Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation

Suyash Gupta Jelle Hellings Sajjad Rahnama Mohammad Sadoghi
Exploratory Systems Lab
Department of Computer Science
University of California, Davis

ABSTRACT

Multi-party data management and blockchain systems require data sharing among participants. To provide resilient and consistent data sharing, transactions engines rely on Byzantine Fault-Tolerant consensus (BFT), which enables operations during failures and malicious behavior. Unfortunately, existing BFT protocols are unsuitable for high-throughput applications due to their high computational costs, high communication costs, high client latencies, and/or reliance on twin-paths and non-faulty clients.

In this paper, we present the *Proof-of-Execution consensus protocol* (PoE) that alleviates these challenges. At the core of PoE are *out-of-order* processing and *speculative execution*, which allow PoE to execute transactions before consensus is reached among the replicas. With these techniques, PoE manages to reduce the costs of BFT in normal cases, while guaranteeing reliable consensus for clients in all cases. We envision the use of PoE in high-throughput multi-party data-management and blockchain systems. To validate this vision, we implement PoE in our efficient ResilientDB fabric and extensively evaluate PoE against several state-of-the-art BFT protocols. Our evaluation showcases that PoE achieves up-to-80% higher throughputs than existing BFT protocols in the presence of failures.

1 INTRODUCTION

In *federate data management* a single common database is managed by many independent stakeholders (e.g., an industry consortium). In doing so, federated data management can ease data sharing and improve data quality [17, 32, 48]. At the core of federated data management is *reaching agreement* on any updates on the common database in an efficient manner, this to enable fast query processing, data retrieval, and data modifications.

One can achieve federated data management by *replicating* the common database among all participant, this by replicating the sequence of transactions that affect the database to all stakeholders. One can do so using commit protocols designed for distributed databases such as two-phase [22] and three-phase commit [49], or by using crash-resilient replication protocols such as Paxos [39] and Raft [45].

These solutions are error-prone in a federated *decentralized* environment in which each stakeholder manages its own replicas and replicas of each stakeholder can fail (e.g., due to software, hardware, or network failure) or act malicious: commit protocols and replication protocols can only deal with crashes. Consequently, recent federated designs propose the usage of Byzantine Fault-Tolerant (BFT) consensus protocols. BFT consensus aims at *ordering client requests among a set of replicas, some of which could be Byzantine, such that all non-faulty replicas reach agreement on a common order for these requests* [9, 21, 29, 38, 51]. Furthermore, BFT consensus comes with the added benefit of *democracy*, as BFT consensus gives all replicas an equal vote in all agreement decisions, while the resilience of BFT can aid in dealing with the billions of dollars losses associated with prevalent attacks on data management systems [44].

Akin to commit protocols, the majority of BFT consensus protocols use a *primary-backup model* in which one replica is designated *the primary* that coordinates agreement, while the remaining replicas act as backups and follow the protocol [46]. This primary-backup BFT consensus was first popularized by the influential PBFT consensus protocol of Castro and Liskov [9]. The design of PBFT requires at least $3\mathbf{f}+1$ replicas to deal with up-to-$\mathbf{f}$ malicious replicas and operates in *three communication phases*, two of which necessitate quadratic communication complexity. As such, PBFT is considered costly when compared to commit or replication protocols, which has negatively impacted the usage of BFT consensus in large-scale data management systems [8].

The recent interest in blockchain technology has revived interest in BFT consensus, has led to several new resilient data management systems (e.g., [3, 18, 29, 43]), and has led to the development of new BFT consensus protocols that promise efficiency at the cost of flexibility (e.g., [21, 28, 38, 51]). Despite the existence of these modern BFT consensus protocols, the majority of BFT-fueled systems [3, 18, 29, 43] still employ the classical time-tested, flexible, and safe design of PBFT, however.

In this paper, we explore different design principles that can enable implementing a scalable and reliable agreement protocol that shields against malicious attacks. We use these design principles to introduce Proof-of-Execution (PoE), a novel BFT protocol that achieves resilient agreement in just three linear phases. To concoct PoE's scalable and resilient design, we start with PBFT and successively add *four* design elements:

(I1) **Non-Divergent Speculative Execution.** In PBFT, when the primary replica receives a client request, it forwards that request to the backups. Each backup on receiving a request from the primary agrees to support by broadcasting a PREPARE message. When a replica receives PREPARE message from the majority of other replicas, it marks itself as *prepared* and broadcasts a COMMIT message. Each replica that has prepared, and receives COMMIT messages from a majority of other replicas, executes the request.

Evidently, PBFT requires two phases of *all-to-all* communication. Our first ingredient towards faster consensus is speculative execution. In PBFT terminology, PoE replicas execute requests after they get *prepared*, that is, they do not broadcast COMMIT messages. This speculative execution is non-divergent as each replica has a partial guarantee–it has prepared–prior to execution.

(I2) **Safe Rollbacks and Robustness under Failures.** Due to speculative execution, a malicious primary in PoE can ensure that only a subset of replicas prepare and execute a request. Hence, a client may or may not receive a sufficient number of matching responses. PoE ensures that if a client receives a *full proof-of-execution*, consisting of responses from a majority of the non-faulty replicas, then such a request persists in time. Otherwise, PoE permits replicas to *rollback* their state if necessary. This proof-of-execution is the cornerstone of the correctness of PoE.

(I3) **Agnostic Signatures and Linear Communication.** BFT protocols are run among distrusting parties. To provide security, these protocols employ cryptographic primitives for signing the messages and generating message digests. Prior works have shown that the choice of cryptographic signature scheme can impact the performance of the underlying system [9, 30]. Hence, we allow replicas to either employ *message authentication codes* (MACs) or *threshold signatures* (TSs) for signing [36]. When few replicas are participating in consensus (up to 16), then a single phase of all-to-all communication is inexpensive and using MACs for such setups can make computations cheap. For larger setups, we employ TSs to achieve linear communication complexity. TSs permit us to split a phase of all-to-all communication into *two linear phases* [21, 51].

(I4) **Avoid Response Aggregation.** SBFT [21], a recently-proposed BFT protocol, suggests the use of a single replica (designated as the *executor*) to act as a response aggregator. In specific, all replicas execute each client request and send their response to the executor. It is the duty of the executor to reply to the client and *send a proof* that a majority of the replicas not only executed this request, but also outputted the same result. In PoE, we avoid this additional communication between the replicas by allowing each replica to respond directly to the client.

In specific, we make the following contributions:

(1) We introduce PoE, a novel Byzantine fault-tolerant consensus protocol that uses *speculative execution* to reach agreement among replicas.
(2) To guarantee failure recovery in the presence of speculative execution and Byzantine behavior, we introduce a novel view-change protocol that can rollback requests.
(3) PoE supports batching, out-of-order processing, and is signature-scheme agnostic and can be made to employ either MACs or threshold signatures.
(4) PoE does not rely on non-faulty replicas, clients, or trusted hardware to achieve safe and efficient consensus.
(5) To validate our vision of using PoE in resilient federated data management systems, we implement PoE and four other BFT protocols (Zyzzyva, Pbft, SBFT, and HotStuff) in our efficient ResilientDB[1] fabric [23–25, 27, 29, 30, 47].
(6) We extensively evaluate PoE against these protocols on a Google Cloud deployment consisting of 91 replicas and 320 k clients under (i) no failure, (ii) backup failure, (iii) primary failure, (iv) batching of requests, (v) zero payload, and (vi) scaling the number of replicas. Further, to prove the correctness of our results, we also stress test PoE and other protocols in a simulated environment. Our results show that PoE can achieve up to 80% more throughput than existing BFT protocols in the presence of failures.

To the best of our knowledge, PoE is the first protocol that achieves consensus in *only two phases* while being able to deal with Byzantine failures and without relying on trusted clients (e.g., Zyzzyva [38]) or on trusted hardware (e.g., MinBFT [50]). Hence, PoE can serve as a drop-in replacement of Pbft to improve scalability and performance in permissioned blockchain fabrics such as our ResilientDB fabric [27–31], MultiChain [20], and Hyperledger Fabric [4]; in multi-primary meta-protocols such as RCC [26, 28]; and in sharding protocols such as AHL [15].

2 ANALYSIS OF DESIGN PRINCIPLES

To arrive at an optimal design for PoE, we studied practices followed by state-of-the-art distributed data management systems and applied their principles to the design of PoE where possible. In Figure 1, we present a comparison of PoE against *four* well-known resilient consensus protocols.

Protocol	Phases	Messages	Resilience	Requirements
Zyzzyva	1	$O(\mathbf{n})$	0	Reliable clients and unsafe
PoE (our paper)	3	$O(3\mathbf{n})$	$\mathbf{f}$	Sign. agnostic
Pbft	3	$O(\mathbf{n} + 2\mathbf{n}^2)$	$\mathbf{f}$	
HotStuff	8	$O(8\mathbf{n})$	$\mathbf{f}$	Sequential Consensus
SBFT	5	$O(5\mathbf{n})$	0	Optimistic path

Figure 1: Comparison of BFT consensus protocols in a system with n replicas of which f are faulty. The costs given are for the normal-case behavior.

To illustrate the merits of PoE's design, we first briefly look at Pbft. The last phase of Pbft ensures that non-faulty replicas only execute requests and inform clients when there is a guarantee that such a transaction will be recovered after any failures. Hence, clients need to wait for only $\mathbf{f} + 1$ identical responses, of which at-least one is from a non-faulty replica, to ensure *guaranteed execution*. By eliminating this last phase, replicas speculatively execute requests before obtaining recovery guarantees. This impacts Pbft-style consensus in two ways:

(1) First, clients need a way to determine *proof-of-execution* after which they have a guarantee that their requests are executed and maintained by the system. We shall show that such a proof-of-execution can be obtained using $\mathbf{nf} \geq 2\mathbf{f} + 1$ identical responses (instead of $\mathbf{f} + 1$ responses).
(2) Second, as requests are executed before they are guaranteed, replicas need to be able to rollback requests that are dropped during periods of recovery.

