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  
Davis, CA 95616-8562, USA

ABSTRACT
Since the introduction of blockchains, several new database systems and applications have tried to employ them. At the core of such blockchain designs are Byzantine Fault-Tolerant (BFT) consensus protocols that enable designing systems that are resilient to failures and malicious behavior. Unfortunately, existing BFT protocols seem unsuitable for usage in database systems due to their high computational costs, high communication costs, high client latencies, and/or reliance on trusted components and 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 still providing reliable consensus toward clients in all cases.

We envision the use of PoE in high-performance resilient database systems. To validate this vision, we implement PoE in our efficient ResilientDB blockchain and database framework. ResilientDB helps us to implement and evaluate PoE against several state-of-the-art BFT protocols. Our evaluation shows that PoE achieves up to 86% more throughput than existing BFT protocols and recover from failures, a consensus protocol helps to provide continuous service, even during malicious attacks.

To achieve consensus among their replicas, several of the recent blockchain databases and applications [3, 35, 40, 78, 79] still depend on the classical Pbft protocol [15], even though, Pbft requires three phases of communication, of which two necessitate quadratic communication. This dependence is partly due to time-tested safe design of Pbft and due to the limiting designs of other BFT protocols. Indeed, even when there are no failures, current BFT consensus protocols such as Zyzzyva, Pbft, and HotStuff [15, 16, 60, 61, 111] incur high client latencies due to the many phases of communication (Pbft, HotStuff, HotStuff-ts), require cryptographic primitives with high computational costs (Zyzzyva, HotStuff, HotStuff-ts), require a large amount of communication (Pbft, HotStuff), cannot handle any failure and require reliable clients (Zyzzyva), or require threshold signatures for which no tested implementation exists (HotStuff-ts).

In this paper, we introduce the Proof-of-Execution consensus protocol (PoE) that achieves low latency and high throughput with moderate communication costs and without depending on clients or expensive cryptographic primitives. A full comparison of the normal-case operations of PoE and other popular BFT protocols can be found in Figure 1.

At the core of PoE is an algorithm that allows replicas to process messages out-of-order and facilitates speculative execution of client transactions. Via the use of out-of-order processing, PoE decouples ordering from execution and eliminates various bottlenecks associated with sequential consensus protocols. With the help of speculative execution, a replica can execute transaction before reaching a guarantee on the agreement, thus, reducing perceived client latencies and communication. If the primary is non-faulty, then PoE ensures an inexpensive and reliable consensus. If the primary is malicious, speculative execution can require a replica to revert any speculatively-executed transactions. The design of PoE still guarantees that all executions detectable by clients are preserved, this by using a special-purpose view-change protocol. Hence, a malicious primary can never prevent replicas from reaching a consistent state.

In specific, we make the following contributions:

1. We introduce the PoE consensus protocol, a novel BFT protocol that uses speculative execution and allows out-of-order processing without relying on clients.
2. We prove that PoE guarantees safety: it replicates a unique sequence of order-decisions among all replicas and ensures that the clients can reliably detect when their transactions
are executed, this independent of any malicious behavior by any of the replicas.

(3) We show that PoE guarantees liveness: whenever the network provides reliable communication, PoE continues successful operation, this independent of any malicious behavior by any of the replicas.

(4) To guarantee safety and liveness, we introduce a safe view-change protocol to deal with malicious primaries in the presence of speculative execution.

(5) To validate our vision of using PoE in blockchain databases, we implement PoE in our efficient ResilientDB framework.\footnote{ResilientDB is a spin-off of ExpoDB [48, 89, 92], our exploratory data platform, aiming at high-performance fault-tolerant data processing.}

We also implemented other state-of-the-art BFT protocols in ResilientDB (Zyzzyva, PbFT, and HotStuff), and evaluated PoE against these BFT protocols using the YCSB benchmark \[23\]. Our experiments focus on the throughput and latency of these protocols in the fault-free case and in case of failures (single replica failure, multiple replica failures, primary failure). We show that PoE achieves up to 86% more throughput than existing BFT protocols.

### 2 BACKGROUND AND APPLICATIONS

The interests in blockchains emerged after the rise of Bitcoin—the first blockchain-based cryptocurrency \[77\]. Bitcoin and its peers promoted an environment of open-membership, where any node can act as a replica and can participate in consensus \[47, 107\]. To do so, these applications employed the Proof-of-Work (PoW) \[52\] consensus algorithm, which has been shown to yield low throughput, high client latencies, consume much energy, and suffer from critical attacks \[27, 86, 102, 105\]. This led to the rise of permissioned blockchains, where only a select group of nodes can act as replicas and participate in the consensus \[29, 47\]. These permissioned blockchains operate in a more secure setting, which allows them to provide higher throughput at lower cost. This enables permissioned blockchains as a resolve for challenges in food production \[39\], energy production and energy trading \[88\], and managing health care \[9, 43, 54\].

Existing blockchain databases and data management applications fall under the category of permissioned blockchain systems \[3, 35, 40, 78\]. Like any other database system, a blockchain database system requires both availability and consistency. To achieve consistency, they employ BFT consensus protocols. To be able to deal with big workloads without making clients wait, they need availability with high throughput. In Figure 1, we illustrate how state-of-the-art BFT protocols fall short in achieving these requirements. Hence, the design of our PoE protocol is in order.

We also examined recent blockchain papers in the database community to analyze what is the right size of a permissioned blockchain system \[3, 29, 35, 78, 91, 94, 109\]. In our study, we found that almost all the works employ up to 16 replicas, and only a couple of works used up to 32 replicas. Further, we saw that some of these works used sharding to scale their applications or database \[26, 35\]. Sharding is an interesting data-partitioning technique, but it is very expensive when the transactions require access to multiple partitions. With such multi-partition transactions, the throughput can be as low as 2000 txns/s \[26\]. Another key observations we had from this study was that a majority of these works employed consensus based on PbFT protocol and none of these works crossed a throughput of 20K txns/s. These systems can greatly benefit from our PoE consensus protocol, as our evaluation (Section 5) shows that PoE can reach a throughput of 110K txns/s on 16 replicas and 70K txns/s on 32 replicas.

### 3 PROOF-OF-EXECUTION

PoE follows the standard primary-backup design \[82\], where one replica is elected the primary and the other replicas act as backups. The primary is responsible for proposing client transactions and the other replicas act as backups.

The primary is responsible for proposing client transactions, that need to be executed, to all the backups. In PoE, each backup replica speculatively executes the client transactions with the hope that primary is behaving correctly. In the normal case, speculative execution helps to expedite execution of transactions, thereby reducing latencies for clients and overall communication in comparison to non-speculative designs (e.g., PbFT). When any malicious behavior is detected, the backups can recover by reverting transactions. Independent of malicious behavior, PoE ensures that all non-faulty replicas eventually reach a unique common sequence of executed transactions (consensus).

