introduction to Hyperledger. Technical report, The Linux Foundation, 2018.

[47] Horst Treiblmaier and Roman Beck, editors. *Business Transformation through Blockchain*. Springer, 2019. doi:10.1007/978-3-319-98911-2.

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

[49] Harald Vranken. Sustainability of bitcoin and blockchains. *Current Opinion in Environmental Sustainability*, 28:1–9, 2017. doi:10.1016/j.cosust.2017.04.011.

[50] Mingli Wu, Kun Wang, Xiaoqin Cai, Song Guo, Minyi Guo, and Chunming Rong. A comprehensive survey of blockchain: From theory to IoT applications and beyond. *IEEE Internet Things J*, 6(5):8114–8154, 2019. doi:10.1109/JIOT.2019.2922538.

[51] Yin Yang. LinBFT: Linear-communication byzantine fault tolerance for public blockchains, 2018. URL: https://arxiv.org/abs/1807.01829.

[52] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019. doi:10.1145/3293611.3331591.

A Analytical Evaluation

In Section 5, we analyzed in-depth the cost of consensus using the PoE consensus protocol and its variants. Next, we compare these costs with the costs of existing consensus protocols. As there are many recently-developed Pbft-style consensus protocols, we will limit ourselves to a representative sample of frequently-used technologies and representative protocols using these technologies. A summary of our comparison can be found in Figure 1.

A.1 The Baseline of Comparison: Pbft

The *Practical Byzantine Fault Tolerance* consensus protocol [10] was introduced two decades ago and to this day is a baseline for providing high-performance consensus in practical environments. Pbft is highly resilient and can even deal with network failures: although network failures can temporarily disrupt *new* consensus decisions in Pbft, Pbft is able to automatically recover its operations once the network becomes reliable, this without ever loosing any previously-made consensus decisions. Furthermore, recent works have shown that highly-optimized and fine-tuned implementations of this protocol can achieve throughputs surpassing more modern protocols in moderately-sized deployments [22, 27].

We have already provided a high-level description of the working of Pbft in Example 3.1. As mentioned in Section 4, PoE shares the primary-backup design of Pbft and, as proven in Section 4.5, PoE shares the high resilience of Pbft. The main differences between Pbft and PoE can be summarized as follows:

(1) PoE utilizes *speculative execution*, due to which replicas can execute transactions and inform clients directly after the *prepare phase*, whereas Pbft only executes client requests after their *commit phase*. Due to this, PoE can inform clients within only *two communication rounds*, while Pbft requires three communication rounds before it can inform clients.
(2) PoE utilizes the *check-commit protocol*, a decentralized single phase protocol that servers the same roles as the *commit phase* of Pbft, which requires a round of communication, and the checkpoint protocol of Pbft, which requires another round of communication. Due to this, the normal case of PoE only takes three rounds of communication (of which only two are all-to-all), whereas the normal case of Pbft requires four rounds of communication (of which three are all-to-all).

The view-change protocols of PoE and Pbft are comparable in communication costs: the main difference between these view-change protocols is that view-changes in PoE need to account for speculative execution (e.g., perform rollbacks), but this accounting is done while determining the state represented by new view proposals and does not impose additional communication costs.

Due to the above analysis, we can conclude that PoE will *outperform* Pbft in all situations, as PoE lowers communication costs *in all cases* (by eliminating one round of all-to-all communication), while also potentially reducing client latencies due to speculative execution.

A.2 Optimistic Protocols: Zyzzyva

Several consensus protocols have attempted to reduce the communication cost of Pbft via *optimistic consensus* [7, 34, 36]. In an optimistic consensus protocol, the normal-case consensus protocol consists of a *fast path* that will succeed under optimal conditions (no faulty behavior or unreliable communication) and a *slow path* to deal with non-optimal conditions.

An example of such optimistic consensus is Zyzzyva [34]: in Zyzzyva, all replicas execute directly after they receive a proposal of the primary, directly inform the client, and proceed with the next round of consensus. Hence, in Zyzzyva the optimal-case cost of consensus are minimal: only one round of primary-to-backup communication. We note that this *fast path* is not able to detect failures

between replicas: Zyzzyva requires clients to inform replicas of any failures, after which replicas can enter a *slow path* to deal with these failures.

Unfortunately, consensus protocols with *optimistic* fast paths such as Zyzzyva [34] and FaB [36] have over time shown vulnerabilities to faulty behavior, this especially in the presence of unreliable communication [2, 3]. In specific, the combination of faulty primaries and unreliable communication can set up a sequence of view-changes in these protocols that break non-divergence.

We note that speculative execution, as used by PoE, and optimistic execution, e.g., as used by Zyzzyva [34] and FaB [36], are not the same: as part of the normal-case of PoE, PoE will internally detect and correct any replica failures, this without any assumptions on correct behavior by any replicas or clients.

Remark A.1. On a first look, the reduction in message complexity provided by the most-optimistic fast path of Zyzzyva is highly appealing. In practice, the impact of this reduction on system throughput is often limited, however.

First, as underlined in Example 5.6, the bandwidth available at the primary is the main bottleneck that determines system throughput (when using out-of-order processing) [22]. As the `Propose` messages broadcasted by the primary are typically much larger than the control messages exchanged to verify whether the primary properly proposed a decision, the elimination of these control messages in the fast path of Zyzzyva only has a limited impact on bandwidth usage at the primary and, hence, on throughput.

Second, a standard technique to minimize the overhead of consensus is by using *batching*: instead of proposing a single client request per consensus round, batching allows the primary to propose a large group of client requests, thereby sharing the overheads of a single consensus round with all requests in the batch. By choosing the batch size sufficiently large, one can practically mitigate the higher communication complexity of the normal-case path of PoE (that can deal with failures), even when comparing with the fast path of Zyzzyva (which cannot deal with failures).