PoE's speculative execution guarantees that requests with a proof-of-execution will never rollback and that only a single request can obtain a proof-of-execution per round. Hence, speculative execution provides the same strong consistency (safety) of Pbft in all cases, this at much lower cost under normal operations. Furthermore, we show that speculative execution is fully compatible with other scalable design principles applied to Pbft, e.g., batching and out-of-order processing to maximize throughput, even with high message delays.

Out-of-order execution. Typical BFT systems follow the *order-execute* model: first replicas agree on a unique order of the client request, and only then they execute the requests in order [9, 21, 29, 38, 51]. Unfortunately, this prevents these systems from providing any support for concurrent execution. A few BFT systems suggest executing prior to ordering, but even such systems need to re-verify their results prior to committing changes [4, 35]. Our PoE protocol lies between these two extremes: the replicas speculatively execute using only partial ordering guarantees. By doing so, PoE can eliminate communication costs and minimize latencies of typical BFT systems, this without needing to re-verify results in the normal case.

Out-of-order processing. Although BFT consensus typically executes requests in-order, this does not imply they need to process proposals to order requests sequentially. To maximize throughput, Pbft and other primary-backup protocols support *out-of-order processing* in which all available bandwidth of the primary is used to continuously propose requests (even when previous proposals are still being processed by the system). By doing so, out-of-order processing can eliminate the impact of high message delays. To provide out-of-order processing, all replicas will process any request proposed as the k-th request whenever k is within some *active window* bounded by a *low-watermark* and *high-watermark* [9]. These watermarks are increased as the

[1] ResilientDB is open-sourced at https://github.com/resilientdb.

system progresses. The size of this active window is—in practice—only limited by the memory resources available to replicas. As out-of-order processing is an essential technique to deliver high throughputs in environments with high message delays, we have included out-of-order processing in the design of PoE.

Twin-path consensus. Speculative execution employed by PoE is different that the *twin-path model* utilized by Zyzzyva [38] and SBFT [21]. These twin-path protocols have an optimistic *fast* path that works only if none of the replicas are *faulty* and require aid to determine whether these optimistic condition hold.

In the fast path of Zyzzyva, primaries propose requests, and backups directly execute such proposals and inform the client (without further coordination). The client waits for responses from all $\mathbf{n}$ replicas before marking the request executed. When the client does not receive $\mathbf{n}$ responses, it *timeouts* and sends a message to all replicas, after which the replicas perform an expensive client-dependent *slow-path* recovery process (which is prone to errors when communication is unreliable [2]).

The fast path of SBFT can deal with up to $\mathbf{c}$ crash-failures using $3\mathbf{f} + 2\mathbf{c} + 1$ replicas and uses threshold signatures to make communication linear. The fast path of SBFT requires a reliable collector and executor to aggregate messages and to send only *a single* (instead of at-least-$\mathbf{f} + 1$) response to the client. Due to aggregating execution, the fast path of SBFT still performs four rounds of communication before the client gets a response, whereas PoE only uses two rounds of communication (or three when PoE uses threshold signatures). If the fast path *timeouts* (e.g., the collector or executor fails), then SBFT falls back to a threshold-version of Pbft that takes an additional round before the client gets a response. Twin-path consensus is in sharp contrast with the design of PoE, which does not need outside aid (reliable clients, collectors, or executors), and can operate optimally even while dealing with replica failures.

Primary rotation. To minimize the influence of any single replica on bft consensus, HotStuff opts to replace the primary after every consensus decision. To efficiently do so, HotStuff uses an extra communication phase (as compared to Pbft), which minimizes the cost of primary replacement. Furthermore, HotStuff uses threshold signatures to make its communication linear (resulting in eight communication phases before a client gets responses). The event-based version of HotStuff can overlap phases of consecutive rounds, thereby assuring that consensus of a client request starts in every one-to-all-to-one communication phase. Unfortunately, the primary replacements require that all consensus rounds are performed in a strictly *sequential* manner, eliminating any possibility of *out-of-order processing*.

3 PROOF-OF-EXECUTION

In our *Proof-of-Execution consensus protocol* (PoE), the primary replica is responsible for proposing transactions requested by clients to all backup replicas. Each backup replica *speculatively* executes these transactions with the belief that the primary is behaving correctly. Speculative execution expedites processing of transactions in all cases. Finally, when malicious behavior is detected, replicas can recover by *rolling back transactions*, which ensures correctness without depending on any twin-path model.

3.1 System model and notations

Before providing a full description of our PoE protocol, we present the system model we use and the relevant notations.

A system is a set $\mathfrak{R}$ of *replicas* that process client requests. 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, manners. We assume that non-faulty replicas (those in $\mathfrak{R} \setminus \mathcal{F}$) behave in accordance to the protocol and are deterministic: on identical inputs, all non-faulty replicas must produce identical outputs. We do not make any assumptions on clients: all client can be malicious without affecting PoE. We write $\mathbf{n} = |\mathfrak{R}|$, $\mathbf{f} = |\mathcal{F}|$, and $\mathbf{nf} = |\mathfrak{R} \setminus \mathcal{F}|$ to denote the number of replicas, faulty replicas, and non-faulty replicas, respectively. We assume that $\mathbf{n} > 3\mathbf{f}$ ($\mathbf{nf} > 2\mathbf{f}$).

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. Depending on the type of message, we use message authentication codes (MACs) or threshold signatures (TSs) to achieve authenticated communication [36]. MACs are based on symmetric cryptography in which every pair of communicating nodes has a *secret key*. We expect non-faulty replicas to keep their *secret keys* hidden. TSs are based on asymmetric cryptography. In specific, each replica holds a distinct *private key*, which it can use to create a signature share. Next, one can produce a valid threshold signature given at least $\mathbf{nf}$ such signature shares (from distinct replicas). We write $s\langle v \rangle_i$ to denote the signature share of the i-th replica for signing value v. Anyone that receives a set $T = \{s\langle v \rangle_j \mid j \in T'\}$ of signature shares for v from $|T'| = \mathbf{nf}$ distinct replicas, can aggregate T into a single signature $\langle v \rangle$. This digital signature can then be verified using a public key.

We also employ a *collision-resistant cryptographic hash function* $\mathrm{D}(\cdot)$ that can map an arbitrary value v to a constant-sized digest $\mathrm{D}(v)$ [36]. We assume that it is practically impossible to find another value v', $v \neq v'$, such that $\mathrm{D}(v) = \mathrm{D}(v')$. We use notation $v||w$ to denotes the *concatenation* of two values v and w.

Next, we define the consensus provided by PoE.

Definition 3.1. A single run of any *consensus protocol* should satisfy the following requirements:

Termination. Each non-faulty replica executes a transaction.

Non-divergence. All non-faulty replicas execute the same transaction.

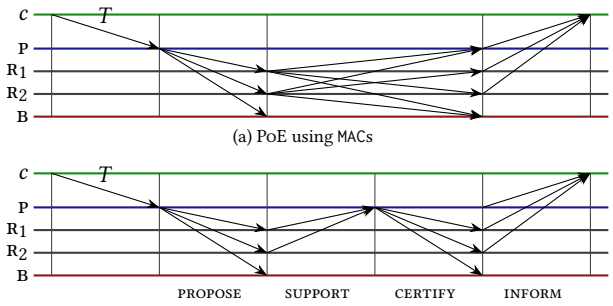
Termination is typically referred to as *liveness*, whereas non-divergence is typically referred to as *safety*. In PoE, execution is speculative: replicas can execute and rollback transactions. To provide safety, PoE provides speculative non-divergence instead of non-divergence:

Speculative non-divergence. If $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas accept and execute the same transaction T, then all non-faulty replicas will eventually accept and execute T (after rolling back any other executed transactions).

To provide *safety*, we do not need any other assumptions on communication or on clients. Due to well-known impossibility results for asynchronous consensus [19], we can only provide *liveness* in periods of *reliable bounded-delay communication* during which all messages sent by non-faulty replicas will arrive at their destination within some maximum delay.

3.2 The Normal-Case Algorithm of PoE

PoE operates in *views* $v = 0, 1, \ldots$. In view v, replica R with $\mathrm{id}(\mathrm{R}) = v \bmod \mathbf{n}$ is elected as the primary. The design of PoE relies on authenticated communication, which can be provided using MACs or TSs. In Figure 2, we sketch the normal-case working of PoE for both cases. For the sake of brevity, we will describe PoE built on top of TSs, which results in a protocol with low—*linear*—message complexity in the normal case. The full pseudo-code for



(a) PoE using MACs

(b) PoE using TSs.

Figure 2: Normal-case algorithm of PoE: Client c sends its request containing transaction T to the primary P, which proposes this request to all replicas. Although replica B is Byzantine, it fails to affect PoE.

this algorithm can be found in Figure 3. In Section 3.6, we detail the minimal changes to PoE necessary when switching to MACs.

Consider a view v with primary P. To request execution of transaction T, a client c signs transaction T and sends the signed transaction $\langle T \rangle_c$ to P. The usage of signatures assures that malicious primaries cannot forge transactions. To initiate replication and execution of T as the k-th transaction, the primary proposes T to all replicas via a PROPOSE message.

After the i-th replica R receives a PROPOSE message m from P, it checks whether at least **nf** other replicas received the same proposal m from primary P. This check assures R that at least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas received the same proposal, which will play a central role in achieving speculative non-divergence. To perform this check, each replica *supports* the first proposal m it receives from the primary by computing a *signature share* $s\langle m \rangle_i$ and sending a SUPPORT message containing this share to the primary.

The primary P waits for SUPPORT messages with valid signature shares from **nf** distinct replicas, which can then be aggregated into a single signature $\langle m \rangle$. After generating such a signature, the primary broadcasts this signature to all replicas via a CERTIFY message.