PoE

### 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.

We model a system as a tuple $(\mathcal{R}, \mathcal{C})$, in which $\mathcal{R}$ is a set of replicas that process client requests and $\mathcal{C}$ is a set of clients making requests. We assign each replica $r \in \mathcal{R}$ a unique identifier $id(r)$ with $0 \leq id(r) < |\mathcal{R}|$. We write $\mathcal{F} \subseteq \mathcal{R}$ to denote the set of Byzantine replicas that can behave in arbitrary, possibly coordinated and malicious, manners. We use notation $N\mathcal{F} = \mathcal{R} \setminus \mathcal{F}$ to denote the set of non-faulty replicas in $\mathcal{R}$. We assume that the non-faulty replicas 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 the clients: all client can be malicious without affecting PoE.

We write $n = |\mathcal{R}|$, $f = |\mathcal{F}|$, and $nf = |N\mathcal{F}|$ to denote the number of replicas, faulty replicas, and non-faulty replicas, respectively. We assume that $nf > 2f$, which is a minimal requirement to deal with Byzantine behavior when not relying on digital signatures \[30, 31\]. We also employ a collision-resistant cryptographic hash function $D(\cdot)$ that can map an arbitrary value $v$ to a constant-sized digest.

![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 with a good primary. PoE provides lowest latency (number of phases) with high resilience, while having low communication and computational costs.](image-url)
D(u) [56, 75]. We assume that it is practically impossible to find another value u’, v ≠ u’, such that D(u) = D(u’).

We assume authenticated communication: byzantine replicas are able to impersonate each other, but no replica can impersonate a non-byzantine replica. Hence, on receipt of a message m from replica r ∈ R, one can determine that r did send m if r ∈ F; and one can only determine that m was sent by a non-faulty replica if r ∈ NF. Authenticated communication is a minimal requirement to deal with Byzantine behavior, as otherwise faulty replicas can impersonate all non-faulty replicas. We use message authentication codes [56, 75] to achieve authenticated communication.

Next, we define the consensus protocol provided by PoE:

**Definition 3.1.** Let (R, Σ) be a system. A single run of any consensus protocol should satisfy the following requirements:

**Termination.** Each non-faulty replica in the set NF executes transactions.

**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 revert transactions. To provide safety, PoE provides speculative non-divergence instead of non-divergence:

**Speculative non-divergence.** If f + 1 non-faulty replicas (a majority) accept and execute the same transaction T, then all the non-faulty replicas will revert any other accepted and executed transactions, and will eventually accept and execute T.

To provide safety, we do not need any other assumptions on communication or on the behavior of clients. Due to well-known impossibility results for asynchronous consensus [11, 12, 38, 42], 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 Normal-case algorithm of PoE

The PoE protocol operates in views v = 0, 1, . . . . In view v, the replica r with id(r) = v mod n is elected as the primary. It is the job of the primary to coordinate replication and execution of client transactions. As the first step of this process, clients send transactions to the primary.

Consider a view v with primary p and a client c that wants execution of a transaction T. In PoE, the client c initiates execution by sending T to p. To assure that malicious primaries cannot forge transactions, the client signs T with its private key. We denote such signed messages by \( (T)_c \). To initiate replication and execution of T as the k-th transaction, the primary proposes T to all replicas by broadcasting the message propose\( ((T)_c, v, k) \).

After a replica r receives a propose message from p, it checks whether at least nf ≥ 2f + 1 other replicas also received the same proposal from p. This check assures r that at least nf - f ≥ f + 1 non-faulty replicas received the same proposal, which helps to achieve speculative non-divergence. To perform this check, each replica agrees to support the first proposal propose\( (T)_c, v, k \) it receives from the primary by broadcasting a message support\( D((T)_c, v, k) \) to all replicas. The replica r also logs this support decision as Support\( t_q((T)_c, k, v) \).

After this broadcast, each replica waits until it receives support messages, identical to the message it sent, from nf distinct replicas. If r receives these messages, it view-commits to T as the k-th transaction in view v. The replica \( \mathcal{N} \) logs this view-commit decision as VCommit\( t_q((T)_c, k, v) \).

After \( \mathcal{R} \) view-commits to T, \( \mathcal{R} \) schedules T for speculative execution as the k-th transaction of view v. Consequently, T will be executed by \( \mathcal{R} \) after all preceding transactions are executed. We write \( \text{Execute}_q((T)_c, k, v) \) to denote this execution. After execution, \( \mathcal{R} \) informs the client of the order of execution and of any execution result r via a message inform\( (D((T)_c), v, k, r) \).

A client considers its transaction successfully executed after it receives identical inform messages from nf distinct replicas. This guarantees that at least nf - f ≥ f + 1 non-faulty replicas executed this transaction as the k-th transaction. Hence, PoE’s speculative non-divergence guarantees that eventually all non-faulty replicas will accept and execute T as the k-th transaction.

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

The communication of the normal-case algorithm of PoE is sketched in Figure 2 and the full pseudo-code of the algorithm can be found in Figure 3.

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

**Proposition 3.2.** Let \( r_i, i \in \{1, 2\} \), be two non-faulty replicas that view-committed to \( (T)_c \) as the k-th transaction of view v \( \text{VCommit} _q((T)_c, k, v) \). We have \( (T)_c = (T)_c \).

**Proof.** Replica \( r_1 \) only view-committed to \( (T)_c \) after \( r_1 \) received identical messages support\( D((T)_c, v, k) \) from a set \( S_1 \) of nf distinct replicas (Line 14 of Figure 3). Let \( X_1 = S_1 \setminus F \) be the non-faulty replicas in \( S_1 \). We have \( |S_1| = nf \) and \( |F| ≤ f \). Hence, \( |X_1| ≥ nf - f \). The non-faulty replicas in \( T_1 \) will only send a single
Client-role (used by client $c$ to request transaction $T$):
1. Send $(T)_c$ to the primary $v$.
2. Await receipt of messages $\text{INFORM}(T)_c, (T)_c, (k, r)$ from $nf$ replicas.
3. Consider $T$ executed, with result $r$, as the $k$-th transaction.