A.3 Consensus with Threshold Signatures: Sbft

Several recent Pbft-style consensus protocols [19, 51, 52] have explored the usage of *threshold signatures* to transform the two phases of all-to-all communication in Pbft (the *prepare* and *commit* phases) to two phases of all-to-one-to-all communication. These transformations are similar to how we can transform the prepare phase of PoE from all-to-all communication to an all-to-one support sub-phase and an one-to-all certify sub-phase. We note that such a transformation, when applied on the two all-to-all communication phases of Pbft, will result in a consensus protocol with a high latency: in such a protocol, it will take five communication rounds before replicas can execute client requests and inform clients. Although such a transformation can successfully reduce the global communication cost of the normal-case operations of the consensus protocol, such threshold signature transformations do little to address the costs associated with any checkpoint and view-change protocols.

A good examples of a Pbft-style consensus protocols that use threshold signatures is Sbft [19], which uses an optimistic fast path to reduce the number of rounds when all replicas are non-faulty, uses threshold signatures to eliminate all-to-all communication, and uses threshold signatures to reduce the number of messages send to the client. The design of Sbft uses a fast path which starts execution of client requests after the prepare phase *if* no replicas are faulty. Furthermore, Sbft can aggregate the a proof of the execution results and send such proof in a single message to the client (instead of $\mathbf{f} + 1$ messages), this to reduce communication costs towards the client. With this optimization, Sbft is able to inform clients in four rounds of communication (which is one more round than Pbft and Linear-PoE and two more rounds than PoE). In case the fast path fails,

Sbft falls back to a *slow path* via a Linear-Pbft implementation. To reduce the load on the primary (see Example 5.5), Sbft uses non-primary replicas as the aggregator that constructs prepared and commit certificates during the prepare and commit phases. The usages of non-primary aggregators at these parts of the protocol does introduce additional failure cases, however, for which Sbft introduces separate recovery mechanisms. Similar fine-tuning can also be applied to PoE, but we have not explored such fine-tuning in this work (as separate aggregators only introduce minimal bandwidth savings for the primary). Besides the fast path and the slow path, Sbft also requires a checkpoint protocol similar to the one utilized by Pbft (which can be run periodically).

A.4 Chained consensus: HotStuff

Another approach toward utilizing threshold signatures in primary-backup consensus protocols is provided by HotStuff. HotStuff provides a clean-slate consensus design that is tuned toward minimizing complexity and communication cost, both during normal-case operations and during view-changes. To achieve this, HotStuff relies on *chaining consensus*: the i-th consensus proposal builds upon the preceding $(i-1)$-th consensus proposal. This allows HotStuff to represent the state of the ledger via a single value, namely the last-made consensus decision (that builds upon all preceding decisions). Finally, HotStuff uses threshold signatures to produce constant-size certificates for each consensus decision. This combination of techniques allows HotStuff to implement cheap *primary rotation*: each round starts with a switch of primary via a constant-sized single-message view-change.

The design of HotStuff requires 4 consecutive all-to-one-to-all phases of communication before consensus is reached on a single request, which leads to 7 rounds of communication between the initial proposal of a client request and replicas being able to execute these requests. Both the normal-case and the view-change of HotStuff are *linear*. To deal with unresponsive replicas and replica failures, HotStuff uses a Pacemaker. Unfortunately, the standard Pacemaker of HotStuff assumes *partial synchrony* and cannot recover from recurring network failures. To improve the resilience against network failure in HotStuff, one can replace the standard Pacemaker that HotStuff uses with a Pacemaker suitable for an asynchronous environment. As stated in Remark 5.4, we believe that any asynchronous Pacemaker that can sufficiently synchronize replicas after network failure will have to operate similarly to the *failure detection stage* of PoE, which would introduce a decentralized all-to-all communication phase in the recovery path of HotStuff.

Due to the chained design of HotStuff, HotStuff does not support out-of-order processing: consensus decisions are strictly made in sequence. HotStuff does support overlapping of rounds of consecutive consensus decisions, however. Hence, in practice, HotStuff is able to propose a request every 2 rounds of communication. Consequently, the performance of typical deployments of HotStuff are *latency based* and non-local deployments can only reach tens-to-hundreds consensus decisions per second. For example, with a message delay of 15 ms HotStuff can perform at-most 33 consensus decisions per second, whereas, as shown in Example 5.6, an out-of-order PoE can easily process thousands of consensus decisions per second.

A.5 Trusted Hardware

There is a large body of work on consensus protocols that utilize *trusted hardware* to simplify and optimize consensus. The usage of trusted hardware restricts the behavior allowed by faulty or malicious replicas, e.g., by assuring faulty replicas cannot share their private keys and cannot forge round numbers and other counters. A representative example of a consensus protocol that utilizes trusted hardware is MinBFT [48]. Due to the usage of trusted hardware, MinBFT can skip the prepare phase of Pbft and can achieve consensus in only two rounds of communication. Furthermore, the usage of trusted hardware makes MinBFT more resilient against failure, as it

requires **f** fewer non-faulty replicas than normal consensus protocols (MinBFT can operate even in deployments in which $3\mathbf{f} \geq \mathbf{n} > 2\mathbf{f}$ holds). Unfortunately, the presented version of MinBFT does require *reliable communication*, as the protocol itself does not deal with message loss.

To the best of our knowledge, MinBFT and other consensus protocols that utilize trusted hardware are the only consensus protocols *besides* PoE that have a latency of two rounds of communication (without relying on a fault-prone optimistic path).