After a replica R receives a valid CERTIFY message, it *view-commits* to T as the k-th transaction in view v. The replica logs this view-commit decision as $\texttt{VCommit}_R(\langle T \rangle_c, v, k)$. After R view-commits to T, R schedules T for speculative execution as the k-th transaction of view v. Consequently, T will be executed by R after all preceding transactions are executed. We write $\texttt{Execute}_R(\langle T \rangle_c, v, k)$ to log this execution.

After execution, R informs the client of the order of execution and of execution result r (if any) via a message INFORM. In turn, client c will wait for a *proof-of-execution* for the transaction T it requested, which consists of identical INFORM messages from **nf** distinct replicas. This proof-of-execution guarantees that at least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas executed T as the k-th transaction and in Section 3.3, we will see that such transactions are always preserved by PoE when recovering from failures.

If client c does not know the current primary or does not get any timely response for its requests, then it can broadcast its request $\langle T \rangle_c$ to all replicas. The non-faulty replicas will then forward this request to the current primary (if T is not yet executed) and ensure that the primary initiates successful proposal of this request in a timely manner.

To prove correctness of PoE in all cases, we will need the following technical safety-related property of view-commits.

Client-role (used by client c to request transaction T) :

1: Send $\langle T \rangle_c$ to the primary P.
2: Await receipt of messages INFORM$(\langle T \rangle_c, v, k, r)$ from **nf** replicas.
3: Considers T executed, with result r, as the k-th transaction.

Primary-role (running at the primary P of view v, $\text{id}(\text{P}) = v \bmod \mathbf{n}$) :

4: Let view v start after execution of the k-th transaction.
5: **event** P awaits receipt of message $\langle T \rangle_c$ from client c **do**
6: Broadcast PROPOSE$(\langle T \rangle_c, v, k)$ to all replicas.
7: $k := k + 1$.
8: **end event**
9: **event** P receives **nf** message SUPPORT$(s\langle h \rangle_i, v, k)$ such that:
(1) each message was sent by a distinct replica, $i \in \{1, \ldots, n\}$; and
(2) All $s\langle h \rangle_i$ in this set can be combined to generate signature $\langle h \rangle$.
do
10: Broadcast CERTIFY$(\langle h \rangle, v, k)$ to all replicas.
11: **end event**

Backup-role (running at every i-th replica R.) :

12: **event** R receives message $m :=$ PROPOSE$(\langle T \rangle_c, v, k)$ such that:
(1) v is the current view;
(2) m is sent by the primary of v; and
(3) R did not accept a k-th proposal in v
do
13: Compute $h := \text{D}(\langle T \rangle_c || v || k)$.
14: Compute signature share $s\langle h \rangle_i$.
15: Transmit SUPPORT$(s\langle h \rangle_i, v, k)$ to P.
16: **end event**
17: **event** R receives messages CERTIFY$(\langle h \rangle, v, k)$ from P such that:
(1) R transmitted SUPPORT$(s\langle h \rangle_i, v, k)$ to P; and
(2) $\langle h \rangle$ is a valid threshold signature
do
18: View-commit T, the k-th transaction of v ($\texttt{VCommit}_R(\langle T \rangle_c, v, k)$).
19: **end event**
20: **event** R logged $\texttt{VCommit}_R(\langle T \rangle_c, v, k)$ and
has logged $\texttt{Execute}_R(t', v', k')$ for all $0 \leq k' < k$ **do**
21: Execute T as the k-th transaction of v ($\texttt{Execute}_R(\langle T \rangle_c, v, k)$).
22: Let r be the result of execution of T (if there is any result).
23: Send INFORM$(\text{D}(\langle T \rangle_c), v, k, r)$ to c.
24: **end event**

Figure 3: The normal-case algorithm of PoE.

PROPOSITION 3.2. *Let* R_i, $i \in \{1, 2\}$, *be two non-faulty replicas that view-committed to* $\langle T_i \rangle_{c_i}$ *as the* k*-th transaction of view* v ($\texttt{VCommit}_R(\langle T \rangle_c, v, k)$). *If* $\mathbf{n} > 3\mathbf{f}$, *then* $\langle T_1 \rangle_{c_1} = \langle T_2 \rangle_{c_2}$.

PROOF. Replica R_i only view-committed to $\langle T_i \rangle_{c_i}$ after R_i received CERTIFY$(\langle h \rangle, v, k)$ from the primary P (Line 17 of Figure 3). This message includes a threshold signature $\langle h \rangle$, whose construction requires signature shares from a set S_i of **nf** distinct replicas. Let $X_i = S_i \setminus \mathcal{F}$ be the non-faulty replicas in S_i. As $|S_i| = \mathbf{nf}$ and $|\mathcal{F}| = \mathbf{f}$, we have $|X_i| \geq \mathbf{nf} - \mathbf{f}$. The non-faulty replicas in X_i will only send a single SUPPORT message for the k-th transaction in view v (Line 12 of Figure 3). Hence, if $\langle T_1 \rangle_{c_1} \neq \langle T_2 \rangle_{c_2}$, then X_1 and X_2 must not overlap and $\mathbf{nf} \geq |X_1 \cup X_2| \geq 2(\mathbf{nf} - \mathbf{f})$ must hold. As $\mathbf{n} = \mathbf{nf} + \mathbf{f}$, this simplifies to $3\mathbf{f} \geq \mathbf{n}$, which contradicts $\mathbf{n} > 3\mathbf{f}$. Hence, we conclude $\langle T_1 \rangle_{c_1} = \langle T_2 \rangle_{c_2}$. □

We will later use Proposition 3.2 to show that PoE provides speculative non-divergence. Next, we look at typical cases in which the normal-case of PoE is interrupted:

Example 3.3. A malicious primary can try to affect PoE by not conforming to the normal-case algorithm in the following ways:

(1) By sending proposals for different transactions to different non-faulty replicas. In this case, Proposition 3.2 guarantees that at most a single such proposed transaction will get view-committed by any non-faulty replica.
(2) By keeping some non-faulty replicas in the dark by not sending proposals to them. In this case, the remaining non-faulty replicas can still end up view-committing the transactions as long as at least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas receive proposals: the faulty replicas in $\mathcal{F}$ can take over the

role of up to **f** non-faulty replicas left in the dark (giving the false illusion that the non-faulty replicas in the dark are malicious).

(3) By preventing execution by not proposing a k-th transaction, even though transactions following the k-th transaction are being proposed.

When the network is unreliable and messages do not get delivered (or not on time), then the behavior of a non-faulty primary can match that of the malicious primary in the above example. Indeed, failure of the normal-case of PoE has only two possible causes: primary failure and unreliable communication. If communication is unreliable, then there is no way to guarantee continuous service [19]. Hence, replicas simply assume failure of the current primary if the normal-case behavior of PoE is interrupted, while the design of PoE guarantees that unreliable communication does not affect the correctness of PoE.

To deal with primary failure, each replica maintains a timer for each request. If this timer expires (*timeout*) and it has not been able to execute the request, it assumes that the primary is malicious. To deal with such a failure, replicas will replace the primary. Next, we present the *view-change algorithm* that performs primary replacement.

3.3 The View-Change Algorithm

If PoE observes failure of the primary P of view v, then PoE will elect a new primary and move to the next view, view $v+1$, via the *view-change algorithm*. The goals of the view-change are

(1) to assure that each request that *is considered executed* by any client is preserved under all circumstances; and

(2) to assure that the replicas are able to agree on a new view whenever communication is reliable.

As described in the previous section, a client will consider its request executed if it receives a *proof-of-execution* consisting of identical INFORM responses from at-least **nf** distinct replicas. Of these **nf** responses, at-most **f** can come from faulty replicas. Hence, a client can only consider its request executed whenever the requested transaction was executed (and view-committed) by at-least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas in the system. We note the similarity with the view-change algorithm of PBFT, which will preserve any request that is *prepared* by at-least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas.

The view-change algorithm of PoE consists of three steps. First, failure of the current primary P needs to be detected by all non-faulty replicas. Second, all replicas exchange information to establish which transactions were included in view v and which were not. Third, the new primary P′ proposes a new view. This new view proposal contains a list of the transactions executed in the previous views (based on the information exchanged earlier). Finally, if the new view proposal is valid, then replicas switch to this view; otherwise, replicas detect failure of P′ and initiate a view-change for the next view ($v+2$). The communication of the view-change algorithm of PoE is sketched in Figure 4 and the full pseudo-code of the algorithm can be found in Figure 5. Next, we discuss each step in detail.

3.3.1 Failure Detection and View-Change Requests. If a replica R detects failure of the primary of view v, then it halts the normal-case algorithm of PoE for view v and informs all other replicas of this failure by requesting a view-change. The replica R does so by broadcasting a message VC-REQUEST(v, E), in which E is a summary of all transactions executed by R (Figure 5, Line 1). Each replica R can detect the failure of primary in two ways:

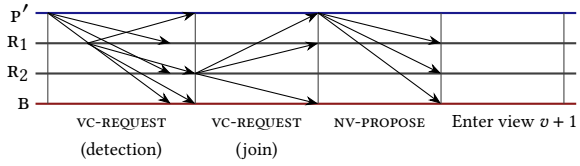


Figure 4: The current primary B of view v is faulty and needs to be replaced. The next primary, P′, and the replica R_1 detected this failure first and request view-change via VC-REQUEST messages. The replica R_2 joins these requests.

vc-request (used by replica R to request view-change) **:**

1: **event** R detects failure of the primary **do**
2: R halts the normal-case algorithm of Figure 3 for view v.
3: $E := \{(\text{CERTIFY}(\langle h \rangle, w, k), \langle T \rangle_c) \mid w \leq v$ and $\text{Execute}_R(\langle T \rangle_c, w, k)$ and $h = \text{D}(\langle T \rangle_c || w || k)\}$.
4: Broadcast VC-REQUEST(v, E) to all replicas.
5: **end event**
6: **event** R receives **f** + 1 messages VC-REQUEST(v_i, E_i) such that
(1) each message was sent by a distinct replica; and
(2) v_i, $1 \leq i \leq \mathbf{f}+1$, is the current view
do
7: R detects failure of the primary (join).
8: **end event**