Primary-role (running at the primary $p$ of view $\nu$, id($p$) = $\nu$ mod $n$):
4. Let view $\nu$ start after execution of the $k$-th transaction.
5. While $\nu$ is the primary do
6. $p$ awaits receipt of message $(T)_p$ from client $c$.
7. Broadcast $\text{PROPOSE}(T)_p, (\nu, k)$ to all replicas.
8. $k := k + 1$.
9. End while

Backup-role (running at every replica $r$ in $R$):
10. Event $e$ receives message $m := \text{PROPOSE}(T)_c, (\nu, k)$ such that:
11. $\nu$ is the current view;
12. $m$ is sent by the primary of $\nu$; and
13. $n$ did not accept a $k$-th proposal in $\nu$.
   do
14. Support $T$, the $k$-th transaction of $\nu$ (Support $t_k((T)_c, k, \nu))$.
15. Broadcast $\text{INFORM}(D((T)_c), (\nu, k, r))$ to all replicas.
16. End event
17. Event $e$ receives $nf$ messages $\text{INFORM}(T)_c, (\nu, k)$ such that:
18. Each message was sent by a distinct replica, and
19. $n$ did log Support $t_k((T)_c, k, \nu)$.
   do
20. View-commit $T$, the $k$-th transaction of $\nu$ (View-commit $t_k((T)_c, k, \nu))$.
21. End event

Figure 3: The normal-case algorithm of PoE.

support message for the $k$-th transaction in view $\nu$ (Line 10 of Figure 3). If $((T)_1)_{c_1} \neq ((T)_2)_{c_2}$, then $X_1$ and $X_2$ must not overlap. Hence, $|X_1 \cup X_2| \geq 2(nf - f)$. As $n = nf + f$, this simplifies to $3f \geq n$, which contradicts our assumption that $n > 3f$. Hence, we conclude $(T)_1 = (T)_2$.

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

**Theorem 3.3.** Consider a system in view $\nu$, 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 $(T)_c$, then the primary can use the normal-case algorithm of PoE (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.** Following the normal-case algorithm of PoE (Figure 3), the primary sends $\text{PROPOSE}(T)_c, (k, r)$ to all replicas (Line 7). In response, all $nf$ non-faulty replicas will broadcast $\text{INFORM}(D((T)_c), (\nu, k, r))$ (Line 12). As there are at most $f < nf$ faulty replicas, the faulty replicas can only force up to $f$ invalid support messages. Consequently, each non-faulty replica will only receive the message $\text{INFORM}(D((T)_c), (\nu, k))$ from at least $nf$ distinct replicas and every non-faulty replica will view-commit $T$ (Line 14). 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 $r$ across all non-faulty replicas. Hence, when the non-faulty replicas inform $c$, they do so by all sending identical messages $\text{INFORM}(D((T)_c), (\nu, k, r))$ to $c$ (Line 17–Line 20). As all $nf$ non-faulty replicas executed $T$, we have non-divergent execution. Finally, as there are at most $f < nf$ faulty replicas, the faulty replicas can only forge up to $f$ invalid inform messages. Consequently, the client $c$ will only receive the message $\text{INFORM}(D((T)_c), (\nu, k, r))$ from at least $nf$ distinct replicas, and will conclude that $T$ is executed yielding result $r$ (Line 3).

Next, we look at how PoE deals with failures. Theorem 3.3 shows that PoE provides consensus under normal-case behavior. If the normal-case behavior of PoE is interrupted, then from Theorem 3.3 we know that there are only two possible causes: faulty primary and unreliable communication.

Example 3.4. A malicious primary can attempt to affect PoE by not conforming to the normal-case algorithm in several ways, e.g.:
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 $nf - f$ non-faulty replicas receive proposals: the faulty replicas in $F$ 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.

We notice that 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 of the second and third case.

If communication is unreliable, then there is no way to guarantee continuous service. Hence, replicas assume failure of the current primary if the normal-case behavior of PoE is interrupted. To deal with such a failure, replicas will replace the primary. The design of PoE is such that the primary replacement during periods of unreliable communication does not affect the correctness of the protocol. The replacement of primaries is done via the view-change algorithm, which we introduce next.

3.3 The view-change algorithm
If PoE observes failure of the primary $p$ of view $\nu$, then PoE will elect the replica $p'$ with id($p'$) = $\nu + 1$ mod $n$ as the new primary, and move to the next view $\nu + 1$ via the view-change algorithm. This algorithm consists of five steps.

First, failure of the current primary needs to be detected by all non-faulty replicas. Second, all replicas exchange information to establish which transactions were included in view $\nu$ 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). Fourth, all the replicas have executed the same view; otherwise, replicas detect failure of $v'^2$ 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 full 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). PoE requires that each replica $r$ detects the failure of primary in the following two ways:

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 $v$ failed from $f + 1$ distinct replicas. As at most $f$ of these messages can come from faulty replicas, at least one other non-faulty replica $q$ must have detected a failure. In this case, $r$ joins the view-change (Figure 5, Line 6).

3.3.2 Exchange of view-change acknowledgements. To start proposing, supporting, and executing transactions in the next view, the non-faulty replicas must know which transactions are already executed. For this purpose, each view-change message $vc-request(v, E)$ contains a summary $E$ of all transactions executed by the sending replica (Figure 5, Line 1). However, this information is not sufficient. First, it is possible that not all the replicas have executed the same transactions as a faulty primary can leave some replicas in the dark (see Example 3.4, Case 2). Second, faulty replicas can include transactions in their summaries that have not been executed at all. To alleviate this issue, each replica $r$ will acknowledge each view-change message it received from other replicas if and only if this message is consistent with the internal state of $r$. In specific, if $r$ receives $vc-request(v, E)$ from $q$, then it checks whether each

![Figure 4: The current primary $r$ of view $v$ is faulty and needs to be replaced. The next primary, $r'$, and the replica $r_2$ detected this failure first and request view-change via $vc-request$ messages. The replica $r_1$ joins these requests. Next, all replicas validate these $vc-request$ messages and broadcast $vc-ack$ messages for all the requests they deem valid. Then, the new primary proposes a new view based on the information it received by broadcasting a $nv-propose$ message. If the received new-view proposal is valid, then the replicas acknowledge this proposal using $nv-ack$ messages. Finally, if a replica receives sufficient new-view acknowledgements, then it enters the new view with $v'$ as the new primary.](image)

![Figure 5: The view-change algorithm of PoE.](image)
Consider a view-commit decision $\text{VComm}_t(T)_c(k,w)$ of replica $r$. This replica requires that every entry $(w', k, t) \in E$ that has the same order $k$ must match to its view-commit decision. This is done by checking whether $w = w'$ and $t = (T)_c$ (Figure 5, Line 9). Furthermore, $\mathfrak{N}$ verifies whether $E$ contains a full sequence of transactions. If all these checks hold, then $r$ acknowledges the received view-change request by broadcasting $\text{VC-ACK}(t, c(q), D(\text{VC-REQUEST} (v, E)))$.

### 3.3.3 Proposing the new view

To start view $v + 1$, the new primary $p'$ (with id$(p') = (v + 1) \mod n$) needs to propose a new view by determining a valid final list of already-executed transactions. To do so, $p'$ waits until it receives sufficient information. In specific, $p'$ waits for a set $S = \mathfrak{N}, |S| \geq nf$, of distinct replicas from which $p'$ received: (i) view-change requests, and (ii) for each of these view-change requests, matching acknowledgments from each replica in the set $S$ (Figure 5, Line 12). Such a set $S$ is guaranteed to exist when the communication is reliable, as all non-faulty replicas will participate in the view-change algorithm.