On receiving nv-propose (use by replica R) **:**

9: **event** R receives $m =$ NV-PROPOSE$(v+1, m_1, m_2, ..., m_{\mathbf{nf}})$ **do**
10: **if** m is a valid new-view proposal (similar to creating NV-PROPOSE) **then**
11: Derive the transactions N for the new-view from $m_1, m_2, \ldots, m_{\mathbf{nf}}$.
12: Rollback any executed transactions not included in N.
13: Execute the transactions in N not yet executed.
14: Move into view $v+1$ (see Section 3.3.3 for details).
15: **end if**
16: **end event**

nv-propose (used by replica P′ that will act as the new primary) **:**

17: **event** P′ receives **nf** messages $m_i =$ VC-REQUEST(v_i, E_i) such that
(1) these messages are sent by a set S, $|S| = \mathbf{nf}$, of distinct replicas;
(2) for each m_i, $1 \leq i \leq \mathbf{nf}$, sent by replica $Q_i \in S$, E_i consists of a consecutive sequence of entries (CERTIFY$(\langle h \rangle, v, k)$, $\langle T \rangle_c$);
(3) v_i, $1 \leq i \leq \mathbf{nf}$, is the current view v; and
(4) P′ is the next primary (id(P′) $= (v+1) \bmod \mathbf{n}$)
do
18: Broadcast NV-PROPOSE$(v+1, m_1, m_2, ..., m_{\mathbf{nf}})$ to all replicas.
19: **end event**

Figure 5: The view-change algorithm of PoE.

(1) R *timeouts* while expecting normal-case operations toward executing a client request. E.g., when R forwards a client request to the current primary, and the current primary fails to propose this request on time.

(2) R receives VC-REQUEST messages, indicating that the primary of view v failed, from **f** + 1 distinct replicas. As at most **f** of these messages can come from faulty replicas, at least one non-faulty replica must have detected a failure. In this case, R joins the view-change (Figure 5, Line 6).

3.3.2 Proposing the New View. To start view $v+1$, the new primary P′ (with id(P′) $= (v+1) \bmod \mathbf{n}$) needs to propose a new view by determining a valid list of requests that need to be preserved. To do so, P′ waits until it receives sufficient information. In specific, P′ waits until it received *valid* VC-REQUEST messages from a set $S \subseteq \mathfrak{R}$ of $|S| = \mathbf{nf}$ distinct replicas.

An i-th view-change request m_i is considered valid if it includes a *consecutive sequence* of pairs $(c, \langle T \rangle_c)$, where c is a valid CERTIFY message for request $\langle T \rangle_c$. Such a set S is guaranteed to exist when communication is reliable, as all non-faulty replicas will participate in the view-change algorithm. The new primary

collects the set S of $|S| = \mathbf{nf}$ valid VC-REQUEST and proposes them in a new view message NV-PROPOSE to all replicas.

3.3.3 Move to the New View. After a replica R receives a NV-PROPOSE message containing a new-view proposal from the new primary P′, R validates the content of this message. From the set of VC-REQUEST messages in the new-view proposal, R chooses, for each k, the pair $(\text{CERTIFY}(\langle h \rangle, w, k), \langle T \rangle_c)$ proposed in the most-recent view w. Furthermore, R determines the total number of such requests $k_{\max}$. Then, R view-commits and executes all $k_{\max}$ chosen requests that happened before view $v + 1$. Notice that replica R can skip execution of any transaction it already executed. If R executed transactions not included in the new-view proposal, then R needs to *rollback* these transactions before it can proceed executing requests in view $v + 1$. After these steps, R can switch to the new view $v + 1$. In the new view, the new primary P′ starts by proposing the $k_{\max} + 1$-th transaction.

When moving into the new view, we see the cost of speculative execution: some replicas can be forced to *rollback execution* of transactions:

Example 3.4. Consider a system with non-faulty replica R. When deciding the k-th request, communication became unreliable, due to which only R received a CERTIFY message for request $\langle T \rangle_c$. Consequently, R speculatively executes T and informs the client c. During the view-change, all other replicas—none of which have a CERTIFY message for $\langle T \rangle_c$—provide their local state to the new primary, which proposes a new view that does not include any k-th request. Hence, the new primary will start its view by proposing client request $\langle T' \rangle_{c'}$ as the k-th request, which gets accepted. Consequently, R needs to rollback execution of T. Luckily, this is not an issue: the client c only got at-most $\mathbf{f}+1 < \mathbf{nf}$ responses for request, does not yet have a proof-of-execution, and, consequently, does not consider T executed.

In practice, rollbacks can be supported by, e.g., undoing the operations of transaction in reverse order, or by reverting to an old state. For the correct working of PoE, the exact working of rollbacks is not important as long as the execution layer provides support for rollbacks.

3.4 Correctness of PoE

First, we show that the normal-case algorithm of PoE provides non-divergent speculative consensus when the primary is non-faulty and communication is reliable.

THEOREM 3.5. *Consider a system in view v, in which the first $k-1$ transactions have been executed by all non-faulty replicas, in which the primary is non-faulty, and communication is reliable. If the primary received $\langle T \rangle_c$, then the primary can use the algorithm in Figure 3 to ensure that*

(1) there is non-divergent execution of T;
(2) c considers T executed as the k-th transaction; and
(3) c learns the result of executing T (if any),

this independent of any malicious behavior by faulty replicas.

PROOF. Each non-faulty primary would follow the algorithm of PoE described in Figure 3 and send $\text{PROPOSE}(\langle T \rangle_c, v, k)$ to all replicas (Line 6). In response, all **nf** non-faulty replicas will compute a signature share and send a SUPPORT message to the primary (Line 15). Consequently, the primary will receive signature shares from **nf** replicas and will combine them to generate a threshold signature $\langle h \rangle$. The primary will include this signature $\langle h \rangle$ in a CERTIFY message and broadcast it to all replicas. Each replica will successfully verify $\langle h \rangle$ and will view-commit to T (Line 17). As the first $k-1$ transactions have already been executed, every non-faulty replica will execute T. As all non-faulty replicas behave deterministically, execution will yield the same result r (if any) across all non-faulty replicas. Hence, when the non-faulty replicas inform c, they do so by all sending identical messages $\text{INFORM}(\text{D}(\langle T \rangle_c), v, k, r)$ to c (Line 20–Line 23). As all **nf** non-faulty replicas executed T, we have non-divergent execution. Finally, as there are at most **f** faulty replicas, the faulty replicas can only forge up to **f** invalid INFORM messages. Consequently, the client c will only receive the message $\text{INFORM}(\text{D}(\langle T \rangle_c), v, k, r)$ from at least **nf** distinct replicas, and will conclude that T is executed yielding result r (Line 3). □

At the core of the correctness of PoE, under all conditions, is that no replica will rollback requests $\langle T \rangle_c$ for which client c already received a proof-of-execution. We prove this next:

PROPOSITION 3.6. *Let $\langle T \rangle_c$ be a request for which client c already received a proof-of-execution showing that T was executed as the k-th transaction of view v. If $\mathbf{n} > 3\mathbf{f}$, then every non-faulty replica that switches to a view $v' > v$ will preserve T as the k-th transaction of view v.*

PROOF. Client c considers $\langle T \rangle_c$ executed as the k-th transaction of view v when it received identical INFORM-messages for T from a set A of $|A| = \mathbf{nf}$ distinct replicas (Figure 3, Line 3). Let $B = A \setminus \mathcal{F}$ be the set of non-faulty replicas in A.

Now consider a non-faulty replica R that switches to view $v' > v$. Before doing so, R must have received a valid proposal $m = \text{NV-PROPOSE}(v', m_1, ..., m_{\mathbf{nf}})$ from the primary of view v'. Let C be the set of **nf** distinct replicas that provided messages $m_1, \ldots, m_{\mathbf{nf}}$ and let $D = C \setminus \mathcal{F}$ be the set of non-faulty replicas in C. We have $|B| \geq \mathbf{nf} - \mathbf{f}$ and $|D| \geq \mathbf{nf} - \mathbf{f}$. Hence, using a contradiction argument similar to the one in the proof of Proposition 3.2, we conclude that there must exists a non-faulty replica $\text{Q} \in (B \cap D)$ that executed $\langle T \rangle_c$, informed c, and requested a view-change.

To complete the proof, we need to show that $\langle T \rangle_c$ was proposed and executed in the last view that proposed and view-committed a k-th transaction and, hence, that Q will include $\langle T \rangle_c$ in its VC-REQUEST message for view v'. We do so by induction on the difference $v' - v$. As the base case, we have $v' - v = 1$, in which case no view after v exists yet and, hence, $\langle T \rangle_c$ must be the newest k-th transaction available to Q. As the induction hypothesis, we assume that all non-faulty replicas will preserve T when entering a new view w, $v < w \leq w'$. Hence, non-faulty replicas participating in view w will not support any k-th transactions proposed in view w. Consequently, no CERTIFY messages can be constructed for any k-th transaction in view w. Hence, the new-view proposal for $w' + 1$ will include $\langle T \rangle_c$, completing the proof. □

As a direct consequence of the above, we have

COROLLARY 3.7 (SAFETY OF POE). *PoE provides speculative non-divergence if $\mathbf{n} > 3\mathbf{f}$.*

We notice that the view-change algorithm does not deal with minor malicious behavior (e.g., a single replica left in the dark). Furthermore, the presented view-change algorithm will recover all transactions since the start of the system, which will result in unreasonable large messages when many transactions have already been proposed. In practice, both these issues can be resolved by regularly making *checkpoints* (e.g., after every 100 requests) and only including requests since the last checkpoint in each VC-REQUEST message. To do so, PoE uses a standard fully-decentralized PBFT-style checkpoint algorithm that enables the independent checkpointing and recovery of any request that is

executed by at least $\mathbf{f}+1$ non-faulty replicas whenever communication is reliable [9]. Finally, utilizing the view-change algorithm and checkpoints, we prove

Theorem 3.8 (Liveness of PoE). *PoE provides termination in periods of reliable bounded-delay communication if* $\mathbf{n} > 3\mathbf{f}$.

Proof. When the primary is non-faulty, Theorem 3.5 guarantees termination as replicas continuously accept and execute requests. If the primary is Byzantine and fails to guarantee termination for at most $\mathbf{f}$ non-faulty replicas, then the checkpoint algorithm will assure termination of these non-faulty replicas. Finally, if the primary is Byzantine and fails to guarantee termination for at least $\mathbf{f}+1$ non-faulty replicas, then it will be replaced using the view-change algorithm. For the view-change process, each replica will start with a timeout δ after it receives $\mathbf{nf}$ matching vc-requests and double this timeout after each view-change (exponential backoff). When communication becomes reliable, this mechanism guarantees that all replicas will eventually view-change to the same view at the same time. After this point, a non-faulty replica will become primary in at most $\mathbf{f}$ view-changes, after which Theorem 3.5 guarantees termination. □

3.5 Fine-Tuning and Optimizations

To keep presentation simple, we did not include the following optimizations in the protocol description:

(1) To reach $\mathbf{nf}$ signature shares, the primary can generate one itself. Hence, it only needs $\mathbf{nf}-1$ shares of other replicas.
(2) The propose, support, inform, and nv-propose messages are not forwarded and only need MACs to provide message authentication. The certify messages need not be signed, as tampering them would invalidate the threshold signature. The vc-request messages need to be signed, as they need to be forwarded without tampering.

Finally, the design of PoE is fully compatible with *out-of-order processing* as a replica only supports proposals for a k-th transaction if it had not previously supported another k-th proposal (Figure 3, Line 12) and only executes a k-th transaction if it has already executed all the preceding transactions (Figure 3, Line 20). As the size of the active out-of-order processing window determines how many client requests are being processed at the same time (without receiving a proof-of-execution), the size of the active window determines the number of transactions that can be rolled back during view-changes.

3.6 Designing PoE using MACs

The design of PoE can be adapted to only use message authentication codes (MACs) to authenticate communication. This will sharply reduce the computational complexity of PoE and eliminate one round of communication, this at the cost of higher *quadratic* overall communication costs (see Figure 2).

The usage of only MACs makes it impossible to obtain threshold signatures or reliably forward messages (as forwarding replicas can tamper with the content of unsigned messages). Hence, using MACs requires changes to how client requests are included in proposals (as client requests are forwarded), to the normal-case algorithm of PoE (which uses threshold signatures), and to the view-change algorithm of PoE (which forwards vc-request messages). The changes to the proposal of client requests and to the view-change algorithm can be derived from the strategies used by Pbft to support MACs [9]. Hence, next we only review the changes to the normal-case algorithm of PoE.

Consider a replica r that receives a propose message from the primary p. Next, r needs to determine whether at least $\mathbf{nf}$ other replicas received the same proposal, which is required to achieve speculative non-divergence (see Proposition 3.2). When using MACs, r can do so by replacing the all-to-one support and one-to-all certify phases by a single all-to-all *support phase*. In the support phase, each replica agrees to *support* the first proposal $\text{propose}(\langle T \rangle_c, v, k)$ it receives from the primary by broadcasting a message $\text{support}(\text{D}(\langle T \rangle_c), v, k)$ to all replicas. After this broadcast, each replica waits until it receives support messages, identical to the message it sent, from $\mathbf{nf}$ distinct replicas. If r receives these messages, it *view-commits* to T as the k-th transaction in view v and schedules T for execution. We have sketched this algorithm in Figure 2.

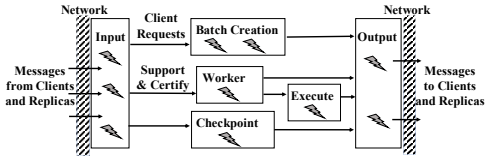


Figure 6: Multi-threaded Pipelines at different replicas.

4 RESILIENTDB FABRIC

To test our design principles in practical settings, we implement our PoE protocol in our ResilientDB fabric [27–31]. ResilientDB provides its users access to a state-of-the-art replicated transactional engine and fulfills the need of a high-throughput permissioned blockchain fabric. ResilientDB helps us to realize the following goals: (i) implement and test different consensus protocols; (ii) balance the tasks done by a replica through a *parallel pipelined architecture*; (iii) minimize the cost of communication through *batching* client transactions; and (iv) enable use of a secure and efficient ledger. Next, we present a brief overview of our ResilientDB fabric.

ResilientDB lays down a *client-server* architecture where clients send their transactions to servers for processing. We use Figure 6 to illustrate the multi-threaded pipelined architecture associated with each replica. At each replica, we spawn multiple *input* and *output* threads for communicating with the network.

Batching. During our formal description of PoE, we assumed that the propose message from the primary includes a single client request. An effective way to reduce the overall cost of consensus is by aggregating several client requests in a single batch and use one consensus step to reach agreement on all these requests [9, 21, 38]. To maximize performance, ResilientDB facilitates batching requests at both replicas and clients.

At the primary replica, we spawn multiple *batch-threads* that aggregate clients requests into a batch. The input-threads at the primary receive client requests, assign them a sequence number and enqueue these requests in the *batch-queue*. In ResilientDB, all batch-threads share a common *lock-free queue*. When a client request is available, a batch-thread dequeues the request and continues adding it to an existing batch until the batch has reached a pre-defined size. Each batching-thread also hashes the requests in a batch to create a unique digest.

All other messages received at a replica are enqueued by the input-thread in the *work-queue* to be processed by the single *worker-thread*. Once a replica receive a certify message from the primary, it forwards the request to the *execute-thread* for execution. Once the execution is complete, the execution-thread creates an inform message, which is transmitted to the client.

Ledger Management. We now explain how we efficiently maintain a blockchain ledger across different replicas. A blockchain is an immutable ledger, where blocks are chained as a linked-list. An i-th block can be represented as $B_i := \{k, d, v,$

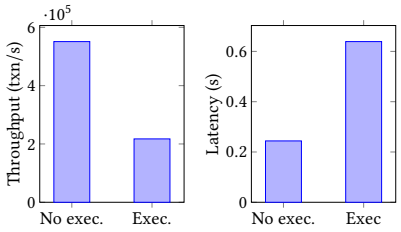


Figure 7: Upper bound on performance when primary only replies to clients (*No exec.*) and when primary executes a request and replies to clients (*Exec.*).

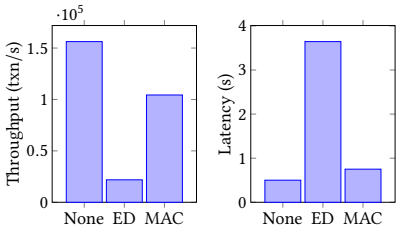


Figure 8: System performance using three different signature schemes. In all cases, n = 16 replicas participate in consensus.

$H(B_{i-1})\}$, in which k is the sequence number of the client request, d the digest of the request, v the view number, and $H(B_{i-1})$ the hash of the previous block. In ResilientDB, prior to any consensus, we require the *first* primary replica to create a *genesis block* [31]. This genesis block acts as the first block in the blockchain and contains some basic data. We use the hash of the identity of the initial primary, as this information is available to each participating replicas (eliminating the need for any extra communication to exchange this block).

After the genesis block, each replica can independently create the next block in the blockchain. As stated above, each block corresponds to some batch of transactions. A block is only created by the execute-thread once it completes executing a batch of transactions. To create a block, the execute-thread hashes the previous block in the blockchain and creates a *new block*. To prove the validity of individual blocks, ResilientDB stores the *proof-of-accepting the k-th request* in the k-th block. In PoE, such a proof includes the threshold signature sent by the primary as part of the certify message.

5 EVALUATION

We now analyze our design principles in practice. To do so, we evaluate our PoE protocol against four state-of-the-art bft protocols. There are many bft protocols we could compare with. Hence, we pick a representative sample: (1) Zyzzyva—as it has the absolute minimal cost in the fault-free case, (2) Pbft—as it is a common baseline (the used design is based on BFTSmart [7]), (3) SBFT—as it is a safer variation of Zyzzyva, and (3) HotStuff—as it is a linear-communication protocol that adopts the notion of rotating leaders. Through our experiments, we want to answer the following questions:

(Q1) How does PoE fare in comparison with the other protocols under failures?
(Q2) Does PoE benefits from batching client requests?
(Q3) How does PoE perform under zero payload?
(Q4) How scalable is PoE on increasing the number of replicas participating in the consensus, in the normal-case?

Setup. We run our experiments on the Google Cloud, and deploy each replicas on a *c2* machine having a 16-core Intel Xeon Cascade Lake CPU running at 3.8 GHz with 32 GB memory. We deploy up to 320 k clients on 16 machines. To collect results after reaching a steady-state, we run each experiment for 180 s: the first 60 s are warmup, and measurement results are collected over the next 120 s. We average our results over three runs.

Configuration and Benchmarking. For evaluating the protocols, we employed YCSB [13] from Blockbench's macro benchmarks [16]. Each client request queries a YCSB table that holds half a million active records. We require 90% of the requests to be write queries as the majority of typical blockchain transactions are updates to existing records. Prior to the experiments, each replica is initialized with an identical copy of the YCSB table. The client requests generated by YCSB follow a Zipfian distribution and are heavily skewed (skew factor 0.9).

Unless *explicitly* stated, we use the following configuration for all experiments. We perform scaling experiments by varying replicas from 4 to 91. We divide our experiments in two dimensions: (1) *Zero Payload* or *Standard Payload*, and (2) *Failures* or *Non-Failures*. We employ batching with a batch size of 100 as the percentage increase in throughput on larger batch sizes is small.