Each replica $q_j \in S, 1 \leq j \leq nf$, would have included a summary $E_i$ in their view-change requests. Based on these summaries, $p'$ proposes a valid final state for view $v$ (and all previous views). This valid state consists of $E_1 \cup \ldots \cup E_{nf}$, which is the union of all transactions summarized in the view-change requests of replicas in $S$.

For validation purposes, the new primary $p'$ also includes a set $V$, which contains details of the replicas whose view-change requests were used to create the new-view proposal (Figure 5, Line 13). Finally, $p'$ makes this new-view proposal by broadcasting $\text{NV-PROPOSE}(v + 1, E_1 \cup \ldots \cup E_{nf}, V)$.

### 3.3.4 Validate the new view

After a replica $r$ receives a new-view proposal via message $m = \text{NV-PROPOSE}(v + 1, E', V)$ from the new primary $p'$ (Figure 5, Line 16), $r$ validates the content of this message. If $p'$ used the view-change request and acknowledgments sent by this replica in the construction of $m$, then $(id(p'), d) \in V$ and $r$ has all the view-change requests that $p'$ used to construct $m$. Hence, $r$ can fully validate $m$ by constructing a $p'$-propose message $m'$ out of the same information and validate whether $m' = m$. If $m = m'$, then $r$ acknowledges the new view. Otherwise, if $m \neq m'$, then $r$ detects failure of $p'$.

We notice that each replica will eventually receive all view-change requests and acknowledgements sent by non-faulty replicas and the new primary $p'$ must have used at least $nf - f \geq f + 1$ view-change requests of non-faulty replicas. Hence, if $p'$ did not use the view-change request of $r$ when constructing $m$, then $r'$ must have used at least $nf - f$ view-change requests that $r$ also received. In this case, $r$ can only partially validate whether $m$ is consistent with each of these $nf - f$ view-change requests of non-faulty replicas, which implies $W \cap B \neq \emptyset$. Hence, at least a single non-faulty replica in $B$, that executed $(T)_c$, was consulted when constructing $m$ and, hence, $(v, k, (T)_c) \in E'$. $\square$

Furthermore, the correctness of PoE also requires that every non-faulty replica that enters a new view, does so using the same new-view proposal. Using the same proof technique as in the proof of Proposition 3.2, we can prove that this requirement holds:

### Proposition 3.6

Let $k_i, i \in \{1, 2\}$, be two non-faulty replicas that entered a view $v'$ due to proposals $\text{NV-PROPOSE}(v, E'_1, V_1)$. We have $E'_1 = E'_2$ and $V_1 = V_2$.

### 3.4 Correctness of PoE

Having presented all essential parts of the PoE protocol, the normal-case algorithm of Figure 2 and the view-change algorithm of Figure 5, we are now ready to conclude that PoE provides speculative consensus (Definition 3.1).

**Theorem 3.7.** PoE is a speculative consensus protocol that always guarantees safety via speculative non-divergence and guarantees liveness in periods of reliable bounded-delay communication.
With these optimizations in place, in the normal-case algorithm will start with a timeout $\delta$. After these view-changes, Theorem 3.3 guarantees liveness. After these view-changes, Theorem 3.3 guarantees liveness.

### 3.5 Fine-tuning and optimizations

To keep presentation simple, we did not include all possible optimizations in the protocol description. We can reduce the number of messages sent and received by all replicas in the following ways:

1. All replicas omit sending messages to themselves.
2. The primary only sends propose messages, which all replicas also interpret as support messages. Hence, the receipt of $\text{PROPOSE}(T_c, v, k)$ from the primary also counts as receipt of $\text{SUPPORT}(T_v, v, k)$ from the primary.
3. Replicas do not send $vc$-ack messages for their own $vc$-request messages.
4. The new primary does not send $nv$-ack message for his $nv$-propose message.

With these optimizations in place, in the normal-case algorithm of PoE, each replica—including the primary—only sends a single message to all the other replicas for every proposed client request.

The view-change algorithm of Section 3.3 will send very large messages if many transactions have already been proposed. To limit the size of these messages, we utilize a standard periodic checkpoint protocol to exchange state information outside of the scope of the view-change algorithm (see, e.g., [14–16]). With these checkpoints in place, the view-change algorithm only needs to exchange information on transactions proposed after the last checkpoint.

### 4 RESILIENTDB FABRIC

Consensus protocols lie at the core of any blockchain application, including resilient databases, and their efficiency heavily impact the throughput of such applications. However, the underlying implementation for these consensus protocols also substantially affect throughput. Taking this into account, we have designed ResilientDB, a high-throughput permissioned blockchain fabric. Using ResilientDB, we wish 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 consensus through batching client transactions; and (iv) enable a secure and efficient ledger for use in resilient database systems.

1. **Multi-threading and Deep Pipelining.** ResilientDB envisions a client-server architecture, where clients send their transactions to servers for processing (see Figure 6). All the servers are a replica of each other, with one replica acting as the primary, while others perform the tasks of a backup. Clients and replicas employ the transport layer to communicate with each other. Replicas use the ResilientDB’s storage layer to maintain all the metadata and its local copy of ledger. It is at the execution layer where replicas run the PoE consensus protocol and execute the corresponding transactions. During this process, the replicas use the secure layer to access cryptographic constructs.

In ResilientDB, we also make a distinction while deciding on the best software architecture for a replica. Figures 7 and 8 illustrate the threads and pipelines at the primary and backup replicas, respectively. With each replica we associate multiple input and output threads. We designate one input-thread to receive client requests, while two other input-threads receive messages from other replicas. ResilientDB balances the message sending tasks between the two output-threads by assigning equal clients and replicas to each output-thread. This allows us to associate a distinct output-queue with each output-thread.

2. **Client Batching.** During our formal description, we assumed that a client sends one transaction and waits for its response before sending another request. This assumption is not practical (e.g., one client could have multiple transactions for transferring money to different accounts). Hence, ResilientDB allows each client $c$ to batch multiple transactions in one request message $(T)_c$ (where $T$ now represents a set of transactions) and to send such a batch to the primary replica.

3. **Modeling a Primary Replica.** At each replica, we reduce the cost of consensus by employing batching. When the primary replica receives a batch of transactions from the client, it treats it as a single request. The input-thread at the primary assigns a sequence number to each incoming client batch and enqueues it in the batch-queue. This batch-queue is accessed by multiple batching-threads, which compete among themselves to fetch the next available client batch. ResilientDB implements the batch-queue as a lock-free queue to avoid contention. Each batching-thread verifies if the dequeued client batch is well-formed. If this is the case, then the batching-thread constructs a propose message and puts this message in the output-queue. Each batching-thread also creates the digest for its client request.