Under Zero Payload conditions, all replicas execute 100 dummy instructions per batch, while the primary sends an empty proposal (and not a batch of 100 requests). Under Standard Payload, with a batch size of 100, the size of Propose message is 5400 B, of Response message is 1748 B, and of other messages is around 250 B. For experiments with failures, we force one backup replica to crash. Additionally, we present an experiment that illustrates the effect of primary failure. 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.

Other protocols: We also implement Pbft, Zyzzyva, SBFT and HotStuff in our ResilientDB fabric. We refer to Section 2 for further details on the working of Zyzzyva, SBFT, and HotStuff. Our implementation of Pbft is based on the BFTSmart [7] framework with the added benefits of out-of-order processing, pipelining, and multi-threading. In both Pbft and Zyzzyva, digital signatures are used for authenticating messages sent by the clients, while MACs are used for other messages. Both SBFT and HotStuff require threshold signatures for their communication.

5.1 System Characterization

We first determine the upper bounds on the performance of ResilientDB. In Figure 7, we present the maximum throughput and latency of ResilientDB when there is *no communication* among the replicas. We use the term *No Execution* to refer to the case where all clients send their request to the primary replica and primary simply responds back to the client. We count every query responded back in the system throughput. We use the term *Execution* to refer to the case where the primary replica executes each query before responding back to the client.

The architecture of ResilientDB (see Section 4) states the use of one worker thread. In these experiments, we maximize system performance by allowing up to two threads to work independently at the primary replica without ordering any queries. Our results indicate that the system can attain high throughputs (up to 500 ktxn/s) and can reach low latencies (up to 0.25 s). Notice that if we employ additional worker-threads, our ResilientDB fabric can easily attain higher throughput.

5.2 Effect of Cryptographic Signatures.

ResilientDB enables a flexible design where replicas and clients can employ both digital signatures (threshold signatures) and

message authentication codes. This helps us to implement PoE and other consensus protocols in ResilientDB.

To achieve authenticated communication using symmetric cryptography, we employ a combination of CMAC and AES [36]. Further, we employ ED25519-based digital signatures to enable asymmetric cryptographic signing. For generating efficient threshold signature scheme, we use Boneh–Lynn–Shacham (BLS) signatures [36]. To create message digests and for hashing purposes, we use the SHA256 algorithm.

Next, we determine the cost of different cryptographic signing schemes. For this purpose, we run three different experiments in which (i) no signature scheme is used (*None*); (ii) everyone uses digital signatures based on ED25519 (*ED*); and (iii) all replicas use CMAC+AES for signing, while clients sign their message using ED25519 (*MAC*). In these three experiments, we run Pbft consensus among 16 replicas. In Figure 8, we illustrate the throughput attained and latency incurred by ResilientDB for the experiments. Clearly, the system attains its highest throughput when no signatures are employed. However, such a system cannot handle malicious attacks. Further, using just digital signatures for signing messages can prove to be expensive. An optimal configuration can require clients to sign their messages using digital signatures, while replicas can communicate using MACs.

5.3 Scaling Replicas under Standard Payload

In this section, we evaluate scalability of PoE both under backup failure and no failures.

(1) ***Single Backup Failure.*** We use Figures 9(a) and 9(b) to illustrate the throughput and latency attained by the system on running different consensus protocols under a backup failure. These graphs affirm our claim that PoE attains higher throughput and incurs lower latency than all other protocols.

In case of Pbft, each replica participates in two phases of quadratic communication, which limits its throughput. For the twin-path protocols such as Zyzzyva and SBFT, a single failure is sufficient to cause massive reductions in their system throughputs. Notice that the collector in SBFT and the clients in Zyzzyva have to wait for messages from all **n** replicas, respectively. As predicting an optimal value for timeouts is hard [11, 12], we chose a very small value for the timeout (3 s) for replicas and clients. We justify these values, as the experiments we show later in this section show that the average latency can be as large as 6 s. We note that high timeouts affect Zyzzyva more than SBFT. In Zyzzyva, clients are waiting for timeouts during which they stop sending requests, which empties the pipeline at the primary, starving it from new request to propose. To alleviate such issues in real-world deployments of Zyzzyva, clients need to be able to precisely predict the latency to minimize the time the clients needs to wait between requests. Unfortunately, this is hard and runs the risk of ending up in the expensive slow path of Zyzzyva whenever the predicted latency is slightly off. In SBFT, the collector may timeout waiting for threshold shares for the k-th round while the primary can continues propose requests for future round l, $l > k$. Hence, in SBFT replicas have more opportunity to occupy themselves with useful work.

HotStuff attains significantly low throughput due to its sequential primary-rotation model in which each of its primaries has to wait for the previous primary before proposing the next request, which leads to a huge reduction in its throughput. Interestingly, HotStuff incurs the least average latency among all protocols. This is a result of intensive load on the system when running other protocols. As these protocols process several requests concurrently (see the multi-threaded architecture in Section 4), these requests spend on average more time in the queue before being processed by a replica. Notice that all out-of-order consensus protocols employ this trade off: a small sacrifice on latency yields higher gains on system throughput.

In case of PoE, its high throughputs under failures is a result of its three-phase linear protocol that does not rely on any twin-path model. To summarize, PoE attains up to 43%, 72%, 24× and 62× more throughputs than Pbft, SBFT, HotStuff and Zyzzyva.

(2) ***No Replica Failure.*** We use Figures 9(c) and 9(d) to illustrate the throughput and latency attained by the system on running different consensus protocols in fault-free conditions. These plots help us to bound the maximum throughput that can be attained by different consensus protocols in our system.

First, as expected, in comparison to the Figures 9(a) and 9(b), the throughputs for PoE and Pbft are slightly higher. Second, PoE continues to outperform both Pbft and HotStuff, for the reasons described earlier. Third, both Zyzzyva and SBFT are now attaining higher throughputs as their clients and collector no longer timeout, respectively. The key reason SBFT's gains are limited is because SBFT requires five phases and becomes computation bounded. Although Pbft is quadratic, it employs MAC, which are cheaper to sign and verify.

Notice that the differences in throughputs of PoE and Zyzzyva are small. PoE has 20% (on 91 replicas) to 13% (on 4 replicas) less throughputs than Zyzzyva. An interesting observation is that on 91 replicas, Zyzzyva incurs almost the same latency as PoE, even though it has higher throughput. This happens as clients in PoE have to wait for only the fastest $\mathbf{nf} = 61$ replies, whereas a client for Zyzzyva has to wait for replies from all replicas (even the slowest ones). To conclude, PoE attains up to 35%, 27% and 21× more throughput than Pbft, SBFT and HotStuff, respectively.

5.4 Scaling Replicas under Zero Payload

We now measure the performance of different protocols under zero payload. In any bft protocol, the primary starts consensus by sending a Propose message that includes all transactions. As a result, this message has the largest size and is responsible for consuming the majority of the bandwidth. A zero payload experiment ensures that each replica executes dummy instructions. Hence, the primary is no longer a bottleneck.

We again run these experiments for both ***Single Failure*** and ***Failure-Free*** cases, and use Figures 9(e) to 9(h) to illustrate our observations. It is evident from these figures that zero payload experiments have helped in increasing PoE's gains. PoE attains up to 85%, 62% and 27× more throughputs than Pbft, SBFT and HotStuff, respectively. In fact, under failure-free conditions, the throughput attained by PoE is comparable to Zyzzyva. This is easily explained. First, both PoE and Zyzzyva are linear protocols. Second, although in failure-free cases Zyzzyva attains consensus in one phase, its clients need to wait for response from all **n** replicas, which gives PoE an opportunity to cover the gap. However, SBFT being a linear protocol does not perform as good as its other linear counterparts. Its throughput is impacted by the delay of five phases.

5.5 Impact of Batching under Failures

Next, we study the effect of batching client requests on bft protocols [9, 51]. To answer this question, we measure performance as function of the number of requests in a batch (*the batch-size*), which we vary between 10 and 400. For this experiment, we use a system with 32 available replicas, of which one replica has failed.

We use Figures 9(i) and 9(j) to illustrate, for each consensus protocol, the throughput and average latency attained by the system. For each protocol, increasing the batch-size also increases