A primary replica may also receive messages from backup replicas. When the input-thread receives a support message, it enqueues that message in the work-queue. The work-queue is accessed by a single worker-thread. The worker-thread dequeues a message and checks if it is well-formed. It continues collecting support messages corresponding to a propose message until the count reaches $n_f - f - 1$ and then informs the execute-thread to execute the client transactions.
A blockchain is an immutable ledger, where blocks are chained as a linked-list. An i-th block can be represented as:

$$B_i := \{ k, d, v, H(B_{i-1}) \}$$

Here, the block $B_i$ contains the sequence number ($k$) of the client request, the digest ($d$) of the request, the view number ($v$), and the hash of the previous block, $H(B_{i-1})$. Figure 9 presents a blockchain maintained by replicas. In ResilientDB, prior to any consensus, we require the first primary replica to create a genesis block [47].

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. These hashes can be computationally expensive. Hence, an alternative approach is to store the proof-of-accepting the k-th request in the k-th block. In PoE, such a proof consists of the n creates distinct replicas for this k-th request.

What is the use of these blocks? These blocks help a failed replica to recover its state. A recovering replica (either a failed replica or a replica left in the dark by a malicious primary) can ask other replicas for the blocks that have been added to the chain. Due to the structure of these blocks, the recovering replica can verify the validity of the received blocks, even if it only receives blocks from malicious replicas.

(7) Cryptographic Constructs: PoE also employs cryptographic constructs for mitigating attacks by Byzantine replicas. To do so, ResilientDB uses a combination of CMAC and AES [56] to provide authenticated communication between replicas. We require clients to sign their request messages using ED25519-based digital signatures, such that these requests can be forwarded by the primary, but cannot be forged. An alternate approach is to require clients to send their requests to all the replicas, but such an approach increases the message complexity of the system and requires non-faulty clients.

To create message digests and for hashing purposes, we use the SHA256 algorithm. We import implementations of all these constructs from the Crypto++ library [22].

(8) Other protocols: Apart from PoE, we also implement PBFT, Zyzzyva, and Hotstuff in our ResilientDB framework. PBFT is a three phase BFT consensus protocol. The first two phases for PBFT are identical to PoE, except that execution does not happen at the end of second phase. Once a worker-thread in PBFT gets $n - f - 1$ support messages, it sends a commit message to all the replicas (instead of informing the execute-thread). When an input-thread receives a commit message from other replicas, it places it in the work-queue. The worker-thread waits for $n$ commit messages and then informs the execute-thread to execute the client request. Hence, PBFT has an extra phase of $O(n^2)$ communication. The client waits for only $f + 1$ responses before marking the request complete.

Zyzzyva: provides an optimization for PBFT in case of no failures. Although Zyzzyva is unsafe [2], we implement it in ResilientDB to
study its performance with or without failures. When the worker-thread at a backup replica receives a client request from the primary, it directly informs the execute-thread. The execute-thread executes the request and replies to the client. Hence, replicas do not wait to confirm: (i) if other replicas also got the same request from the primary, and (ii) if the order for client request is same for all the replicas. This forces the client to wait for identical responses from all n replicas. In case the client does not receive n responses, it timeouts and sends a message to all the replicas, after which an expensive client-dependent view-change and recovery protocol is used to figure out the correct outcome. For this view-change protocol, it is required that all messages sent to clients are signed using expensive digital signatures.

**HotStuff**: As described earlier, **HotStuff** requires four phases of achieve consensus on a transaction. Furthermore, **HotStuff** avoids all-to-all replica communication by centralizing the primary as the point of contact. To do so, it splits each phase into two parts (a primary broadcast to all replicas, followed by all replicas responding to the primary). This creates a dependency of each replica on the primary and requires the primary to send proofs of previous phases. These proofs can be generated using expensive threshold signatures (which even the authors skip in their paper), or requires the primary to forward each replica’s message to every other replica. This also implies that **HotStuff** needs each of its replicas to sign every messages using digital signatures. Further, **HotStuff** necessitates change of primary at the end of every third phase, but we permit all of its replicas to act as primaries and work in parallel. In our implementation, we neither require **HotStuff** to use threshold signatures, nor require its primaries to forward each replica’s message to every other replica.

## 5 Evaluation

We now use our ResilientDB framework to evaluate our PoE protocol. We deploy ResilientDB on Amazon AWS and use up to 32 virtual machines of type c5.2xlarge for replicas. These machines have Intel Xeon Skylake SP CPUs with a turbo frequency of up to 3.4GHz. Each of these machines has 8-cores and 32GB memory. We deploy 200K clients on 10 4-core machines. Each clients send its next request once it receives responses for its previous request batch. We run each experiment for 180 seconds: the 60 seconds are warmup, and results are collected over the next 120 seconds. We average our results over three runs.

In each experiment, the workload is provided by the **Yahoo Cloud Serving Benchmark** (YCSB) [23]. Each client request queries a YCSB table with an active set of 600K records. For our evaluation, we use write queries, as the majority of typical blockchain requests are updates to existing data. Prior to the experiments, each replica is initialized with an identical copy of the YCSB table. The client requests generated by YCSB follow a uniform Zipfian distribution.

We evaluate PoE against three state-of-the-art BFT protocols, which are also part of popular blockchain applications: PBFT, Zyzzyva and HotStuff. In the following figures, we use ZYZ to refer to Zyzzyva and HS for HotStuff. The goal of our experiments is to answer the following questions:

(Q1) How does the system performance of PoE compares to the other protocols, under failure of a single replica?

(Q2) Is PoE’s performance affected by failure of f replicas? Does PoE retains its benefits under a primary failure?

(Q3) Does PoE benefits from batching client failures?

(Q4) How scalable is PoE on increasing the number of replicas participating in the consensus, in the normal-case?

### 5.1 Impact on Scalability under Failures

The first key question we ask is: **how scalable is PoE under primary and replica failures?** To answer this question, we measure performance as function of the number of replicas, which we vary between 4 and 32. For these experiments, we employ batching with 100 client requests per batch.

1. **Single Backup Failure**: We first run experiments in which not all n replicas participate in consensus. In specific, only 3f replicas will participate. Notice, when \( n = 4 \Rightarrow f = 1, n = 8 \Rightarrow f = 2, n = 16 \Rightarrow f = 5, \) and \( n = 32 \Rightarrow f = 10 \). Figures 10a and 10b show, for each consensus protocol, the throughput and average latency attained by the system. We also analyze the results for 99%ile latency, which helps in detecting the delay incurred by the top 1% of client requests.

   These graphs affirm our claim that PoE attains higher throughput and incurs lower latency than all the other protocols. PBFT has to perform an extra phase of \( O(n^2) \) communication, which reduces its throughput. HotStuff, on the other hand, has to complete eight phases (for each client request) and employs digital signatures. This causes a significant decrease in its throughput (even lower than PBFT when using at most 16 replicas). In case of Zyzzyva, a single failure causes the clients to wait for their timer to timeout (n identical responses are needed by a client to mark its request complete), which significantly lowers its throughput (similar limitations of Zyzzyva have been observed in other works, e.g. [20, 21]). Similar reasoning holds for the higher average and 99%ile latency incurred by PBFT and HotStuff in comparison to PoE. We skip plotting latencies for Zyzzyva, to better scale the graphs (as on average they are at least 4× more than any other protocol). For example, the average latency incurred by Zyzzyva at 4 replicas is 13.24s and at 32 replicas it is 13.88s.

   Another important observation is that the throughputs for both PoE and PBFT decrease when increasing the number of replicas, while HotStuff is less affected by increasing the number of replicas. HotStuff receives parallelism benefits, as each of its replicas also acts as a primary. Further, its \( O(n) \) communication cost causes it to send and process fewer message in comparison to both PoE and PBFT when increasing the number of replicas. In fact for 32 replicas, HotStuff outperforms PBFT as the high communication complexity of PBFT eliminates any benefits gained from the use of MACs or due to having fewer phases. However, as expected, HotStuff incurs significant 99%ile latency in comparison to other protocols. To summarize, PoE attains up to 40% more throughput (incurs 29% less latency) than PBFT and 130% more throughput (incurs 58% less latency) than HotStuff. When going from 16 to 32 replicas, PBFT has more severe throughput degradation (39%) in comparison to PoE (30%).

   Figure 10c also shows the total number of messages sent (or received) per second by the system. In our implementation, the
Figure 10: System throughput, latency and messages sent, as a function of the number of replicas used for consensus. We set the batch size to 100 and allow at most 3f replicas to be active at any time during the consensus.

Figure 11: System throughput as a function of the number of replicas used for consensus. We set the batch size to 100 and allow: (a) at most $n - f$ replicas to be active at any time during consensus, and (b) the primary replica to fail once.

Figure 12: System throughput and latency as a function of the number of requests per batch. We use 16 replicas for consensus and allow at most 3f replicas to be active at any time.

size of a support message is around 250 bytes, while the client request (batch) and client response is around 8 kilobytes. The trends in these graphs follow our preceding discussion. Due to linear message complexity, HotStuff has the least number of messages communicated, while PbFT has two rounds of $O(n^2)$ communication, and thus higher message complexity than PoE.

(2) f Backups Failures: We now analyze the throughput of the system when only nf replicas are active and all f faulty replicas have failed. This experiment also helps us to study the case when f of the byzantine replicas decide to ignore the requests from the primary. Figure 11a shows the throughput attained by different protocols. On performing a head-on comparison with previous figures, it can be seen that there is a reduction in the throughput attained by the
system for all the protocols. This happens because each replica spends more time to reach consensus for each request: each replica only needs at most \( n_f \) responses, which, in the previous experiment, could be served by the fastest of the 3\( f \) active replicas. Now, each replica has to wait for all responses, even those sent by the slowest replicas. However, the overall trends previously observed remain the same.

(3) **Primary Failure:** We now measure throughput attained by PoE and Pbft in case of a benign primary failure. For our experiments, we let the primary replica lead the consensus up to 900 transactions and then cause a primary failure. This causes clients to timeout and send requests to other replicas. These replicas forward the request to the primary and wait. Once the replicas timeout, they initiate the view-change protocol, which requires these replicas to aggregate the summary of all the transactions since the last checkpoint. In our experiments, we allow replicas to create checkpoints after every 600 requests. After the New-View message, the new primary manages the consensus as usual. The results are shown in Figure 11b. As is clear from the results, the whole view-change process reduces the system throughput for all protocols, as time is spent in reaching a stable new view. PoE still achieves higher throughput than Pbft (up to 49%), however. For HotSTUFF, we don’t show results, as it does not have a separate phase of primary replacement. For Zyzzyva, which already has very low throughput with non-primary failure, we skip its view-change.

5.2 **Impact of Batching under Failures**

The next key question we ask is: how much does PoE benefit from batching client requests when the system faces a replica failure? 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 300. For this experiment, we use a system with 16 available replicas, of which 15 participate in consensus (one replica has failed).

Figures 12a and 12b show, for each consensus protocol, the throughput and average latency attained by the system. For each protocol, increasing the batch-size also increases 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 to their requests. Although increasing the batch-size reduces the consensus rounds, the large message size causes a proportional decrease in throughput (or increase in latency). This is evident from experiments at higher batch-sizes: increasing the batch-size beyond 100 only has minimal impact on the performance for both PoE and Pbft. As in the previous experiments, Zyzzyva yields a significantly lower throughput, as it cannot handle failures. We again skip plotting latencies for Zyzzyva to better scale the graphs (they are, again, on average at least 4x more than any other protocol). To conclude, PoE attains up to 86% more throughput (incurs 44% less latency) than Pbft and 3.5\( x \) more throughput (incurs 74% less latency) than HotSTUFF.

5.3 **Scalability: Fault-Free**

Finally, we further look into the key question how scalable is PoE in the best case (when there are no failures)? To answer this question, we measure the system performance as a function of the number of replicas, which we vary between 4 and 32. For these experiments, we employ batching with 100 client requests per batch.

Figures 13a and 13b show, for each consensus protocol, the throughput and average latency attained by the system. These plots help us to bound the maximal throughput that can be attained by different consensus protocols in our system. First, in comparison to the scalability plots in Section 5.1, the throughputs for PoE, Pbft, and HotSTUFF are slightly higher than their throughputs under failures, which is to be expected. Second, PoE still continues to outperform both Pbft and HotSTUFF, for the reasons described earlier. Third, these plots clearly show that PoE is more scalable than Pbft, as on 32 replicas HotSTUFF outperforms Pbft in both throughput and latency. Finally, the results of Zyzzyva help us to determine an upper bound for the maximum throughput attained by the system. Unlike previous experiments, in the no-failure case Zyzzyva requires each replica to just execute the request as soon as they receive a propose message from the primary. Hence, it achieves a higher throughput in comparison to PoE. However, the difference in throughput for PoE and Zyzzyva is small, that is, PoE has 7% (on 4 replicas) to 18% (on 16 replicas) less throughput than Zyzzyva. An interesting observation is that on 32 replicas, Zyzzyva incurs higher latency than PoE, even though it has higher throughput. This happens as clients in PoE have to wait for only the fastest \( n_f = 22 \) replies, whereas a client for Zyzzyva has to wait for replies from all replicas (even the slowest ones). To conclude, PoE attains up to 47% more throughput (incurs 32% less latency) than Pbft and 2.5% more throughput (incurs 61% less latency) than HotSTUFF. This affirms our belief that PoE performs better than other protocols.