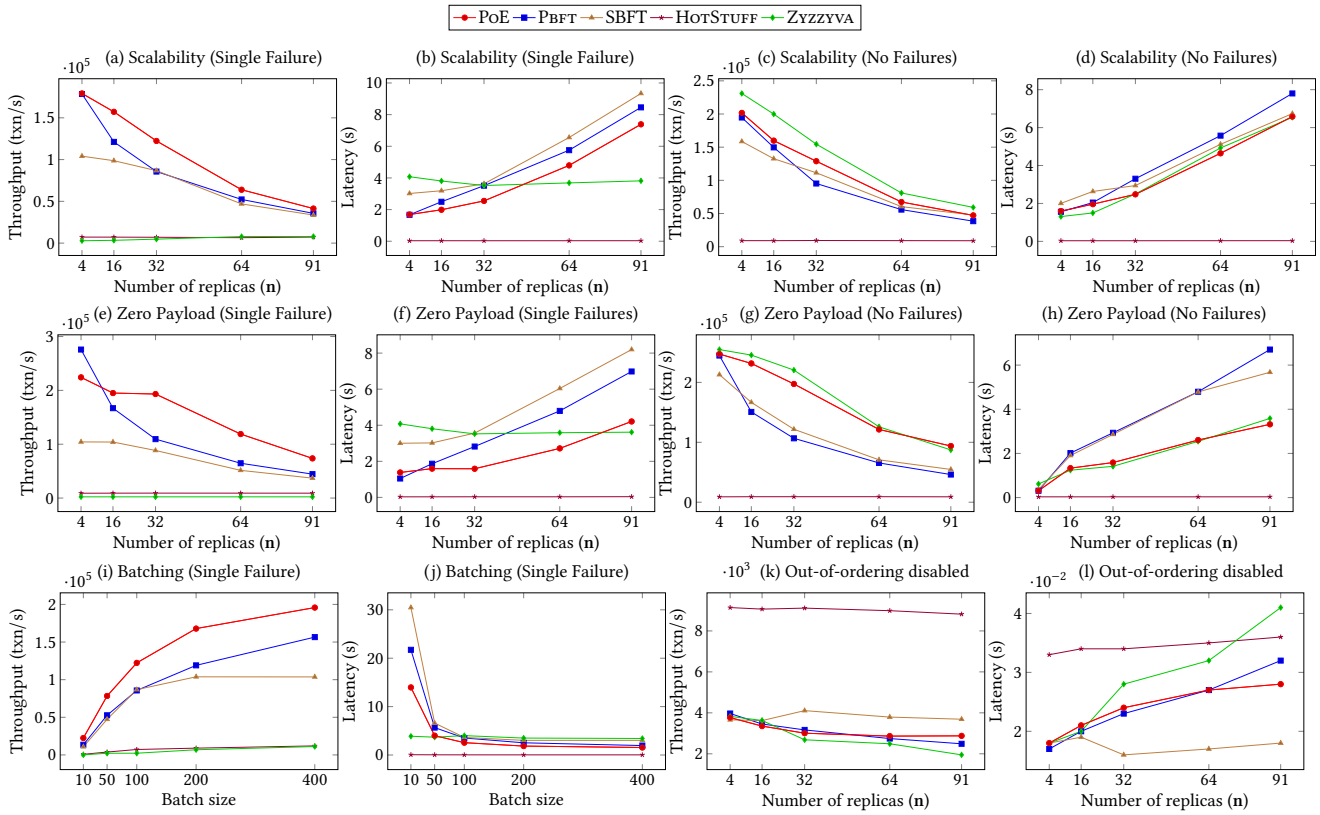


Figure 9: Evaluating system throughput and average latency incurred by PoE and other BFT protocols.

throughput, while decreasing the latency. This happens as larger batch-sizes require fewer consensus rounds to complete the exact same set of requests, reducing the cost of ordering and executing the transactions. This not only improves throughput, but also reduces client latencies as clients receive faster responses for their requests. Although increasing the batch-size reduces the number of consensus rounds, the large message size causes a proportional decrease in throughput (or increase in latency). This is evident from the experiments at higher batch-sizes: increasing the batch-size beyond 100 gradually curves the throughput plots towards a limit for PoE, Pbft and SBFT. For example, on increasing the batch size from 100 to 400, PoE and Pbft see an increase in throughput by 60% and 80%, respectively, while the gap in throughput reduces from 43% to 25%. As in the previous experiments, Zyzzyva yields a significantly lower throughput as it cannot handle failures. In case of HotStuff, an increase in batch size does increases its throughput but due to high scaling of the graph this change seems insignificant.

5.6 Disabling Out-of-Ordering

Until now, we allowed protocols like Pbft, PoE, SBFT and Zyzzyva to process requests *out-of-order*. As a result, these protocols achieve much higher throughputs than HotStuff, which is restricted by its sequential primary-rotation model. In Figures 9(k) and 9(l), we evaluate the performance of the protocols when there are no opportunities for out-of-ordering.

In this setting, we require each client to only send its request when it has accepted a response for its previous query. As HotStuff pipelines its phases of consensus into a *four*-phase pipeline, so we allow it to access four client requests (each on a distinct subsequent replica) at any time. As expected, HotStuff performs better than all other protocols at the expense of a higher latency as it rotates primaries at the end of each consensus, which allows

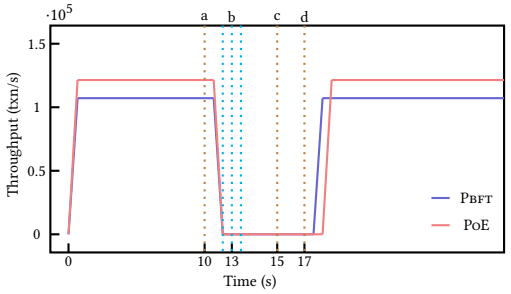


Figure 10: System throughput under instance failures (n = 32). (a) replicas detect failure of primary and broadcast VC-REQUEST; (b) replicas receives VC-REQUEST from others; (c) replicas receives NV-PROPOSE from new primary; (d) state recovery;

it to pipeline four requests. However, notice that once out-of-ordering is disabled, throughput drops from 200 ktransactions/s to just under a few *thousand* transactions/s. Hence, from a practical standpoint, out-of-ordering is simply crucial. Further, the difference in latency of different protocols is quite small, and the visible variation is a result of graph scaling while the actual numbers are in the range of 20 ms–40 ms.

5.7 Primary Failure–View Change

In Figure 10, we study the impact of of a benign primary failure on PoE and Pbft. To recover from a primary failure, backup replicas run the view-change protocol. We skip illustrating view-change plots for Zyzzyva and SBFT as they already face severe reduction in throughput for a single backup failure. Further, Zyzzyva has an *unsafe* view-change algorithm and SBFT's view-change algorithm is no less expensive than Pbft. For HotStuff, we do not

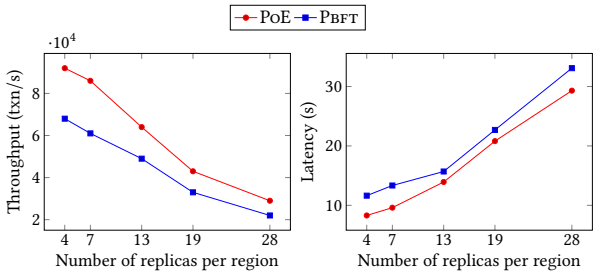


Figure 11: System throughput and average latency incurred by PoE and Pbft in a WAN deployment of five regions under a single failure. In the largest deployment, we have 140 replicas spread equally over these regions.

show results as it changes primary at the end of every consensus. Although single primary protocols face a momentary loss in throughput during view-change, these protocols easily cover this gap through their ability to process messages out-of-order.

For our experiments, we let the primary replica complete consensus for 10 s (or around a million transactions) and then fail. This causes clients to timeout while waiting for responses for their pending transactions. Hence, these clients forward their requests to backup replicas.

When a backup replica receives a client request, it forwards that request to the primary and waits on a timer. Once a replicas timeouts, it detects a primary failure and broadcasts a VC-REQUEST message to all other replicas—initiate view-change protocol (a). Next, each replica waits for a new view message from the next primary. In the meantime, a replica may receive VC-REQUEST messages from other replicas (b). Once a replica receives NV-PROPOSE message from the new primary (c), it moves to the next view.

5.8 WAN Scalability

In this section, we use Figure 11 to illustrate the throughputs and latencies for different PoE and Pbft deployments on a wide-area network in the presence of a single failure. In specific, we deploy clients and replicas across *five* locations across the globe: Oregon, Iowa, Montreal, the Netherlands, and Taiwan. Next, we vary the number of replicas from 20 to 140 by equally distributing these replicas across each region.

These plots affirm our existing observations that PoE outperforms existing state-of-the-art protocols and scales well in wide-area deployments. In specific, PoE achieves up to 1.41× higher throughput and incurs 28.67% less latency than Pbft. We skip presenting plots for SBFT, HotStuff and Zyzzyva due to their low throughputs under failures.

5.9 Simulating bft Protocols

To further underline that the *message delay* and not *bandwidth requirements* becomes a determining factor in the throughput of protocols in which the primary does not propose requests out-of-order, we performed a separate simulation of the maximum performance of PoE, Pbft, and HotStuff. The simulation makes 500 consensus decisions and processes all message send and receive steps, but delays the arrival of messages by a predetermined message delay. The simulation skips any expensive computations and, hence, the simulated performance is entirely determined by the cost of message exchanges. We ran the simulation with $\mathbf{n} \in \{4, 16, 128\}$ replicas, for which the results can be found in Figure 12, first three plots. As one can see, if bandwidth is not a limiting factor, then the performance of protocols that do not propose requests out-of-order will be determined by the number of communication rounds and the message delay. As both Pbft and PoE have one communication round more than the two rounds of HotStuff, their performance is roughly two-thirds that of HotStuff, this independent of the number of replicas or the message delay. Furthermore, doubling message delay will roughly half performance. Finally, we also measured the maximum performance of protocols that do allow out-of-order processing of up to 250 consensus decisions. These results can be found in Figure 12, last plot. As these results show, out-of-order processing increases performance by a factor of roughly 200, even with 128 replicas.

6 RELATED WORK

Consensus is an age-old problem that received much theoretical and practical attention (see, e.g., [34, 39, 45]). Further, the use of rollbacks is common in distributed systems. E.g., the crash-resilient replication protocol Raft [45] allows primaries to rewrite the log of any replica. In a Byzantine environment, such an approach would delegate too much power to the primary, as they can maliciously overwrite transactions that need to be preserved.

The interest in practical bft consensus protocols took off with the introduction of Pbft [9]. Apart from the protocols that we already discussed, there are some interesting protocols that achieve efficient consensus by requiring $5\mathbf{f} + 1$ replicas [1, 14]. However, these protocols have been shown to work only in the cases where transactions are non-conflicting [38]. Some other bft protocols [10, 50] suggest the use of *trusted components* to reduce the cost of bft consensus. These works require only $2\mathbf{f}+1$ replicas as the trusted component helps to guarantee a correct ordering. The safety of these protocols relies on the security of trusted component. In comparison, PoE does (i) not require extra replicas, (ii) not depend on clients, (iii) not require trusted components, and (iv) not need the two phases of quadratic communication required by Pbft.

As a promising future direction, Castro [9] also suggested exploring speculative optimizations for Pbft, which he referred to as tentative execution. However, this lacked: (i) formal description, (ii) non-divergence safety property, (iii) specification of rollback under attacks, (iv) re-examination of the view change protocol, and (v) any actual evaluation.