6 **RELATED WORK**

The approach of reverting decisions and redoing another transactions is a common problem in both database and distributed systems. Several concurrency control protocols [49] use this approach. In the domain of fault-tolerant consensus, Raft [81] also suggested that the log a replica holds can be over-written by the leader. However, Raft allows the leader to forcefully over-write logs of any replica. Although this approach works when handling crash-failures, such an approach affects safety when the leader can be malicious (byzantine).

Generally speaking, consensus is an age-old problem that received much theoretical and practical attention (see, e.g., [4, 8, 53, 64, 65, 69, 73, 87, 95, 103]). The interest in practical bft consensus protocols [10, 13, 71] took off with the introduction of Pbft [15, 17], the first practical well-performing consensus protocol. This led to several works [62, 63, 74, 108, 110] that try to improve the Pbft’s design. Some of the interesting works in this direction include quorum-based approaches [1, 18, 25, 68, 72, 96]. For instance, Q/U tries to attain a single phase bft consensus, but needs \( 5f + 1 \) replicas to handle just f byzantine failures and cannot handle concurrency. Its successor HQ [25], can handle concurrency if none of the transactions are conflicting. Cowling et al. [96] present an approach that
builds on top of HQ and handles conflicting operations, but requires an additional non-malicious entity, the preserializer.

**ZYZZYVA** introduced speculative execution to **BFT** consensus, this to achieve consensus in a single phase while only requiring $3f + 1$ replicas to handle $f$ byzantine failures. However, **ZYZZYVA**’s design requires no failures to achieve high throughput and requires reliable clients if the primary is malicious. Recent work [2] even showed that **ZYZZYVA** is unsafe. There have been other works [32, 51, 93] that employ **ZYZZYVA**’s model and face similar limitations.

Some other **BFT** protocols [6, 19, 24, 55, 67, 104] suggest the use of trusted components to reduce the cost of **BFT** consensus. These works require only $2f + 1$ replicas as the trusted component helps to guarantee a correct ordering. However, a trusted component can be compromised [84] and can act as a sink for different attacks.

In comparison to all these protocols, **PoE** does not: (i) require extra replicas, (ii) depend on clients, (iii) require trusted components, and (iv) does not need two phases of quadratic communication required by **Part**.

**Consensus for Blockchain:** Bitcoin [77], the well-known cryptocurrency that popularized blockchains, employs the **Proof-of-Work** [52] consensus algorithm (**PoW**). Due to the limitations of **PoW** (being computationally intensive, while having low throughput and high latencies [27, 85, 105]), several other similar algorithms have been suggested. Moreover, **PoW** can also cause forks (divergence) in the blockchain: separate chains can exist on non-faulty replicas, which in turn can cause double-spending attacks [47].

Several variants of **PoW** have been proposed such as **Proof-of-Stake** [58] (**PoS**) and **Proof-of-Space** [5, 34] (**PoC**). **PoS** states that any replica owning $n\%$ of the total resources gets $n\%$ times opportunity to create new blocks. As **PoS** is resource driven, it can face attacks where replicas are incentivized to work simultaneously on several forks of the blockchain, without ever trying to eliminate these forks. **PoC** is a space variant to **PoW**, that is, it expects replicas to solve computational problems that require massive space.

Several works [36, 46, 90, 112] improve the throughput of Bitcoin through interesting optimizations. **Bitcoin-NG** [36] suggests selecting a leader using **PoW** algorithm and then the leader is assigned the task of generating a block. **Byzcoin** [59], **Ouroboros** [57] and **Hybrid Consensus** [83] propose selecting a group of users that will run the consensus and create the next block. **Hybrid consensus** selects a group of replicas that will run **PoW**, while **Ouroboros** relies on **PoS**.

There are also a set of interesting alternative designs such as **Ghost** [99], **ConFlux** [66], **Spectre** [98] and **MeshCash** [7] that suggest the use of directed acyclic graphs (DAGs) to store a blockchain to improve the performance of Bitcoin. However, these protocols still rely on **PoW** for consensus. Works like **HoneyBadger** [76] and **Beat** [33] provide **BFT** consensus designs for fully asynchronous systems. **Honeybadger** has the same order of complexity as **PBFT** and has been shown to be even slower in some cases. **Beat** presents a suite of algorithms from which users can select according to their needs. Further, **Beat** applies techniques such as erasure codes and fingerprint checksums to **BFT** protocols, this to reduce message size and computational work. Note that most of the blockchain consensus designs that we have seen till now still have very low throughput, not more than 10K tx/s, and usually much lower.

**Elastico** [70], **RapidChain** [113], **Monoxide** [106] and **AHL** [26] suggest use of sharding instead of replication for maintaining the blockchain. **Elastico** requires the **PoW** algorithm to decide which replicas can process transactions, while **Monoxide** requires either **PoW** or **PoS** for achieving consensus on a block. **RapidChain** uses randomization to select a small set of processing replicas, improving performance, but it requires strong synchronous communication among its selected set of replicas, an unrealistic assumption. To further enable the design and implementation of complex sharded **BFT** systems, we recently proposed a specialized high-performance fault-tolerant communication primitive that enables efficient communication between shards [50]. **Algorand** [41] introduces probabilistic **BFT** algorithms and defines several high probability estimates for attaining fast consensus. However, in the worse case, **Algorand** can create forks and is susceptible to nothing-at-stake attack [113].

**PoE** does not face the limitations faced by **PoW** [52], **PoS** [58] and **PoC** [34], for which several attacks have been shown [28, 37, 101]. The use of DAGs [7, 66, 98, 99], erasure codes and checksums [33, 76], and sharding [70, 113] is orthogonal to the design of **PoE**. Hence, their use with **PoE** can reap further benefits.

### 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 two 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 complete. Due to speculative execution, PoE may require replicas to revert executed transactions, however. We envision PoE to be used at the core of blockchain databases, thereby increasing their throughput. To support this vision, we implemented PoE in our prototype blockchain database—ResilientDB—showing that PoE is not only safe and live, but also achieves up to 86% more throughput than existing state-of-the-art BFT protocols.