Consensus for Blockchains: Since the introduction of Bitcoin [42], the well-known cryptocurrency that led to the coining of the term blockchain, several new bft consensus protocols that cater to cryptocurrencies have been designed [33, 37]. Bitcoin [42] employs the *Proof-of-Work* [33] consensus protocol (PoW), which is computationally intensive, achieves low throughput, and can cause forks (divergence) in the blockchain: separate chains can exist on non-faulty replicas, which in turn can cause *double-spending attacks* [31]. Due to these limitations, several other similar algorithms have been proposed. E.g., *Proof-of-Stake* (PoS) [37], which is design such that any replica owning $n\%$ of the total resources gets the opportunity to create $n\%$ of the new blocks. As PoS is resource driven, it can face attacks where replicas are incentivized to work simultaneously on several forks of the blokchain, without ever trying to eliminate these forks.

There are also a set of interesting alternative designs such as ConFlux [40], Caper [3] and MeshCash [6] that suggest the use of directed acyclic graphs (DAGs) to store a blockchain to improve the performance of Bitcoin. However, these protocols either rely on PoW or Pbft for consensus.

Meta-protocols such as RCC [28] and RBFT [5] run multiple Pbft consensuses in parallel. These protocols also aim at removing dependence on the consensus led by a single primary. A recent protocol, PoV [41], provides fast bft consensus in a

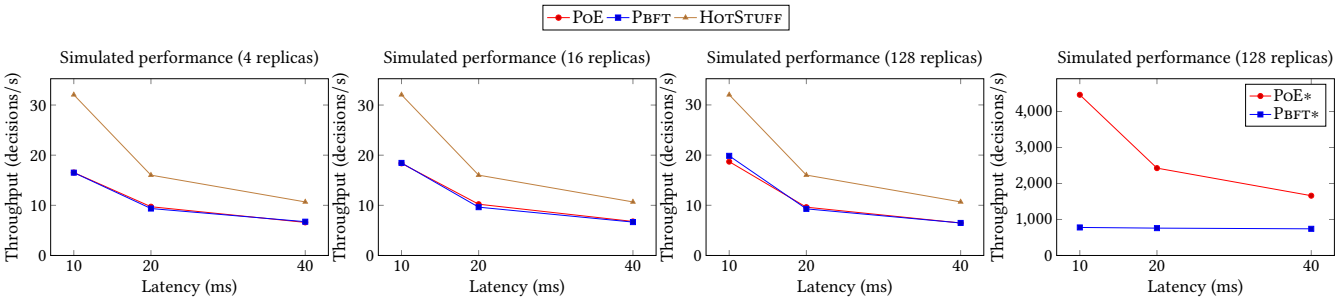


Figure 12: The simulated number of consensus decisions PoE, Pbft, and HotStuff can make as a function of the latency. Only the protocols in the right-most plot and marked with $*$ process requests out-of-order processing.

consortium architecture. PoV does this by restricting the ability to propose blocks among a subset of trusted replicas.

PoE does not face the limitations faced by PoW [33] and PoS [37]. The use of DAGs [3, 6, 40], and sharding [15, 52] is orthogonal to the design of PoE. Hence, their use with PoE can reap further benefits. Further, PoE can be employed by meta-protocols and does not restrict consensus to any subset of replicas.

7 CONCLUSIONS

We present Proof-of-Execution (PoE), a novel Byzantine fault-tolerant consensus protocol that guarantees safety and liveness and does so in only three linear phases. PoE decouples ordering from execution by allowing replicas to process messages out-of-order and execute client-transactions speculatively. Despite these properties, PoE ensures that all the replicas reach a single unique order for all the transactions. Further, PoE guarantees that if a client observes identical results of execution from a majority of the replicas, then it can reliably mark its transaction committed. Due to speculative execution, PoE may require replicas to revert executed transactions, however. To evaluate PoE's design, we implement it in our ResilientDB fabric. Our evaluation shows that PoE achieves up-to-80% higher throughputs than existing bft protocols in the presence of failures.

REFERENCES

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-scalable Byzantine Fault-tolerant Services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. ACM, 59–74. https://doi.org/10.1145/1095810.1095817

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. https://arxiv.org/abs/1712.01367

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: A Cross-application Permissioned Blockchain. *Proc. VLDB Endow.* 12, 11 (2019), 1385–1398. https://doi.org/10.14778/3342263.3342275

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30:1–30:15. https://doi.org/10.1145/3190508.3190538

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

[6] Iddo Bentov, Pavel Hubáček, Tal Moran, and Asaf Nadler. 2017. Tortoise and Hares Consensus: the Meshcash Framework for Incentive-Compatible, Scalable Cryptocurrencies. https://eprint.iacr.org/2017/300

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

[8] Manuel Bravo, Zsolt István, and Man-Kit Sit. 2020. Towards Improving the Performance of BFT Consensus For Future Permissioned Blockchains. https://arxiv.org/abs/2007.12637

[9] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

[10] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-Only Memory: Making Adversaries Stick to Their Word. *SIGOPS Oper. Syst. Rev.* 41, 6 (2007), 189–204. https://doi.org/10.1145/1323293.1294280

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

[12] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 153–168.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[14] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX, 177–190.

[15] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 123–140. https://doi.org/10.1145/3299869.3319889

[16] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1085–1100. https://doi.org/10.1145/3035918.3064033

[17] Wayne W. Eckerson. 2002. *Data quality and the bottom line: Achieving Business Success through a Commitment to High Quality Data*. Technical Report. The Data Warehousing Institute, 101communications LLC.

[18] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A Shared Database on Blockchains. *Proc. VLDB Endow.* 12, 11 (2019), 1597–1609. https://doi.org/10.14778/3342263.3342636

[19] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

[20] Gideon Greenspan. 2015. MultiChain Private Blockchain–White Paper. https://www.multichain.com/download/MultiChain-White-Paper.pdf

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

[22] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, 393–481. https://doi.org/10.1007/3-540-08755-9_9

[23] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2019. An In-Depth Look of BFT Consensus in Blockchain: Challenges and Opportunities. In *Proceedings of the 20th International Middleware Conference Tutorials, Middleware*. ACM, 6–10. https://doi.org/10.1145/3366625.3369437

[24] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Blockchain consensus unraveled: virtues and limitations. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. ACM, 218–221. https://doi.org/10.1145/3401025.3404099

[25] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Building High Throughput Permissioned Blockchain Fabrics: Challenges and Opportunities. *Proc. VLDB Endow.* 13, 12 (2020), 3441–3444. https://doi.org/10.14778/3415478.3415565

[26] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2019. Brief Announcement: Revisiting Consensus Protocols through Wait-Free Parallelization. In *33rd International Symposium on Distributed Computing (DISC 2019)*, Vol. 146. Schloss Dagstuhl, 44:1–44:3. https://doi.org/10.4230/LIPIcs.DISC.2019.44

[27] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. *Fault-Tolerant Distributed Transactions on Blockchain*. Morgan & Claypool. https://doi.org/10.2200/S01068ED1V01Y202012DTM065

[28] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *37th IEEE International Conference on Data Engineering*. IEEE. to appear.

[29] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (2020), 868–883. https://doi.org/10.14778/3380750.3380757

[30] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned. In *Proceedings of the 40th IEEE International Conference on Distributed Computing Systems*.

[31] Suyash Gupta and Mohammad Sadoghi. 2019. Blockchain Transaction Processing. In *Encyclopedia of Big Data Technologies*. Springer, 1–11. https://doi.org/10.1007/978-3-319-63962-8_333-1

[32] Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. 2007. *Data Quality and Record Linkage Techniques*. Springer. https://doi.org/10.1007/0-387-69505-2

[33] Markus Jakobsson and Ari Juels. 1999. Proofs of Work and Bread Pudding Protocols. In *Secure Information Networks: Communications and Multimedia Security IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99)*. Springer, 258–272. https://doi.org/10.1007/978-0-387-35568-9_18

[34] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*. IEEE, 245–256. https://doi.org/10.1109/DSN.2011.5958223

[35] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. USENIX, 237–250.

[36] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography* (2nd ed.). Chapman and Hall/CRC.

[37] Sunny King and Scott Nadal. 2012. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. https://www.peercoin.net/whitepapers/peercoin-paper.pdf

[38] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.* 27, 4 (2009), 7:1–7:39. https://doi.org/10.1145/1658357.1658358

[39] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (2001), 51–58. https://doi.org/10.1145/568425.568433 Distributed Computing Column 5.

[40] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. 2018. Scaling Nakamoto Consensus to Thousands of Transactions per Second. https://arxiv.org/abs/1805.03870

[41] Kejiao Li, Hui Li, Han Wang, Huiyao An, Ping Lu, Peng Yi, and Fusheng Zhu. 2020. PoV: An Efficient Voting-Based Consensus Algorithm for Consortium Blockchains. *Front. Blockchain* 3 (2020), 11. https://doi.org/10.3389/fbloc.2020.00011

[42] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

[43] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A Global-Scale Byzantizing Middleware. In *35th International Conference on Data Engineering (ICDE)*. IEEE, 124–135. https://doi.org/10.1109/ICDE.2019.00020

[44] The Council of Economic Advisers. 2018. *The Cost of Malicious Cyber Activity to the U.S. Economy*. Technical Report. Executive Office of the President of the United States. https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf

[45] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX, 305–320.

[46] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems*. Springer. https://doi.org/10.1007/978-3-030-26253-2

[47] Sajjad Rahnama, Suyash Gupta, Thamir Qadah, Jelle Hellings, and Mohammad Sadoghi. 2020. Scalable, Resilient and Configurable Permissioned Blockchain Fabric. *Proc. VLDB Endow.* 13, 12 (2020), 2893–2896. https://doi.org/10.14778/3415478.3415502

[48] Thomas C. Redman. 1998. The Impact of Poor Data Quality on the Typical Enterprise. *Commun. ACM* 41, 2 (1998), 79–82. https://doi.org/10.1145/269012.269025

[49] Dale Skeen. 1982. *A Quorum-Based Commit Protocol*. Technical Report. Cornell University.

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

[51] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 347–356. https://doi.org/10.1145/3293611.3331591

[52] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling Blockchain via Full Sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 931–948. https://doi.org/10.1145/3243734.3243853