REFERENCES

[1] Michael Abadi-Malaeke, 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 Guestrin, Dahlia Malkhi, Lorenzo Oliosi, Ramakrishna Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. https://arxiv.org/abs/1702.01367

[3] Mohammad Jawad Amiri, Dhrayakant Agrawal, and Amr El Abbadi. 2019. CAP: A Cross-application Permissioned Blockchain. Proceedings of the VLDB Endowment 12, 11 (2019), 1385–1398. https://doi.org/10.14778/3342263.3342275

[4] Masoud Saeida Ardekani, Rayman Preet Singh, Nitin Agrawal, Douglas B. Terry, and Riza O. Sumanto. 2017. Rivotel: A Fault-tolerant Platform for Smart-home Applications. In Proceedings of the 18th ACM/FIPSE/USENIX Middleware Conference. ACM, 41–54. https://doi.org/10.5555/3159974.3159988

[5] Giuseppe Atteneise, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. 2014. Prata: A Scalable Byzantine Agreement Protocol. ACM Transactions on Computer Systems 32, 4 (2014), 1–34. https://doi.org/10.1145/2604376.2604423

[6] Iddo Bentov, Pavel Hubáček, Tal Moran, and Asaf Nadler. 2017. Tenderly. A Hybrid Byzantine Agreement Protocol for Blockchain. In Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing. ACM, 222–237. https://doi.org/10.1145/3064376.3064431

[7] Christian Decke, Jochen Seidel, and Roger Wattenhofer. 2016. Bitcoin Meets Strong Consistency. In Proceedings of the 17th International Conference on Distributed Computing and Networking. ACM, 13:1–13:10. https://doi.org/10.1145/2833132.2833321

[8] Byung-Goon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Cooperative Collection: Content Distribution and Proactive Recovery. In Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. USENIX Association, 153–168.

[9] Ittay Eyal and Emin Gün Sirer. 2018. Majority is Not Enough: Bitcoin Mining is Vulnerable. Commun. ACM 61, 8 (2018), 14–25. https://doi.org/10.1145/3200825.3200949

[10] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. Ph.D. Dissertation. Massachusetts Institute of Technology. http://hdl.handle.net/1721.1/68581

[11] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In Proceedings of the Third Symposium on Operating Systems Design and Implementation. USENIX Association, 173–186.

[12] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. ACM Transactions on Computer Systems 20, 4 (2002), 398–451. https://doi.org/10.1145/751637.751640

[13] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. 2003. BASE: Using Abstraction to Improve Fault Tolerance. ACM Transactions on Computer Systems 21, 3 (2003), 256–269. https://doi.org/10.1145/859716.859718

[14] Gregory Chockler, Dahlia Malkhi, and Michael K. Reiter. 2001. Backoff protocols for distributed mutual exclusion and ordering. In Proceedings 21st International Conference on Distributed Computing Systems. IEEE, 11–20. https://doi.org/10.1109/ICDCS.2001.918928
Itay Tsabary and Ittay Eyal. 2018. The Gap Game. In Symantec. 2018. Internet Security Threat Report, Volume 32. https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf

Pandian Raju, Soujanya Ponnapalli, Eyal Kaminzis, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. mLSM: Making Authenticated Storage Faster in Ethereum. In Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems. USENIX Association, 10–10.

Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Yonatan Sompolinsky, and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In Financial Cryptography and Data Security. Springer Berlin Heidelberg, 507–527. https://doi.org/10.1007/978-3-662-47854-7_33

Thamir M. Qadah and Mohammad Sadoghi. 2018. QueCC: A Queue-oriented, Conflict-free Concurrency Architecture. In Proceedings of the 19th International Middleware Conference. ACM, 13–25. https://doi.org/10.1145/3274088.3274810

Mohammad Sadoghi and Spyros Blanas. 2019. Transaction Processing on Modern Hardware. Morgan & Claypool. https://doi.org/10.2200/S00896ED1V01Y201901DM058

Marco Seragnini, Peter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. 2010. Scrout: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas. In 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN). IEEE, 353–362. https://doi.org/10.1109/DSN.2010.5544295

Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the Lines Between Blockchains and Database Systems: The Case of Hyperledger Fabric. In Proceedings of the 2019 International Conference on Management of Data. ACM, 105–122. https://doi.org/10.1145/3298986.3319883

Rong Shi and Yang Wang. 2016. Cheap and Available State Machine Replication. In Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference. USENIX Association, 265–279.

Atul Singh, Petros Maniatis, Peter Druschel, and Timothy Roscoe. 2007. Conflict-free Quorum-based BFT Protocols. Technical Report. Max Planck Institute for Software Systems. https://www.mpi-sws.org/tr/2007-001.pdf

Dale Skeen. 1982. A Quorum-Based Commit Protocol. Technical Report. Cornell University.

Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. 2018. SPECTRE: A Fast and Scalable Cryptocurrency Protocol. https://eprint.iacr.org/2016/1159

Yonatan Sompolinsky and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In Financial Cryptography and Data Security. Springer Berlin Heidelberg, 507–527. https://doi.org/10.1007/978-3-662-47854-7_32

Symantec. 2018. Internet Security Threat Report, Volume 32. https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf

Itay Tsabary and Ittay Eyal. 2018. The Gap Game. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 713–728. https://doi.org/10.1145/3243734.3243737

Amie Tsang. 2018. Bitcoin Plunges After Hacking of Exchange in Hong Kong. https://www.nytimes.com/2018/08/04/business/dealbook/bitcoin-bitfinex-hacked.html

Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. Comput. Surveys 47, 3, Article 42 (2015), 36 pages. https://doi.org/10.1145/1413976.1413985

Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Luc Cherk S PwC. 2016. Blockchain – an opportunity for energy producers and consumers? https://www.pwc.com/gx/en/industries/energy-utilities-resources/publications/opportunity-for-energy-producers.html

Harald Vranken. 2017. Sustainability of bitcoin and blockchains. Current Opinion in Environmental Sustainability 28 (2017), 1–9. https://doi.org/10.1016/j.cosust.2017.04.011

Jiaping Wang and Hao Wang. 2019. Monoxide: Scale out Blockchains with Asynchronous Consensus Zones. In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation. USENIX Association, 95–112.

Gavin Wood. 2016. Ethereum: a secure decentralised generalised transaction ledger. https://github.com/ethereum/paper.pdf EIP-150 revision.

Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Caccetta. 2011. ZZ and the Art of Practical BFT Execution. In Proceedings of the Sixth Conference on Computer Systems. ACM, 123–138. https://doi.org/10.1145/1966445.1966457

Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases. In Proceedings of the 2019 International Conference on Management of Data. ACM, 141–158. https://doi.org/10.1145/3299869.3300883

Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. ACM, 253–267. https://doi.org/10.1145/945445.945470

Maocon Yin, Dahila Malakib, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. ACM, 347–356. https://doi.org/10.1145/3296111.3331591

Kai Yuan, Yu-Bin Xia, Xue-Bao Chen, Bin-Yu Zang, and Jan Xie. 2018. ShadowEth: Private Smart Contract on Public Blockchain. Journal of Computer Science and Technology 33, 3 (2018), 542–556. https://doi.org/10.1007/s11399-018-1839-y

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